In diesem Kapitel:

Java Messaging Service JMS

In diesem Kapitel lernen Sie ein weiteres grundlegendes Konzept, das Messaging (Brokering), kennen. Das System ist auf der gleichen Ebene anzusiedeln wie CORBA oder RMI

1.1. Einführung, Zusammenfassung und Übersicht

1.1.1. Einführung

Java Messaging stellt eine vollständige Basis an Interfaces und wenigen Klassen für die kompatible Implementierung von Messaging Systemen zur Verfügung.

Was sind Messaging Systeme?
Was ist der Unterschied zu traditionellen Systemen wie CORBA, RPC oder RMI?
....
Auf diese Fragen wollen wir im Folgenden eingehen.

Zum Aufbau:

- zuerst die Übersicht
- dann gehen wir auf die Spezifikation ein
- schliesslich zeigen wir einige Anwendungsbeispiele mit Hilfe eines einfachen JMS Systems

Wesentlich ist folgender Punkt:
in der Regel haben wir bisher fix fertige Klassen besprochen. Jetzt besprechen wir fast ausschliesslich Interfaces, eben Spezifikationen!

Das Beispiel JMS System, mit dem die Beispiele implementiert sind, ist eine einfache Implementation dieser Interfaces.

Es gibt mehrere kommerzielle Anbieter von JMS Systemen, sie finden eine (heute noch) aktuelle Liste im Anhang.

Mit Hilfe eines JMS Systems kann man komplexe verteilte Applikationen entwickeln. Das JMS Konzept unterstützt Ereignis- basierte, Peer-to-Peer und traditionelle Client Server Applikationen.

Typische Anwendungsbereiche für JMS Systeme sind:
- Auftragsabwicklung
- Lagerverwaltung
- Finanz- und Rechnungswesen
- integrierte Logistiksysteme
- verteilte Decision Support Systeme

Moderne Anwendungen umfassen:
- verteilte Produktionssysteme
- Echtzeitdatenerfassung
- verteilte Datenbanken
- kollaborative Anwendungen (Arbeitsgruppen, Workflow)
- verteilte Kontrollsoftware

JMS gestattet unterschiedliche Architekturen, Betriebsysteme und in der Regel Portierbarkeit. Das Schlagwort im Umfeld von JMS und 'Advanced Messaging Systemen' ist: 'Enterprise Integration Architectur' EIA.

Zur Integration der einzelnen Applikationen müssen Informationen gesendet und empfangen werden und zwischen Prozessen kommuniziert werden.

Ziel ist es, dass JMS die allfällig nötigen Datenkonversionen zentral ausführen kann. Wesentlich ist aber das Sharen von Daten über mehrere Applikationen, auf dynamische Art und Weise.

Beispiele:
- verteilte Web Applikationen
- Integration bestehender Applikationen

JMS ermöglicht eine einfache Verbindung sonst getrennter Applikationen. Dabei muss man sich, wie in Java üblich, nicht um die Low Level Details kümmern. Das macht das System für einem. Man kann sich also ganz auf die Business Logik konzentrieren.

JMS Applikationen bestehen aus drei Hauptkomponenten:
- einem Kommunikationsprotokoll
  dieses beschreibt das Format zum Senden und Empfangen von Datenobjekten (Nachrichten, Messages, 'Meldungen') mit Hilfe eines physikalischen Netzwerkes
- ein Router Prozess
  der die Datenobjekte dem zuständigen Prozess zuliefert
- einer Klassenbibliothek
  die das API (Application Programming Interface) implementiert

Schauen wir uns kurz an, was wir darunter verstehen, bevor wir ins Detail, die Spezifikation eingehen. Wir befassen uns zuerst mit der generellen Frage : was sind Messaging Systeme? Dann diskutieren wir die grundlegenden Mechanismen (Data Distribution Architektur, Kommunikationsmodelle, synchrone und asynchrone Kommunikation, Messages und Transaktionen und Entwurfsmuster).

1.1.1.1. Messaging Systeme

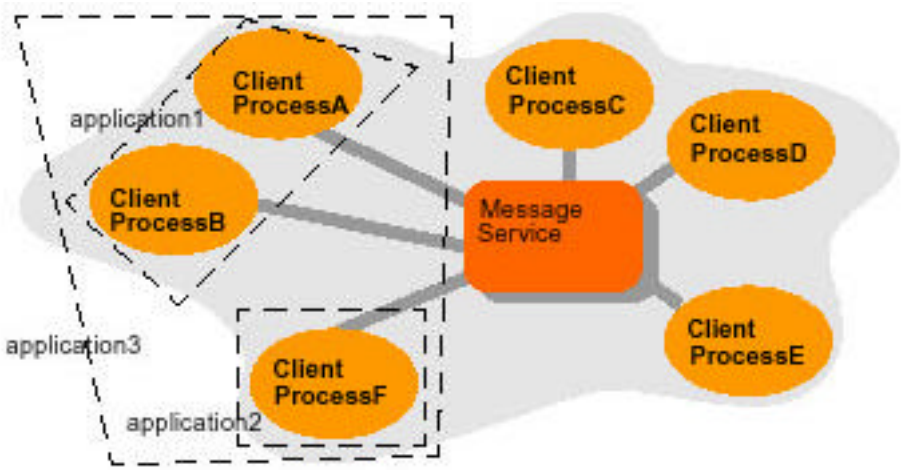
Ziel eines Messaging Systems ist es, Daten von einem Client zu empfangen, um sie an einen andern Client abzuliefern. Die Daten werden zusammen mit den Metadaten, den Daten über die Daten, in Form von Messages (Nachrichten, Meldungen) übertragen.

Den Begriff 'Messages' muss man dabei nicht im Sinne eines Telefonanrufes verstehen: es handelt sich um die Übermittlung von Daten und Objekten, in einer Form, die sowohl der Client- Sender als der Client- Empfänger verstehen können.

Das Messaging System besteht aus:

- einem Messaging Service - dieser Service umfasst
  den Message Server (auch Message Router oder Message Broker genannt),
  administrierte Objekte (typischerweise Topics und Warteschlangen, aber auch Channels)
  und ein Message Protokoll
- Clients
- Messages

Eine Client Applikation, oder schlicht Applikation, besteht aus einem oder mehreren Prozessen, welche bestimmte Funktionalitäten implementieren. Diese Prozesse können auf einer oder mehreren Maschinen verteilt sein und können selbst Teil einer grösseren Applikation sein.



Weil der Datentransfer zwischen der Frontend und der Backend Applikation im Vordergrund steht, spricht man auch von *Message-Oriented-Middleware (MOM).*

Middleware kann aus mehreren Layern bestehen, die jeweils aus unterschiedlichen Produkten bestehen. Bisher versuchten Hersteller jeweils unabhängige Messaging Produkte zu entwickeln, die natürlich nicht mit Konkurrenzprodukten kommunizieren konnten.

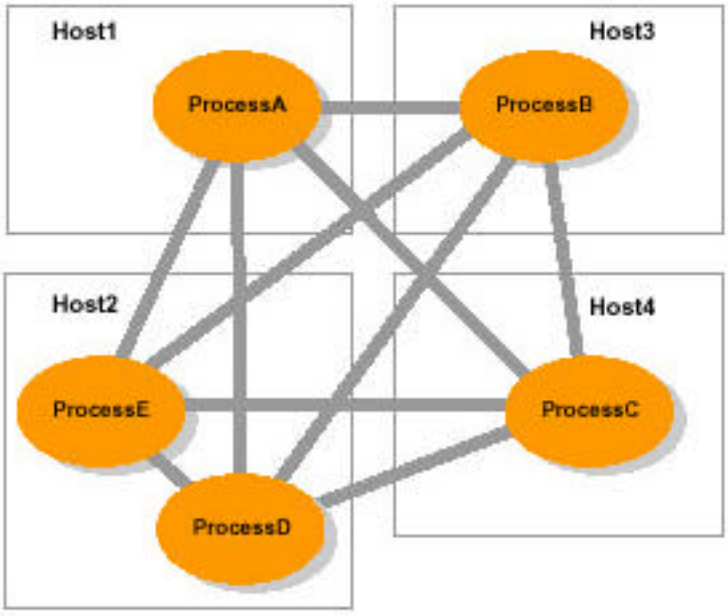
Analog zu CORBA hat Sun versucht /versucht Sun mit dem JMS Java Messaging Service eine offene, herstellerunabhängige Plattform zu definieren. Die Tatsache, dass diese Spezifikation bereits von vielen wichtigen Anbietern aufgenommen wurde (IBM, Oracle, Weblogic, Software AG, Sybase, Novell...) zeigt, dass die Spezifikation brauchbar ist, nicht zuletzt weil die Hersteller existierender Messaging Systeme am Standardisierungsprozess mitbeteiligt sind.

Ein *JMS Provider* ist ein Message-Service Produkt, welches die JMS Spezifikation und eventuell zusätzliche, Provider-spezifische Funktionen implementiert. Typische Funktionen können sein: Sicherheitsfunktionen, Administrationswerkzeuge, Load Balancing und vieles mehr. Der iBus von Softwared unterstützt zum Beispiel Multicasting und WAP; Weblogic unterstützt (zu mindestens gemäss dem Marketing) XML.... als Dokumenttyp.

1.1.1.2. Daten Distributions Architekture

1.1.1.2.1. Vollvernetzte Netzwerke

In einem voll vernetzten Netzwerk werden alle Prozesse direkt mit allen andern Prozessen verknüpft.



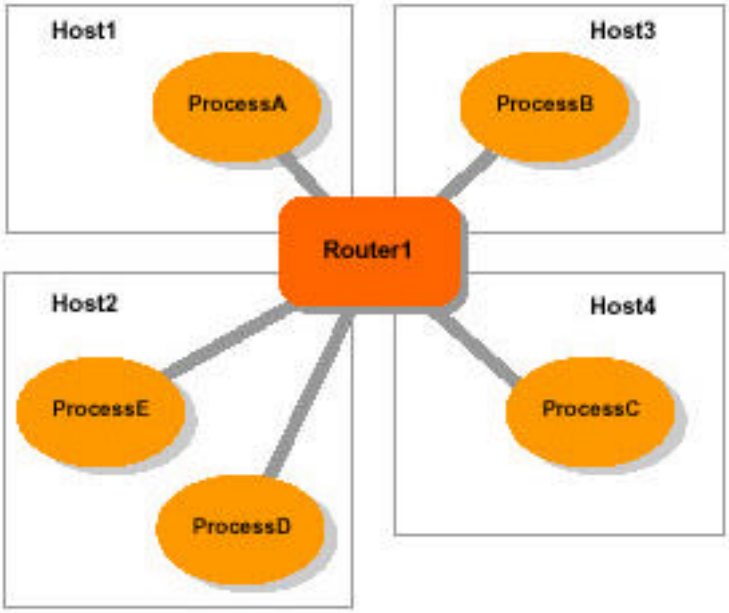
Da jeder Prozess eine Verbindung mit jedem andern Prozess hat, kann jeder Prozess Informationen von einem andern Prozess lesen oder schreiben.

Eine Applikation in einem solchen Umfeld muss also geöffnete Dateien für jeden andern Prozess besitzen.

Dies generiert einen entsprechenden Overhead, das System wird anfällig und ist komplex.

1.1.1.2.2. Virtuell vollverknüpfte Netzwerk

Java Message Systeme implementieren virtuelle, vollverknüpfte Netzwerke, wie in folgender Abbildung skizziert.



JMS nimmt der Applikation die Aufgabe ab, die Verbindungen der Prozesse untereinander zu regeln. Dies geschieht mit Hilfe eines eigenen speziellen Prozesses, dem *Router*, der die Verbindungen aufrecht erhält und managed.
Ein Anwendungsprogramm hat somit nur noch eine einzige Verbindung – mit dem Java Message Router.

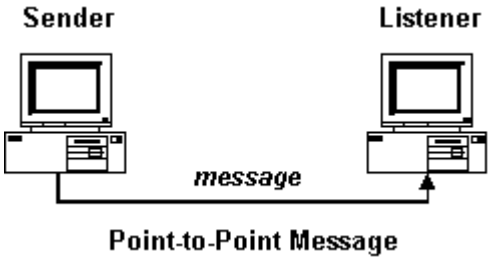
Damit reduziert sich die Komplexität und erhöht sich die Performance.

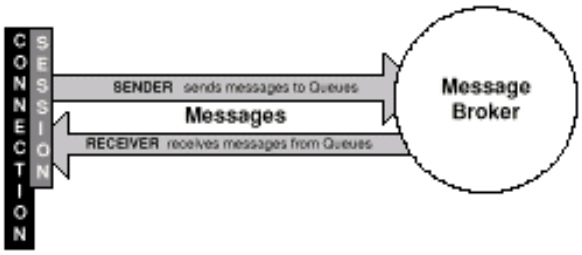
Alle Intra-Prozess und Inter-Prozesskommunikationen werden durch den Router abgehandelt.

1.1.1.3. Kommunikations Modelle

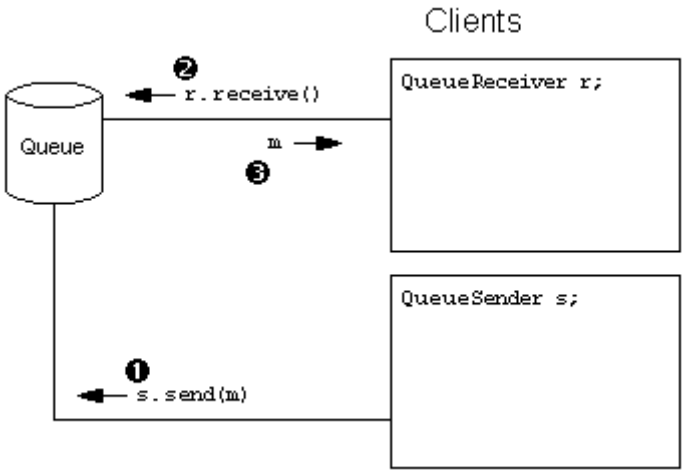
1.1.1.3.1. Punkt-zu-Punkt / Point-to-point

In diesem Kommunikationsmodell besitzt eine Message höchstens einen Empfänger. Der sendende Client (einer oder mehrere) adressieren die Message an eine Warteschlange *Queue,* welche die Nachrichten für den beabsichtigten (empfangenden) Client aufgewahrt. Eine Warteschlange ist also als Mailbox / Briefkasten zu verstehen.





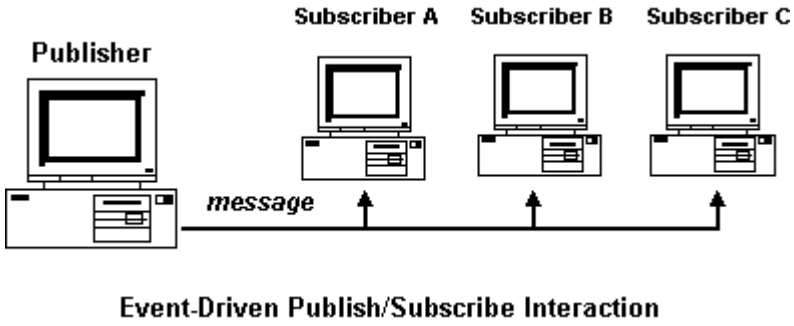
Viele Clients können Nachrichten an die Warteschlange senden;aber eine Nachricht kann nur von einem Client aus der Warteschlange gelesen werden. Und wie bei einem Briefkasten bleibt die Nachricht solange in der Warteschlange, bis sie entfernt wird. Damit kann der sendende Client Nachrichten selbst dann senden, wenn der empfangende Client nicht aktiv oder anderswo aktiv ist.



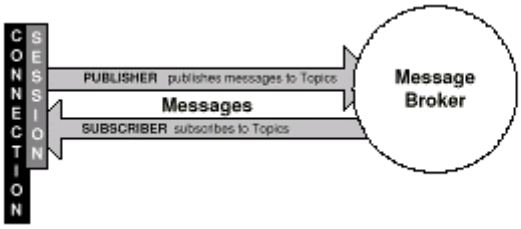
In einem Punkt-zu-Punkt System kann ein Client ein Sender (Message Producer), ein Empfänger (Message Consumer) oder beides sein.

1.1.1.3.2. Publish-and-subscribe

Im Gegensatz zum Punkt-zu-Punkt Modell gestattet das Pub/Sub Modell Nachrichten an mehrere Clients abzuliefern. Der sendende Client adressiert (publiziert) die Message an ein *Topic*. Mehrere Clients können sich bei einem Topic eintragen.

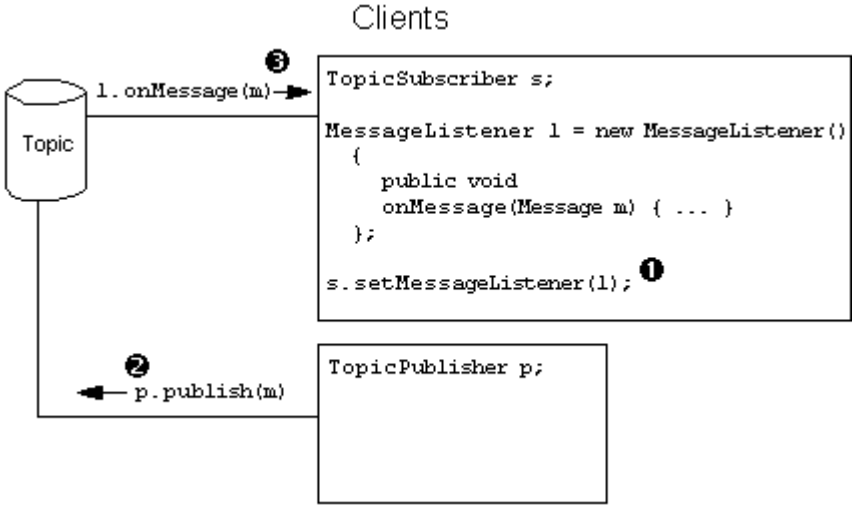


Event-Driven Publish/Subscribe Interaction



Eine *durable subscription (interest)*, eine dauerhafte Subskription existiert auch dann noch, wenn der Client inaktiv wird, zum Beispiel wegen Shutdown oder Restart. Während dieser Zeit werden die Nachrichten zwischengespeichert. Sie werden an den Client weitergeleitet, sobald dieser wieder aktiv ist.

In einem Pub/Sub System kann ein Client Publisher (Message Produzent) und / oder Subscriber (Message Konsument) sein oder beides gleichzeitig.



1.1.1.4. Synchrone und Asynchrone Kommunikation

1.1.1.4.1. Traditionelle Kommunikation

In einer *traditionellen synchronen* Kommunication muss der sendende und der empfangende Client die Kommunikationsaktivitäten selber koordinieren.

Damit eine gegebene Nachricht an einen Client geliefert werden kann, muss einer der Clients den andern zum Senden auffordern und einer den andern auffordern die Nachrichten zu empfangen. Die Kommunikationspartner sind solange beschäftigt, bis beide die Kommunikation abgeschlossen haben. Es findet ein sogenanntes 'Blocking' statt. Die Clients können nichts anderes machen als zu kommunizieren (mindestens der entsprechende Thread). Im schlimmsten Fall muss der sendende Client beliebig lange versuchen den empfangenden Client zu erreichen.

In der *traditionellen asynchronen* Kommunikation werden Nachrichten dann abgeliefert, wenn sie anfallen. Der empfangende Client muss also den Sender nicht zum Senden aufrufen. Dies geschieht ereignisgesteuert.

1.1.1.4.2. JMS Kommunikation

In der JMS Spezifikation wird das traditionelle end-to-end Kommunikationsmodell nicht verwendet. Daher müssen die zwei Kommunikationsmuster, die wir eben besprochen haben, modifiziert werden.

Der sendende Client ist immer asynchron *(asynchronous send*). Mit dem Senden an einen Message Server (Router, Broker) hat der sendende Client seine Arbeit getan.

Der empfangende Client kann entweder synchron oder asynchron sein. Der empfangende Client muss die Nachrichten beim Router abholen. Synchrones Empfangen *(synchronous receipt)* (per default) geschieht indem der Client die passende Receiver Methode für eine eintreffende Nachricht wählt oder periodisch die Verbindung auf neue Nachrichten überprüft (polling).

Im asynchronen Fall (*asynchronous receipt*) setzt der empfangende Client einen MessageListener auf. Eintreffende Nachrichten werden automatisch durch den Router abgeliefert, wobei der Router die `onMessage` Methode des MessageListener aufruft.

In allen Fällen ist der Router für die korrekte Ablieferung der Nachrichten zuständig. Der sendende Client kann dabei noch bestimmte Parameter setzen, zum Beispiel Liefergarantie, Time to Live, Quality of Service, was im Fehlerfall zu geschehen hat.

1.1.1.5. Administrierte Objekte

Administrierte Objekte sind Provider-spezifische Implementationen der JMS Interfaces. Diese Objekte werden vom Administrator des Messagingsystems kreiert und durch die Client Applikationen genutzt. Diese Objekte enthalten in der Regel Provider- spezifische Konfigurationsinformationen. Indem man Provider- spezifische Informationen in (administrierte) Objekte kapselt, bleibt der Client unabhängig von allen Details einer Provider- spezifischen Konfiguration. Zudem können diese Eigenschaften leichter mit Hilfe einer Adminstrationskonsole verwaltet werden.

Administratoren kreieren diese Objekte in einem Namensraum / namespace, in dem sie von den Client Applikationen nachgeschaut werden können, zum Beispiel mit Hilfe von Java Naming and Directory Interface (JNDI).

Ein standardisierter Namensraum erleichert die Portierung auf andere Systeme. Es wäre auch möglich diese Funktionalität direkt mit Provider- spezifischen Werkzeugen zu implementieren, aber eben mit dem Nachteil der reduzierten Portabilität.

JMS definiert zwei aministrierte Objekte:
- ConnectionFactory und
- Destination.

Diese Objekte können gleichzeitig von mehreren Clients benutzt werden. Beide Objekte gibt es auch in zwei Ausprägungen : für
- transaktionsorientierte und
- nicht transaktionsorientierte Sessions

Administrierte Objekte enthalten Konfigurationsparameter für den Verbindungsaufbau mit dem Messaging System des Providers.

Aus den Objekten ConnectionFactory und Destination Interfaces leitet JMS die folgenden Implementationen (administrierter Objekte) ab:
- QueueConnectionFactory
- TopicConnectionFactory
- Topic
- Queue

1.1.1.6. Messages

JMS verwendet einen *data-centered* Approach für die Distribution der Daten unter den Prozessen. Das heisst konkret, dass die Prozesse, welche Daten empfangen oder Senden wollen, den Datentypus angeben.

Andere Systeme sind in der Regel Prozess- zentriert. In diesen Systemen müssen die Kommunikationspartner die jeweilige ProzessID des Partners kennen.

Der Daten- zentrierte Approch gestattet eine bessere Kapselung der Informationen und damit eine bessere Strukturierung der Systeme.

Beispiel einer Daten- zentrierten Anwendung ist eine verteilte Datenbank. Alle Prozesse teilen sich die gemeinsamen Daten.

In Java Message Service Systemen werden diese Daten in der Form von Nachrichten ausgetauscht.

1.1.1.6.1. Struktur

In JMS wird festgelegt, dass eine Message aus folgenden Teilen besteht:

- Header
- Properties (Erweiterungen des Headers)
- Body

Ein Header wird immer verlangt. Er enthält Informationen für das Routing und die Identifikation (weitere Details finden Sie weiter unten).

Properties sind optional – sie enthalten zum Beispiel Informationen, mit deren Hilfe ein Client Nacrichten filtern kann.

Der Body ist auch zwingend erforderlich – er enthält die aktuellen Daten (Text, Objekte,...).

1.1.1.6.2. Header Fields

Einige Header Felder werden beim Senden, andere beim Empfangen und noch andere vom Provider gesetzt. Sie finden weiter unten in der Spezifikation von JMS die einzelnen Felder und deren Beschreibung.

1.1.1.6.3. Properties

Beschreibende Felder einer Nachricht bezeichnet man als Properties, Eigenschaften. Sie enthalten zusätzliche Informationen über die Daten, den Prozess, der sie kreiert hat ...

Die Syntax zur Spezifikation der Properties ist *property name: property value*.

Ein Client kann auch die Komunikation mit einer Warteschlange oder einem Topic optimieren, indem er passende Property Werte setzt, zum Beispiel Selektionskriterien.
Die Syntax zur Selektionsspezifikation lehnt sich an SQL an (siehe unten).

1.1.1.6.4. Transactions

Eine Session ist eine Sequenz von Messages, auf einer Verbindung zwischen Client und Provider gesendet und empfangen

Eine Session kann entweder als Transaktion oder als non-transacted default) definiert werden:

- eine *transacted* Session garantiert, dass eine Gruppe von Messages als Ganzes gesendet und empfangen wird, also atomar.
- eine *non-transacted* Session bedeutet, dass Messages individuell gesendet und empfangen werden

Ein Beispiel für eine transaktionsorientierte Session wäre ein Online Shopping:
der Kunde eröffnet eine Bestellung, fügt mehrere Artikel hinzu und beendet schliesslich seinen Einkauf. Die Auswahl eines Artikels ist eine Message. Der Abbruch der Bestellung geschieht mit der rollback Methode , die Transaktion *rolls back*.

Soweit die Übersicht und Einführung. Nun wollen wir uns mit JMS im Detail beschäftigen und anschliessend einige Beispiele anschauen. Doch nun zuerst die Theorie.

1.1.2. Zusammenfassung JMS

Die JMA (JMS Architektur) Spezifikation beschreibt die Zielsetzungen und Funktionen des Java Messaging Service JMS. JMS stellt eine Platform für Java Programme zur Verfügung, um Enterprise Messaging Systeme zu lesen, in diese Systeme zu schreiben und Nachrichtenzu kreieren.

1.1.3. Übersicht über JMS

Enterprise Messaging Producte (oder wie sie auch oft genannt werden: Message Oriented Middleware Producte) werden zunehmend wesentliche Komponenten für die Integration von intra-Company Operationen. Sie erlauben es, separate Geschäftsprozesse und Geschäftskomponenten flexibel und zuverlässig zu kombinieren.

Neben den traditionellen MOM Anbietern für Enterprise Messaging Produkte bieten zunehmend auch Datenbankanbieter und Internet orientierte Firmen MOM Produkte an. Java basierte Clients und Java basierte Middle Tier Services müssen in der Lage sein, mit diesen Message Systemen zu kommunizieren. JMS stellt eine allgemeine Basis dar, auf der Produkte aufbauen können, also ein allgemeines API. JMS besteht aus Interfaces und dazugehörender Semantik, welche definiert wie ein JMS Client die Funktionalität eines Enterprise Management Systems nutzen kann.

Da Messaging peer-to-peer ist, werden alle Benutzer eines JMS generisch als *Clients* bezeichnet.. Eine JMS *Applikation* besteht aus einem Set von applikationsspezifischen Nachrichtenund einem Set von Clients, die diese Nachrichtenaustauschen.

Produkte, welche JMS implementieren, tun dies, indem sie einen P*rovider* zur Verfügung stellen, welcher die JMS Interfaces implementiert.

1.1.3.1. Handelt es sich um ein Mail API?

Der Begriff *messaging* wird sehr breit definiert in der Informatik. Man verwendet den Begriff, um unterschiedliche Betriebsystemkonzepte zu beschreiben; darunter fallen email und Fax Systeme; und hier verwenden wir den Begriff, um asynchrone Kommunikation zwischen Unternehmensapplikationen zu beschreiben.

Messages, wie wir sie hier verstehen, sind asynchrone Requests, Reports oder Events, welche von Enterpriseapplikationen, nicht von Menschen konsumiert werden.

Sie enthalten vitale Information, welche benötigt wird, um diese Systeme zu koordinieren. Sie enthalten präzise formatierte Daten, welche spezifische Businessaktionen beschreiben. Durch Nachrichtsaustausch hält jede Applikation den Zustand und Fortschritt der unternehmensweiten Applikationen fest.

Eine typische Anwendung wäre die Verbindung eines Online Stores mit einer Enterprise Datenbankapplikation, also in etwa das, was man in einer Semesterarbeit bewerkstelligen sollte.

1.1.3.2. Existierende Messaging Systeme

Messaging Systeme sind peer-to-peer Facilities. Im Allgemeinen kann jeder Client Messages an Clients senden und von Clients empfangen. Jeder Client steht in Verbindung mit einem

Messageing Agent, der die nötigen Dienste / Facilities zur Verfügung stellt, um Nachrichten zu senden, zu empfangen oder zu kreieren.

Jedes System kennt seine spezielle Art der Nachrichtenadressierung. Jedes System generiert Nachrichten auf eine spezielle Art und Weise und speichert Daten in den Nachrichten, die dann übermittelt werden.

Einige Systeme können Nachrichten an viele Stationen broadcasten; andere beschränken sich auf eine bestimmte Destination. Einige Systeme stellen Dienste zur Verfügung, mit derenHilfe Nachrichten asynchron empfangen werden können (also die Nachrichten verarbeitet, in der Reihenfolge, in der sie eintreffen). Andere unterstützen lediglich synchrone Kommunikation (der Client muss jede Nachricht anfordern).

Jedes Messaging System stellt typischerweise Dienste zur Verfügung, die pro Nachricht gewählt werden können. Ein wichtiges Attribut ist zum Beispiel die Lebensdauer ("time-to-live") und der Aufwand, der getrieben wird, um die Nachricht abzuliefern. Dieser vairiiert zwischen "best effort" und garantierter Ablieferung der Nachricht / Nachrichten. Andere wichtige Attribute sind die Priorität, oder ob eine Antwort benötigt wird.

1.1.3.3. JMS Zielsetzungen

JMS kann nicht alle Aspekte aller bestehender Message Systeme berücksichtigen: das wäre zu komplex. Auf der andern Seite ist JMS mehr, als der gemeinsame Nenner existierender Messaging Systeme. JMS enthält Features, Funktionen, die benötigt werden, um unternehmenskritische Applikationen zu realisieren.

JMS definiert ein Set von Enterprise Messaging Konzepten und Diensten. Gleichzeitig wird versucht, die Anzahl Konzepte auf ein Minimum zu beschränken. Zudem wird die Portabilität der Applikationen in den Vordergrund gestellt.

1.1.3.4. JMS Provider

Wie bereits erwähnt, ist ein JMS Provider eine Implementation von JMS für ein Messaging Produkt. Idealerweise sind aus Sicht von Sun die JMS Provider 100% in Java geschrieben. Damit sind sie portabel und universell einsetzbar. Eines der Ziele von JMS ist es, den Implementationsaufwand auf ein Minimum zu begrenzen.

1.1.3.5. JMS Messages

JMS definieniert ein Set von Message Interfaces. Clients nutzen die Message Implementation, welche vom JMS Provider zur Verfügung gestellt wird. Eines der wichtigsten Ziele von JMS ist, dass Clients ein konsistentes API verwenden können, um mit Nachrichten zu arbeiten. Dieses API muss vom JMS Provider unabhängig sein..

1.1.3.6. JMS Domains

Messaging Produkte kann man grob einteilen in *point-to-point* und *publish-subscribe* Systeme.

"Point-to-point" (PTP) Produkte benutzen das Konzept der Message Queue.Jeder Nachricht wird an eine spezifische Warteschlange gesandt; Clients extrahieren Nachrichten aus der Warteschlange.

"Publish and subscribe" (Pub/Sub) Clients adressieren Nachrichten an einen Knoten in einer Inhaltshierarchie (content hierachie). Publisher und Subscriber sind in der Regel anonym und

können dynamisch an die Content Hierarchie publizieren. Das System übernimmt die Verteilung des Nachrichten, die an einem Knoten ankommen, an die Knoten/ Clients, die sich angemeldet / "subscribed" haben. JMS stellt Client Interfaces für die jeweiligen Domänen zur Verfügung.

1.1.3.7. Portabilität

Bezüglich Portabilität hat man sich das Ziel gesetzt, reine JMS Applikationen portabel zu gestalten. Obschon JMS Clients erlaubt wird, mit bestehenden Message Formate in heterogenen Umgebungen zu arbeiten, ist es im Allgemeinen nicht möglich, dieses Ziel zu erreichen, da zum Beispiel in Umgebungen, welche unterschiedliche Programmiersprachen verwenden, ausserhalb der Möglichkeiten eines JMS ist.

1.1.3.8. Was umfasst JMS nicht?

JMS kümmert sich nicht um folgende Funktionalität:

- Load Balancing / Fault Tolerance
  viele kommerzielle Produkte unterstützen kooperierende Clients bei der Implementierung von kritischen Diensten.JMS spezifiziert nicht, wie Clients zusammen arbeiten müssen, um einen einheitlichen Dienst anbieten zu können.

- Error/Advisory Notification
  viele Messaging Produkte definieren System- Nachrichten, welche asynchrone Notifikation über Probleme oder Systemereignisse für Clients anbieten. Falls man sich an die JMS Spezifikation hält, kann man auf diese Nachrichten weitestgehend verzichten und damit eine Portabilität erreichen, die man sonst eben nicht erzielen könnte.
  JMS versucht nicht, diese Nachrichtenzu standardisieren.

- Administration
  JMS definiert kein Management API für die Administration von Messagingprodukten.

- Security
  JMS spezifiziert kein API für die Kontrolle der Privacy und Integrität von Messages. Auch wie digitale Signaturen oder Keys verteilt werden, wird nicht spezifiziert. Es wird angenommen, dass diese Funktionalität durch andere Systeme gewährleistet wird.

- Wire Protocol
  JMS spezifiziert kein Messagingprotokoll / Protokoll.für die Message Übermittlung.

- Message Type Repository
  JMS enthält kein Repository für das Speichern von Message Typ Definitionen. JMS stellt auch keine Sprache zur Verfügung, mit deren Hilfe Messagetypen definiert werden könnten.

1.1.4. Was wird von JMS benötigt?

Alle JMS Provider müssen die in der JMS Spezifikation festgehaltene Funktionalität implementieren, in folgendem Sinne:

- Provider, die JMS Point-to-Point Funktionalität implementieren, brauchen keine Publish / Subscribe Funktionalität zu implementieren
- und umgekehrt.

1.1.5. Beziehung zu andern JavaSoft Enterprise APIs

1.1.5.1. JDBC

JMS Clients können JDBC einsetzen. Es sind viele Applikationen denkbar, welche beide APIs, JDBC und JMS in der selben Transaktion einsetzen. Dies wird in der Regel am einfachsten mit Hilfe von Java Enterprise Beans zu realisieren sein. Aber auch JTA (siehe unten)liefert eine Platform für solche gemischten Applikationen.

1.1.5.2. JavaBeans

JavaBeans können eine JMS Session benutzen, um Nachrichten zu senden und zu empfangen. Allerdings ist das JMS API nicht so entworfen worden, dass Java Beans dieses API direkt einsetzen können.

1.1.5.3. Enterprise Java Beans

JMS ist eine wichtige Resource für den EJB Komponenten Entwickler. EJBs können JMS zusammen mit JDBC einsetzen. Die jetzige Definition der EJBs sieht lediglich eine synchrone Kommunikation vor; eine der nächstes Versionen wird auch asynchrone Kommunikation gestatten und damit eine leichtere Integration mit JMS erlauben.

1.1.5.4. Java Transaction (JTA)

Das Java Transaction Package stellt ein Client API zur Verfügung, um verteilte Transaktionen auszuführen und an verteilten Transaktionen teilzunehmen.

Ein JMS Provider kann optional verteilte Transaktionen mit Hilfe von JTA unterstützen.

1.1.5.5. Java Transaction Service (JTS)

JMS kann zusammen mit JTS kombiniert werden, um verteilte Transaktionen, welche Nachrichtensenden und empfangen, mit Datenbank Updates und andern JTS Diensten zu kombinieren.

Auch solche Applikationen werden von EJBs unterstützt. EJBs sind also auch hier eine gute Platform.

1.1.5.6. Java Naming und Directory Service (JNDI)

JMS Clients benutzen JNDI, um JMS Objekte zu finden. Mit der Arbeitsteilung (JMS, JNDI) kann man eine bessere Portabilität erreichen, da die Dienste klar aufgeteilt werden können.

1.2. Architektur

1.2.1. Übersicht

Dieser Abschnitt beschreibt die Umgebung von Nachrichten-basierten Applikationen und welche Rolle JMS in diesem Umfeld spielen kann.

1.2.2. Was ist eine JMS Applikation

Eine JMS Applikation besteht aus folgenden Teilen:

- JMS Clients
  Dies sind Java Programme, welche Nachrichten senden und empfangen

- Nicht-JMS Clients
  diese Clients benutzen das API des Message Systems, also nicht JMS. Falls das System eine JMS Komponente besitzt, wird das System vermutlich JMS und fremde Clients besitzen.

- Messages
  Jede Applikation definiert bestimmte Nachrichten, die für die Kommunikation zwischen den Clients eingesetzt werden sollen.

- JMS Provider
  dies ist ein Messagingsystem, welches JMS implementiert. Zusätzlich werden andere administrative und Kontroll-Funktionen implementiert, die für den Betrieb des Messagingsystems benötigt werden.

- Administrierte Objekte
  administrierte Objekte sind JMS Objekte, welche von einem Administrator für den Einsatz durch Clients eingesetzt werden.

1.2.3. Administration

JMS setzt voraus, dass jeder JMS Provider unter Umständen wesentlich unterschiedliche Technologien einsetzen wird. Daher werden sich auch die Managementsysteme wesentlich unterscheiden. Damit JMS Clients portabel sind, muss daher vom konkreten Managementsystem abstrahiert werden.

Dies wird erreicht, indem man JMS administrierte Objekte definiert, welche vom Administrator des JMS Providers kreiert und administriert werden. Und später werden diese vom Client eingesetzt. Der Client verwendet diese Objekte mit Hilfe von JMS API Schnittstellen, welche portabel sind, während der Administrator die Werkzeuge des Providers benutzt, um die administrierten Objekte zu kreieren.

Es gibt zwei Typen von JMS administrierten Objekten:

- ConnectionFactory
  ein Objekt, welches der Client benutzen kann, um eine Verbindung aufzubauen.
- Destination
  damit kann der Client das Ziel der Nachrichts angeben und die Quelle der Nachricht.

Administrierte Objekte werden vom Administrator in ein JNDI (Java Naming and Directory Interface) Namespace eingetragen.

Der JMS Client kennt typischerweise die JMS administrierten Objekte, die er benötigt, und wie der Client zum JNDI Namen dieser Objekte gelangt.

1.2.4. Zwei Messaging Style

Eine JMS Applikation verwendet entweder den "point-to-point" (PTP) oder die "publish-and-subscribe" (Pub/Sub) Messaging-Style. In einer Applikation können diese Stylformen auch kombiniert werden. JMS fokusiert sich aber auf Applikationen, die eine der beiden Formen bevorzugt.

JMS definiert diese zwei Messaging-Style, weil es die am verbreitesten sind. Da aber die meisten Systeme nur die eine oder die andere Form unterstützen, kennt JMS beide und definiert Kriterien, unter denen beide miteinander zusammen arbeiten können.

1.2.5. JMS Interfaces

JMS basiert auf einem Set von Messaging Konzepten. Jede Messaging Domäne - PTP oder Pub/Sub - definieren ihre eigenen Interfaces, speziell für diese Konzepte.

JMS Parent	PTP Domain	Pub/Sub Domain
`ConnectionFactory`	`QueueConnectionFactory`	`TopicConnection Factory`
`Connection`	`QueueConnection`	`TopicConnection`
`Destination`	`Queue`	`Topic`
`Session`	`QueueSession`	`TopicSession`
`MessageProducer`	`QueueSender`	`TopicPublisher`
`MessageConsumer`	`QueueReceiver, QueueBrowser`	`TopicSubscriber`

Im folgenden definieren wir diese Konzepte in einer ersten Phase etwas vertieft. Später schauen wir uns Beispiele an und lernen diese Konzepte genauer und konkreter kennen.

- **ConnectionFactory**
  ein administriertes Objekt, welches vom Client benutzt wird, um eine Verbindung aufzubauen.
- **Connection**
  eine aktive Verbindung zu einem JMS Provider
- **Destination**
  kapselt die Identität des Nachrichtsempfängers
- **Session**
  ein Contextthread, der Nachrichten senden und empfangen kann
- **MessageProducer**
  ein Objekt, welches von einer Session kreiert wurde und eingesetzt wird, um Nachrichten an eine Zieladresse zu senden.
- **MessageConsumer**
  ein Objekt, welches durch eine Session kreiert wurde und eingesetzt wird, um Nachrichten zu empfangen, die an die Destination gesandt wurden.

Der Begriff *consume / konsumieren* bedeutet, dass Nachrichten empfangen werden können, vom JMS Client. Der JMS Provider hat eine Nachricht empfangen und sie an einen seiner Clients weitergereicht. Da JMS synchrones und asynchrones Empfangen von Nachrichten unterstützt, wird der Begriff *konsumieren* / *consume* verwendet, falls eine Unterscheidung der beiden Techniken unwesentlich ist.

Analog wird *produce / produzieren* als der allgemeine Begriff verwendet, wenn es um das Senden von Nachrichten geht. Senden einer Nachricht heisst, dass eine Nachricht an einen JMS Provider übergeben wird. Der JMS Provider ist dann für die Ablieferung der Nachricht an die Destination zuständig.

1.2.6. Entwicklen einer JMS Applikation

Grob gesprochen besteht eine JMS Applikation aus einem oder mehreren Clients, die untereinander Messages austauschen. Die Applikation kann auch nicht-JMS basierte Clients umfassen. Diese Clients verwenden das JMS API oder den JMS Provider.

Eine JMS Applikation kann man als Einheit ansehen. Man kann also eine JMS Applikation zu einer bestehenden Applikation "hinzufügen".

1.2.6.1. Entwicklung eines JMS Client

Ein typischer JMS Client durchläuft folgende Setup Prozedur:

- Einsatz von JNDI (Name & Directory) , um ein ConnectionFactory Objekt zu finden
- Einsatz von JNDI, um ein oder mehrere Destination Objekte zu finden
- Einsatz der ConnectionFactory, um eine JMS Verbindung, ein Connection Objekt, herzustallen
- Einsatz des Connection Objektes, um eine oder mehrere JMS Sessions zu kreieren
- Einsatz einer Session und der Destinations, um benötigte MessageProducers und MessageConsumers zu kreieren
- Mitteilung an die Connection, die Message Delivery zu starten.

Damit ist das System bereit, Nachrichten zu senden und zu empfangen.

1.2.7. Security

JMS stellt keine Mechanismen zur Verfügung, um die Integrität und Privacy der Nachrichten zu kontrollieren und zu konfigurieren.

.

Es wird vorausgesetzt, dass viele JMS Provider diese Fähigkeiten einbaun werden oder ein Server Administrationstool dafür eingesetzt wird.

Der Client wird die korrekte Sicherheitsinformationen von den Objekten erhalten, mit denen er zusammenarbeitet..

1.2.8. Multi-Threading

JMS ist nicht von Grund auf multithreadingfähig ausgelegt, weil dadurch ein bestimmter Overhead und eine bestimmte Komplexität eingebaut worden wäre. Das aktuelle JMS Design beschränkt die Mehruserfähigkeit auf jene Teile,die natürlicherweise von mehreren Clients benutzt werden

.

JMS Object	**Unterstützt Concurrent Use**
`Destination`	ja
`ConnectionFactory`	ja
`Connection`	ja
`Session`	nein
`MessageProducer`	nein
`MessageConsumer`	nein

JMS definiert einige spezifische Regeln, welche die gleichzeitige Verwendung von Sesions einschränken. Es gibt zwei Gründe, den gleichzeitigen Zugriff aus Sessions einzuschränken:

1. eine Session ist eine JMS Entität, die als Transaktion aufgefasst werden kann, also atomar sein sollte. Es ist sehr schwierig, Transaktionen zu implementireien, welche echte Concurrency unterstützen.
2. Sessions unterstützen asynchronen Message Konsum. Es ist wichtig, dass JMS *nicht* verlangt, dass der Client Code zum asynchronen Lesen von Messages so ausgelegt wird, dass gleichzeitig mehrere Nachrichten asynchron gelesen werden können. Falls eine Session mehrere asynchrone Konsumenten unterstützt, ist es wichtig, dass die Clients nicht gezwungen sind, gleichzeitig aktiv zu sein.

Diese Einschränkungen vereinfachen den Einsatz von JMS für eine typischen Client. Falls Concurrency benötigt wird, kann man einfach mehrere Sessions verwenden.

1.2.9. Triggering von Clients

Es gibt Clients, die so designed sind, dass sie periodisch aufwachen und Nachrichten verarbeiten, die auf sie warten. Solche Systeme lassen sich leicht mit Message-basierten Mechanismen realisieren. Der Trigger ist typischerweise ein Schwellenwert, zum Beispiel die Anzahl wartender Messages.

JMS selber stellt diesen Mechanismus nicht zur Verfügung. Kommerzielle Produkte zum Teil schon.

1.2.10. Request/Reply

JMS stellt das *JMSReplyTo* Message Header Feld für die Spezifikation einer Destination für die Message zur Verfügung.

Das *JMSCorrelationID* Header Feld der Antwort, kann als Referenz zur Originalnachricht verwendet werden. Die Details des Message Aufbaus besprechen wir später.

JMS kann auch temporäre Warteschlangen zur Verfügung stellen, welche als Ziel von Nachrichten verwendet werden können.

Es gibt viele unterschiedliche Formen des request/reply Patterns in Enterprise Messaging Produkten:
von einem einfachen "eine Nachricht verlangt eine Antwort" bis zu "eine Nachricht verlangt Feedback Nachrichten von vielen Kommunikationspartnern".

Daher hat man sich bei der Architektur von JMS darauf beschränkt, die grundlegenden Mechanismen zur Verfügung zu stellen. Darauf aufbauend kann man komplexere Mechanismen konstruieren und definieren.

JMS definiert *request/reply* Helper Klassen (Klassen, welche JMS verwenden)für beide Domänen - PTP und Pub/Sub - welche die grundsätzliche Form des request/reply implementieren.

JMS Provider und Clients können spezialisierte Implementationen zur Verfügung stellen.

1.3. Das JMS Message Modell

1.3.1. Hintergrund

Enterprise Messaging Produkte behandeln Messages als leichtgewichtige Entitäten, welche aus einem Header und einem Messagebody bestehen.

Der Header besteht aus Feldern, mit deren Hilfe die Messages identifiziert und gerouted werden.

Der Body enthält die Daten, die von der Applikation versandt werden müssen.

Im Rahmen dieser allgemeinen Form sieht man bei vielen der bekannten Messaging Produkte Varianten und Variationen. Vorallem im Body gibt es sehr unterschiedliche Darstellungen; aber auch beim Header sieht man Unterschiede betreffend Inhalt und Semantik.

Einige Produkte verwenden klare Verfahren für die Kodierung der Nachrichtsdaten; andere sind eher nicht transparent. Einige Messaging Produkte verwenden Repositories zum Speichern von Message Beschreibungen.

Daher muss das JMS Modell möglicht generell sein, um alle denkbaren Fälle abdecken zu können.

1.3.2. Ziele von JMS

Das JMS Message Model definiert folgende Ziele:

- ein einheitliches, vereinheitlichtes Message API zur Verfügung stellen
- das API sollte die gängigen Message Formate unterstützen
- Unterstützung von heterogenen Applikationen, welche unterschiedliche Betriebsysteme, Maschinenarchitekturen und Programmiersprachen
- Unterstützung von Nachrichten, welche ganze Java Objekte enthalten.
- Unterstützung von Messages, welche Extensible Markup Language Seiten (siehe http://www.w3.org/XML) enthalten.

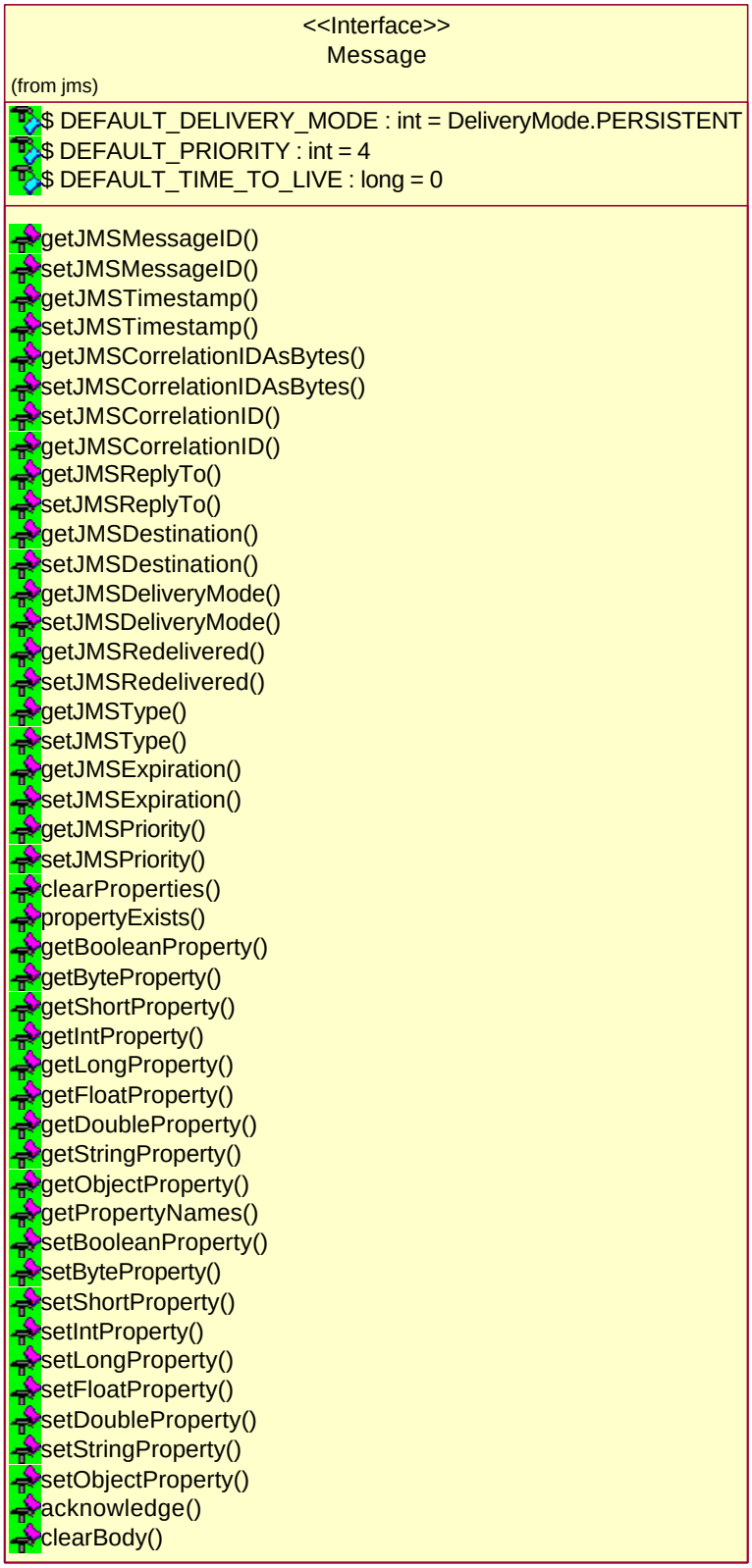
1.3.3. JMS Messages

JMS Messages bestehen aus den folgenden Bestandteilen:

- **Heade**r - Alle Messages unterstützen eine bestimmte Menge von Header Feldern. Header Felder enthalten Werte, die vom Client und Provider für das Routinh benötigt werden..
- **Properties** - Zuätzlich zu den Standard Headerfeldern kann ein Header weitere optionale Headerfelder aufweisen.
  - applikationsspezifische Eigenschaften - mit diesem Mechanismus kann man spezielle, zusätzliche Headerfelder definieren, also die Struktur des Headers wesentlich ändern und erweitern.
  - Standard Eigenschaften - JMS definiert einige Standard Properties, die in Wahrheit auch wieder optionale Header Felder sind..
  - Provider-spezifische Eigenschaften - Die Integration eines JMS Client mit einem JMS Provider Client kann es nötig machen, Provider-spezifische Properties zu definieren. JMS definiert eine Namenskonvention für diese.
- **Body** - JMS definiert mehrere Message Body Typen, mit deren Hilfe die gängigen Messages beschrieben werden können, welche es zur Zeit gibt.

1.3.4. Message Header Felder

In den folgenden Abschnitten beschreiben wir die unterschiedlichen Message Header Felder.



Ein Message Header wird als Ganzes an den Client übermittelt, sofern es sich im einen JMS Client handelt. Für nicht-JMS Clients werden die Header Felder nicht definiert.

1.3.4.1. JMSDestination

Das *JMSDestination* Header Feld enthält die Destination, zu der die Nachricht gesandt werden soll.

1.3.4.2. JMSDeliveryMode

Das *JMSDeliveryMode* Header Feld enthält den Delivery Modus, welcher beim Senden der Nachricht spezifiziert wird.

1.3.4.3. JMSMessageID

Das *JMSMessageID* Header Feld enthält eine eindeutige Identifikation der Message.

Eine *JMSMessageID* ist eine Zeichenkette *String* , die als eindeutiger Index in einem Historyfile dient.

Inwiefern der Index eindeutig ist, hängt vom konkreten Produkt ab. Aber idealerweise muss die Eindeutigkeit unternehmensweit gewährleistet sein.

Alle *JMSMessageID* Werte starten mit dem Präfix ‘ID:’.

Der JMS *MessageProducer* kann auch die Message ID unterbinden. Dies besagt, dass die Nachricht nicht von ihrer MessageID abhängig ist..

1.3.4.4. JMSTimestamp

Das *JMSTimestamp* Header Feld enthält die Zeit, zu der eine Nachricht an einen Provider zum Versenden übergeben wurde. Es ist also nicht der Zeitpunkt, zu dem die Nachricht aktuell versandt wurde.

1.3.4.5. JMSCorrelationID

Ein Client kann das *JMSCorrelationID* Header Feld benutzen, um Nachrichten miteinander zu verbinden. Eine typische Anwendung ist die Verbindung einer AntwortsNachricht mit der AnfrageNachricht. *JMSCorrelationID* kann folgende Informationen enthalten:

- eine Provider-spezifische Message ID
- eine Anwendungs-spezifische Zeichenkette *String*
- ein Provider-spezifischer *byte[]* Wert

Da alle Messages, welche von JMS versandt werden, eine eindeutige Message ID besitzen, ist dies ein idealer Weg, die einzelnen Messages miteinander zu verknüpfen. Alle Message ID's müssen mit dem Präfix ‘ID:’ starten.

Es gibt aber auch Beispiele, in denen ein Applikation-spezifischer Wert zum Linken der Nachrichten benutzt wird. Zum Beispiel kann eine Applikation das Feld *JMSCorrelationID* benutzen, um irgend eine externe Referenz aufzunehmen. Diese Werte müssen dann allerdings nicht mit dem Präfix ‘ID:’ starten; dieser Präfix ist eigentlich reserviert für Provider-generierte Message ID Werte.

1.3.4.6. JMSReplyTo

Das *JMSReplyTo* Header Feld enthält eine Destination, welche vom Client zur Verfügung gestellt wird, wann immer er eine Nachricht verschickt.

Messages ohne *JMSReplyTo* sind zum Beispiel einfache Bestätigungen an den Provider. Messages mit *JMSReplyTo* Wert erwarten typischerweise eine Antwort. Diese kann auch optional sein.

1.3.4.7. JMSRedelivered

Falls der Client eine Message erhält, bei der der *JMSRedelivered* Indikator gesetzt ist, hat der Client vermutlich bereits eine Nachricht erhalten, die er aber nicht bestätigt hat.

1.3.4.8. JMSType

Das *JMSType* Header Feld enthält einen Messagetypidentifier, der vom Client gesetzt wird. Das *type* Header Feld kann auch eine Verknüpfung zu einem Eintrag in den Repository des Providers enthalten.

JMS definiert keinen Standard Message Definition Repository oder eine Namensgebungsregel.

1.3.4.9. JMSExpiration

Beim Senden einer Nachricht werden die jeweiligen time-to-live Werte addiert. Bei Antworten wird das *JMSExpiration* Header Feld mit diesem Wert überschrieben. Falls dieser Wert null ist, gibt es kein Expiration Date. Falls die Lebensdauer einer Nachricht abgelaufen ist, wird sie zerstört.

1.3.4.10. JMSPriority

Das *JMSPriority* Header Feld enthält die Priorität einer Nachricht.

JMS definiert zehn Prioritätslevel mit 0 als der tiefsten Priorität und 9 als der höchsten.
Die Prioritäten 0-4 sind zusammengefasst zu *normal;*
Prioritäten 5-9 gehören zur *expedited* Priorität.

JMS verlangt von keinem Provider, dass er diese Prioritäten implementiert; aber JMS versucht Messages mit höherer Priorität vor jenen mit tiefer Priorität abzuliefern.

1.3.4.11. Wie werden Message Header Werte gesetzt?

Header Feld	**gesetzt von**
JMSDestination	Send Methode
JMSDeliveryMode	Send Methode
JMSExpiration	Send Methode
JMSPriority	Send Methode
JMSMessageID	Send Methode
JMSTimestamp	Send Methode
JMSCorrelationID	Client
JMSReplyTo	Client
JMSType	Client
JMSRedelivered	Provider

1.3.4.12. Überschreiben der Message Header Felder

JMS erlaubt es einem Administrator JMS so zu konfigurieren, dass die Client- seitig spezifizierten Felder *JMSDeliveryMod*e, *JMSExpiration* und *JMSPriorit*y überschrieben werden können.

Falls dies getan wird, muss der Administrator wissen, was wo steht, er hat freie Hand, JMS macht ihm keine Vorschriften.

1.3.5. Message Eigenschaften

Zusätzlich zum Headerfeld enthalten Messages die Möglichkeit Eigenschaftswerte (Properties) zu bestimmen, zu setzen. Damit kann man auch die Werte von optionalen Feldern des Headers spezifizieren.

1.3.5.1. Property Namen

Property Namen müssen bestimmten Syntaxregeln gehorchen. Die genaue Syntax wird weiter unten im Detail beschrieben.

1.3.5.2. Property Werte

Die Property Werte können sein: *boolea*n, *byt*e, *shor*t, *in*t, *lon*g, *floa*t, *doubl*e und *Strin*g.

1.3.5.3. Properties benutzen

Property Werte werden vor dem Senden der Message gesetzt. Beim Empfangen der Message durch den Client sind die Properties read-only. Falls der Client versucht, die Werte zu verändern, wird eine *MessageNotWriteableException* geworfen.

Obschon JMS nicht definiert, was eine Eigenschaft / Property sein darf, sollte man sich bewusst sein, dass die Bearbeitung von Nachrichten(-Rümpfen) wesentlich effizienter ist, als die Bearbeitung der Properties / die Bearbeitung der Nachricht mit Hilfe der Properties. Die beste Performance erreicht man, wenn man Proerties nur zur Bearbeitung der Message Header einsetzt.

1.3.5.4. Property Wertkonversion

Properties unterstützt auch eine Konversion, gemäss der folgenden Konversionstabelle. Alle nicht unterstützten Konversionen werfen eine JMS *MessageFormatExceptio*n.
Die *String* zu numerisch Konversionen werfen die *java.lang.NumberFormatException* falls der numerische Wert *valueOf()* den *String* nicht akzeptiert.

	boolean	**byte**	**short**	**int**	**long**	**float**	**double**	**String**
boolean	X							X
byte		X	X	X	X			X
short			X	X	X			X
int				X	X			X
long					X			X
float						X	X	X
double							X	X
String	X	X	X	X	X	X	X	X

Tabelle 1 Ein Wert vom Zeilentyp kann in einen Wert vom Spaltentyp konvertiert werden

1.3.5.5. Property Werte als Objekte

Zusätzlich zu den typenspezifischen set/get Methoden für Properties, stellt JMS auch objektbasierte Methoden zur Verfügung:

- *setObjectProperty*
- *getObjectProperty*

Und wie bei Properties üblich, ist der Grund für diese Methode der Wunsch Eigenschaften der Programme erst zur Laufzeit zu spezifizieren.

Die *setObjectProperty* Methode akzeptiert *Boolea*n, *Byt*e, *Sho*rt, *Integ*er, *Lon*g, *Floa*t, *Double* und *Strin*g. Andere Klassen als Parameter werfen die JMS *MessageFormatExceptio*n.

Die *getObjectProperty* Methode liefert Werte vom Typ *null*, *Boolea*n, *Byt*e, *Sho*rt, *Integ*er, *Lon*g, *Floa*t, *Double* and *Strin*g. Ein *null* Wert wird zurück gegeben, falls diese Eigenschaft nicht existiert.

1.3.5.6. Property Iteration

Die Reihenfolge der Properties ist nicht definiert. Mit der Methode *getPropertyNames* wird eine Liste der Eigenschaftsnamen erstellt. Anschliessend können die Eigenschaften mit Hilfe verschiedener Methoden gelesen werden.

1.3.5.7. Zurücksetzen eines Property Wertes einer Message

Die Methode *clearProperties* löscht der Wert der Eigenschaft.

1.3.5.8. Nicht-existierende Properties

Falls versucht wird, ein Wert zu lesen, der nicht gesetzt wurde, wird ein null Wert zurück geliefert.

1.3.5.9. JMS definierte Properties

JMS reserviert den ‘JMSX’ Property Namen Präfix für JMS definierte Properties. Unten sind die Properties aufgelistet. Neue JMS Properties können später noch hinzu kommen.

Die Methode
*String[] ConnectionMetaData.**getJMSXPropertyNames()***

liefert die Namen der JMSX Properties, welche durch eine Connection unterstützt werden.

Name	Typ	gesetzt von	Use
`JMSXUserID`	`String`	Provider beim Senden	Die Identität des Message Senders
`JMSXAppID`	`String`	Provider beim Senden	Die Identität des Message Senders
`JMSXDeliveryCount`	`int`	Provider beim Empfangen	Anzahl Versuche, die Nachrichten abzuliefern; der erste Versuch ist 1, der zweite 2,...
`JMSXGroupID`	`String`	Client	Die Idenntität der Message Gruppe zu der diese Nachricht gehört
`JMSXGroupSeq`	`int`	Client	Die Sequenznummer der Nachricht in der Gruppe: die erste ist 1, die zweite 2,...
`JMSXProducerTXID`	`String`	Provider beim Senden	Der Transaktions Identifier der Transaktion, im Rahmen derer die Message produziert wurde.
`JMSXConsumerTXID`	`String`	Provider beim Empfangen	Der Transaktions Identifier der Transaktion, im Rahmen derer die Message konsumiert wurde.
`JMSXRcvTimestamp`	`long`	Provider beim Empfangen	Der Zeitpunkz, zu dem JMS die Nachricht an den Konsumenten abgeliefert hat.
JMSXState	int	Provider	Angenommen es existiert ein Message Warehouse, welches eine Kopie aller Messages enthält, die an die Konsumenten gesandt wurden Jede Kopie ist in einem der folgenden Zustände: 1(waiting), 2(ready), 3(expired) oder 4(retained) Da der Zustand für den Konsumenten und Produzenten unwichtig ist, wird der Zustand beiden nicht mitgeteilt. Der Zustand ist nur für das Message Warehouse relevant.

Tabelle 2

Alle JMSX Properties, welche beim Senden gesetzt werden, sind sowohl dem Provider als auch dem Konsumenten zugänglich.
JSMX Properties, welche beim Empfangen gesetzt werden, stehen logischerweise nur dem Konsumenten zur Verfügung.

JMSXGroupID und *JMSXGroupSeq* sind Standard Properties, welche Clients verwenden sollten, falls sie Messages gruppieren möchten. Alle Provider müssen diese unterstützen.

1.3.5.10. Provider-spezifische Properties

JMS reserviert die 'JMS_<Anbieter_Name>' Property Namespräfix für Provider-spezifische Properties. Jeder Provider definiert seinen <Anbieter_Name>.Damit kann ein Anbieter seine speziellen Dienste pro Message einem JMS Client anbieten.

Die Provider-spezifischen Properties stellen Erweiterungen dar, die von einzelnen Anbietern angeboten werden. Diese sollten also nicht für JMS zu JMS Messaging eingesetzt werden.

1.3.6. Message Acknowledgment

Alle JMS Messages unterstützen die *acknowledge* Method für den Fall, dass ein Client spezifiziert, dass die Nachrichten eines JMS Consumer's explizit bestätigt werden müssen.. Ein Client kann auch automatisches Bestätigen verwenden. In diesem Fall werden die Acknowledge Aufrufe ignoriert.

1.3.7. Das Message Interface

Das *Message* Interface ist das Root Interface für alle JMS Messages. Es definiert die JMS Message Header Felder, Property Möglichkeiten und die Acknowledge Methode, welche für alle Messages benutzt wird.

1.3.8. Message Selektion

Viele Messaging Applilationen müssen die Nachrichten, die sie produzieren, filtern und in Kategorien einteilen.

Falls eine Nachricht nur an einen Client adressiert ist, lässt sich dies einfach bewerkstelligen, indem zum Beispiel die Kriterien in die Message integrieren. Der Empfänger kann dann alle Nachrichten, die seine Kriterien nicht erfüllen, einfach ignorieren.

Falls eine Message an viele Kunden versandt wird, bietet es sich an, die Kriterien im Header aufzuführen, so dass sie dem JMS Provider sichtbar sind und dieser entsprechend selektiv versenden kann.

JMS verfügt über Mechanismen, die es einem Client erlauben, die Message Selektion an den JMS Provider zu delegieren. Dadurch kannn das Kommunikationsvolumen reduziert werden. Applikationsspezifische Auswahlkriterien werden als Properties den Nachrichten angehängt; Client-spezifische Auswahlkriterien werden mit Hilfe von JMS *message selector* Ausdrücken spezifiziert.

1.3.8.1. Message Selector

Ein JMS Message Selector erlaubt es einem Client mit Hilfe des Message Headers zu spezifizieren, an welchen Messages er interessiert ist.

Nur Nachrichten, deren Header und Properties dem Selektor entsprechen, werden abgeliefert. Dabei muss der Begriff "*nicht abgeliefert*" geklärt werden:
es werden nur jene Nachrichten akzeptiert, bei denen die Header und Property Felder mit jenen des Selektors identisch sind.

1.3.8.1.1. Message Selector Syntax

Der Message Selektor ist eine Zeichenkette *String*, deren Syntax auf einem Subset von SQL92[1] *Conditional Expression Syntax* basiert:

Der Message Selektor wird von links nach rechts mit Präzedenzregeln aufgelöst. Mit Hilfe von Klammern kann die Reihenfolge angepasst werden.

Vordefinierte Selektor Literale und Operatorennamen werden in Grossbuchstaben geschrieben, obschon sie nicht sensitiv sein sollen (non case sensitive).

Ein Selektor kann enthalten:

- Literale:
- ein Zeichenkettenliteral wird in einfachen Anführungszeichen eingeschlossen; ein Anführungszeichen wird als Doppelanführungszeichen geschrieben.
  Die Syntax ist also identisch mit der Notation in Java:
  'Zeichenkette', 'Hans'' Auto' (entspricht: Hans' Auto)
  - ein exaktes numerisches Literal ist ein numerischer Wert ohne Dezimalpunkt, wie zum Beispiel 57, -957, +62; die Java *long* Daten werden unterstützt und folgen der Java Integer Syntax.
  - ein approximiertes numerisches Literal ist ein numerischer Wert in wissenschaftlicher Notation wie zum Beispiel 7E3, -57.9E2 oder ein numerischer Wert mit einem Dezimalpunkt wie 7., -95.7, +6.2; der Wertebereich entspricht jenem von Java *double*. Approximierte Literale verwenden die Java floating Point Syntax.
  - die boolean Literale sind *TRUE* und *FALS*E.
- Identifiers:
  - ein Identifier ist eine Zeichensequenz unlimitierter Länge, welche mit einem Java konformen Startzeichen starten muss, gefolgt von Java konformen Zeichenketten:
    beim Methodenaufruf *Character.isJavaIdentifierStart* muss true zurück gegeben werden. Dies umfasst '_' und '$'.
    Ein Zeichen der Zeichenkette muss Java konform sein:
    beim Methodenaufruf *Character.isJavaIdentifierPart* muss true zurück gegeben werden.
  - Identifiers dürfen nicht *NUL*L, *TRU*E,oder *FALS*E sein.
  - Identifiers dürfen nicht *NOT, AND, OR, BETWEEN, LIKE, IN,* und *IS* sein.
  - Identifiers sind entweder Header Feld Referenzen oder Property Referenzen.
  - Identifiers sind *case sensitiv*.
  - Message Header Feld Referenzen sind beschränkt auf *JMSDeliveryMod*e, *JMSPriority*, *JMSMessageID, JMSTimestamp, JMSCorrelationID* und *JMSType*.

    JMSMessageID, JMSCorrelationID und *JMSType* Werte können *null* sein und werden, falls sie das sind, als NULL Werte behandelt.
  - jeder Name, der mit 'JMSX' beginnt, ist ein JMS definierter Property Name.
  - jeder Name, der mit 'JMS_' beginnt, ist ein Provider-spezifischer Property Name.

[1] Siehe X/Open CAE Specification Data Management: Structured Query Language (SQL), Version 2, ISBN: 1-85912-151-9 March 1996.

  - jeder Name, der nicht mit 'JMS' beginnt, ist ein applikation-spezifischer Property Name.
- Whitespaces sind die selben wie in Java:
  Leerzeichen, horizontale Tabulatoren, Zeilenvorschub und Zeilenende.
- Ausdrücke / Expressions:
  - ein Selektor ist ein konditionaler Ausdruck;
    ein Selektor "*matched*", falls er true ist;
    ein Selektor *matched nicht*, falls er false oder unbekannt ist.
  - arithmetische Ausdrücke bestehen aus
    arithmetischen Ausdrücken selbst,
    Operationen,
    Identifiers mit numerischen Werten und numerischen Literalen.
  - konditionale Ausdrücke bestehen aus
    konditionalen Ausdrücken selbst,
    Vergleichsoperationen,
    logischen Operationen,
    Identifiers mit Boolean Werten und Boolean Literalen.
- Standard Klammern () zum Ordnen von Ausdrucksevaluationen wird unterstützt.
- logische Operatoren: NOT, AND, OR wobei die Präzedenz genau in dieser Reihenfolge ist
- Vergleichsoperatoren: =, >, >=, <, <=, <> (nicht gleich)
  - es können nur *vergleichbare* Werte verglichen werden. Eine Ausnahme ist der Vergleich von exakten numerischen Werten mit approximativen numersichen Werten (wie in Java). Falls versucht wird, inkompatible Datentypen zu vergleichen, wird der Selektor automatisch false.
  - *String* und *Boolean* Vergleiche sind beschränkt auf = und <>. Zwei Strings sind gleich genau dann, wenn sie die selbe Sequenz von Zeichen enthalten.
- Arithmetische Operatoren in Präzedenz Reihenfolge:
  - +, - unär (Vorzeichen)
  - *, / Multiplikation und Division
  - +, - Addition und Subtraktion
  - arithmetische Operationen müssen den Java Regeln folgen.
- *arithmetischer Ausdruck1* [NOT] BETWEEN *arithmetischer Ausdruck* and *arithmetischer Ausdruck*
  Vergleichsoperator
  - age BETWEEN 15 and 19 ist äquivalent zu age >= 15 AND age <= 19
  - age NOT BETWEEN 15 and 19 ist äquivalent zu age < 15 OR age > 19
- *identifier* [NOT] IN *(string-literal*1, *string-literal*2,...)
  Vergleichsoperator
  wobei *identifier* einen *String* oder NULL Wert enthält.
  - Country IN (' UK', 'US', 'France') ist true für 'UK' und false für 'Peru'
    es ist äquivalent zum Ausdruck
    (Country = ' UK') OR (Country = ' US') OR (Country = ' France')
  - Country NOT IN (' UK', 'US', 'France') ist false für 'UK' und true für 'Peru'
    es ist äquivalent zum Ausdruck
    NOT ((Country = ' UK') OR (Country = ' US') OR (Country = ' France'))
  - falls der *identifier* einer IN oder NOT IN Operation NULL ist, ist der Wert der Operation unbekannt.
- *identifier* [NOT] LIKE *pattern-value* [ESCAPE *escape-character*] Vergleichsoperator
  wobei *identifier* einen *String* Wert hat; *pattern-value* ist ein String Literal wobei '_' für

irgend ein Zeichen steht; '%' steht für eine Sequenz von Zeichen (inklusive der leeren Sequenz); und alle andern Zeichen werde als dieses Zeichen interpretiert. Der optionale *escape-character* ist ein ein Zeichen langes String Literal.
  - *phone LIKE '12%3'* ist true für '123' '12993' und false für '1234'
  - *word LIKE 'l_se'* ist true für 'lose' und false für 'loose'
  - *underscored LIKE '_%' ESCAPE '\'* ist true für '_foo' und false für 'bar'
  - *phone NOT LIKE '12%3'* ist false für '123' und '12993' und true für '1234'
  - falls *identifier* einer LIKE oder NOT LIKE Operation NULL ist, ist der Wert der Operation unbekannt.
- *identifier* IS NULL Vergleichsoperator testet, ob ein Header Feld null ist, oder ein Propertywert fehlt.
  - *prop_name IS NULL*
- *identifier* IS NOT NULL Vergleichsoperator testet ob ein Header Feld oder Property Wert null ist.
  - *prop_name IS NOT NULL*

JMS Providers müssen die syntaktische Korrektheit eines Selektors einer Nachricht überprüfen, falls einSelektor präsent ist. Falls der Selektor syntaktisch inkorrekt ist, muss eine JMS *InvalidSelectorException* geworfen werden.

Der folgende Massage Selektor selektiert Nachrichten, mit einem Messagetyp *car* und Farbe color = *blue* und Gewicht weight grösser als 2500 Pfund:

```
"JMSType = 'car' AND color = 'blue' AND weight > 2500"
```

1.3.8.1.2. Null Werte

Wie oben bereits erwähnt, können Header Felder und Property Werte NULL sein. Die Evaluation von Selektor Ausdrücken, welche NULL Werte enthalten, ist in der SQL 92 NULL Semantik beschrieben.

SQL betrachtet einen NULL Wert als unbekannt. Vergleiche oder Arithmetik mit einem unbekannten Wert liefert immer einen unbekannten Wert. Der IS NULL und IS NOT NULL Operator konvertiert einen unbekannten Header oder Property Wert in TRUE und FALSE Werte. Die Boolean Operatoren benutzen dreiwertige Logik, wie in folgender Tabelle definiert:

Tabelle 3 Definition des AND Operators

AND	**T**	**F**	**U**
T	T	F	U
F	F	F	F
U	U	F	U

Tabelle 4 Definition des OR Operators

OR	**T**	**F**	**U**
T	T	T	T
F	T	F	U
U	T	U	U

Tabelle 5 Definition des NOT Operators

NOT	
T	F
F	T
U	U

1.3.8.1.3. Spezielle Bemerkungen

In einem Message Selektor kann *JMSDeliveryMode* die Werte 'PERSISTENT' und 'NON_PERSISTENT'annehmen.

Date und Time Werte müssen gemäss der Java Syntax definiert werden. Als Selektorwert wird Zeit in Millis (Integer) angegeben, wie in *java.util.Calenda*r definiert.

1.3.9. Zugriff auf gesendete Messages

Eine Nachricht kann, nachdem sie vesandt wurde, beliebig verändert werden, ohne irgend einen Einfluss auf die bereits versandte Nachricht. Das selbe Message Objekt kann auch mehrfach versandt werden. Während eine Nachricht versendet wird, darf sie nicht verändert werden. Dies würde zu einem undefinierten Ergebnis führen.

1.3.10. Werte einer empfangenen Message ändern

Nach dem Empfang einer Nachricht können deren Header Felder verändert werden. Properties und Message Body sind allerdings Read Only.

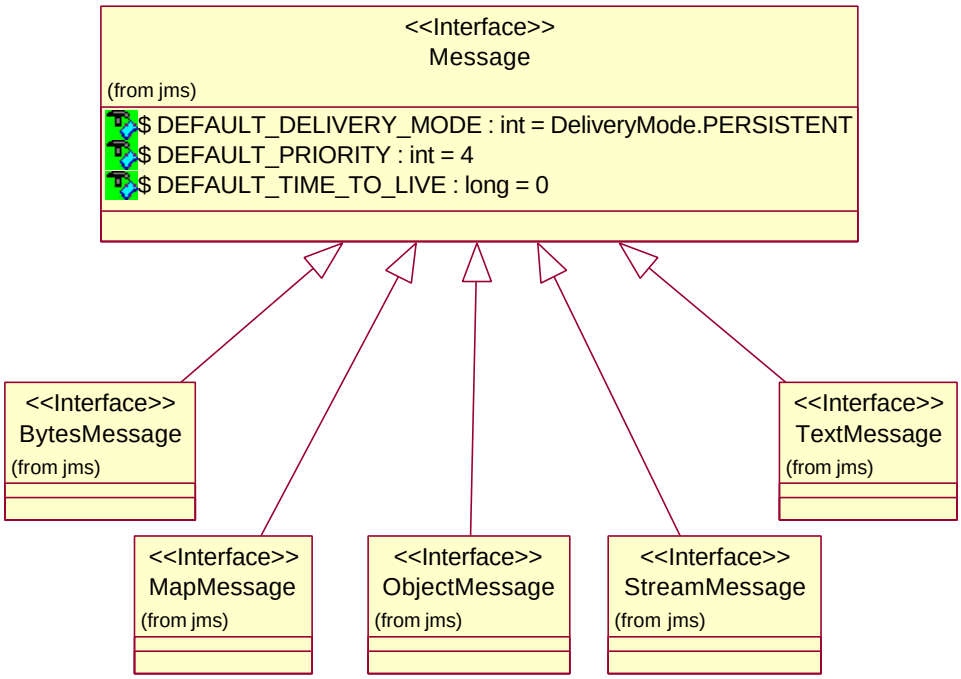
Der Grund für Read Only liegt darin, dass damit das Management der empfangenen Nachrichten vereinfacht werden kann.

1.3.11. JMS Message Body

JMS stellt fünf Formen des Message Body zur Verfügung. Jede der Formen wird mit Hilfe eines Message Interfaces beschrieben:

- **StreamMessage** - eine Message, deren Rumpf einen Stream von Java Basistype Werten umfasst. Der Rumpf wird sequentiell gelesen.
- **MapMessage** - eine Message, deren Rumpf <Name-Wert> Paare enthält, wobei die Namen Zeichenketten *Strings* sind und die Werte Java Basistypen. Auf die Einträge kann man entweder sequentiell durch Aufzählung oder zufällig mit dem Namen als Index zugreifen kann. Die Reihenfolge der Einträge ist nicht definiert.
- **TextMessage** - eine Nachricht, deren Rumpf eine Java Zeichenkette enthält: *java.lang.String*. Der Grund, warum dieser Messagetyp aufgenommen wurde ist, dass man davon ausgeht, dass *String* Messages extensiv benutzt werden.
  Einer der Gründe dafür ist, dass XML populär werden dürfte und sich XML basierte Nachrichten am einfachsten mit TextMessages realisieren lassen.
- **ObjectMessage** - eine Nachricht, die ein serialisierbares Java Objekt enthält. Falls eine Collection im Java Sinne benötigt wird, kann man eine Collection Class gemäss JDK 1.2+ verwenden..
- **BytesMessage** - eine Nachricht, welche im Rumpf uninterpretierte Bytes enthält. Dieser Message Typ kann sehr universell eingesetzt werden, auch für eigene Formattierungen..
  Obschon JMS erlaubt Message Properties für Byte Messages anzugeben, werden

Properties für Byte Messages in der Regel nicht eingesetzt, da sie das Message Format verändern oder beeinflussen könnten.



1.3.11.1. Löschen eines Message Body

Die *clearBody* Methode einer *Message* setzt den Wert des Nachrichtenrumpfes auf ‘empty’ zurück, wie die create Methode der Session Klasse. Das Löschen eines Nachrichtenrumpfes löscht jedoch die Property Entries nicht.

1.3.11.2. Read-Only Message Body

Beim Empfangen einer Nachricht ist deren Rumpf auf 'read only' gesetzt. Falls versucht wird, den Rumpf zu verändern, wird eine *MessageNotWriteableException* geworfen. Falls anschliessend der Rumpf der Nachricht gelöscht wird, ist der Rumpf im selben Status wie der leere Rumpf einer neu kreierten Nachricht.

1.3.11.3. Konversionen von StreamMessage und MapMessage

Die beiden Nachrichtentypen StreamMessage und MapMessage unterstützen die selben Basisdatentypen.

Beide, *StreamMessage* und *MapMessage*, unterstützen die folgende Konversionstabelle. Die markierten Fälle müssen unterstützt werden, alle andern werfen eine JMS *MessageFormatException*. Die *String* zu numerisch Konversion muss eine *java.lang.NumberFormatException* werfen, falls die Methode *valueOf()* die Zeichenkette nicht akzeptiert.

Tabelle 6 ein Wert vom Zeilentyp, kann als Spaltentyp gelesen werden

	boolean	**byte**	**short**	**char**	**int**	**long**	**float**	**double**	**String**	**byte[]**
boolean	x								x	
byte		x	x		x	x			x	
short			x		x	x			x	
char				x					x	
int					x	x			x	
long						x			x	
float							x	x	x	
double								x	x	
String	x	x	x		x	x	x	x	x	
byte[]										x

1.3.11.4. Messages für Nicht-JMS Clients

Viele Enterprise Messaging Systeme unterstützen die eine oder andere Form der selbst definierten Stream und/oder Map Message Typen, obschon auch Byte Typus Nachrichten eingesetzt werden könnten. JMS stellt daür die *StreamMessage* und *MapMessage* Typen zur Verfügung, als mögliche APIs

1.3.12. Provider Implementationen der JMS Message Interfaces

JMS stellt Message Interfaces zur Verfügung, welche das JMS Message Modell implementieren. Die Implementation dieser Interfaces wird aber nicht mitgeliefert.

Jeder JMS Provider stellt eigene Implementationen der Session's Message Konstruktor Methoden zur Verfügung. Damit kann ein Provider eine Message Implementation zur Verfügung stellen, welche den Anforderungen angepasst ist.

Ein Provider muss vorbereitet sein, Messages zu empfangen, die nicht seinen implementierten Typen entsprechen.

1.4. JMSCommonFacilities

1.4.1. Übersicht

Diese Kapitel beschreibt die JMS Facilities, welche beiden Domänen, PTP und Pub/Sub gemeinsam sind.

1.4.2. Administrierte Objekte

JMS administriert Objekte sind Objekte, welche JMS Konfigurationsinformation enthalten, die von einem JMS Administrator kreiert wurden und von JMS Clients benutzt werden. Sie erleichtern die praktische Administration von JMS Applikationen in einem Unternehmen.

Obschon die Interfaces für administrierte Objekte nicht explizit von JNDI (Java Naming and Directory Interface APIs) abhängt, sorgt JMS dafür, dass JMS Clients diese Objekte in einem Namensraum mit Hilfe von JNDI findet

Ein Administrator kann ein administriertes Objekt irgendwo im Namensraum ablegen, JMS definiert dafür keine Regeln.

Diese Art der Administration besitzt verschiedene Vorteile:

- sie versteckt Provider- spezifische Konfiguration- Details vor JMS Clients.
- sie abstrahiert die administrativen Informationen von JMS in Java Objekte, welche sich leicht organisieren und administrieren lassen, mit Hilfe einer gemeinsamen Managementkonsole.
- da es JNDI Providers für alle populären Naming Services geben wird, heisst dies, dass JMS Provider eine Implementation der administrierten Objekte zur Verfügung stellen kann, welche auf allen System lauffähig sind.

Ein administriertes Objekt sollte keine remote Resourcen verwenden. Es sollten lediglich die von JNDI zur Verfügung gestellten Möglichkeiten genutzt werden. Clients sollten administrierte Objekte wie lokale Java Objekte behandeln. Das Nachschlagen der Objekte in einem Verzeichnis sollte keinerlei Nebeneffekte haben.

JMS definiert zwei administrierte Objekte, *Destination* und *ConnectionFactor*y.

Es wird erwartet, dass ein JMS Provider die grundlegenden Werkzeuge zur Verfügung stellt, die ein Administrator benötigt, um administrierte Objekte in einem JNDI Namensraum zu kreieren und zu administrieren. JMS Provider Implementationen von administrierten Objekten sollten sowohl *javax.naming.Referenceable* als auch *java.io.Serializable* implementieren, so dass sie in allen JNDI Namenskontexten verwendet werden können.

Zusätzlich wird empfohlen, dass diese Implementationen den JavaBeans Design Patterns folgen.

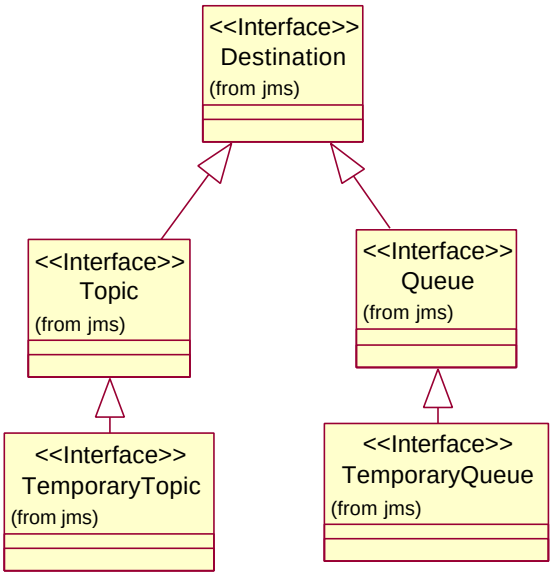
1.4.2.1. Destination

JMS definiert keine Standardsyntax für Adressen. Der Grund dafür ist, dass es zu grosse Unterschiede gibt bei den einzelnen Messaging Systemen. JMS definiert das *Destination*Objekt, welches alle Provider- spezifischen Adressen kapselt.

Da *Destination* ein administriertes Objekt ist, kann es auch Provider- spezifische Informationen, neben der Adressinformation, enthalten.

JMS unterstützt auch die Verwendung von Provider- spezifischen Adressnamen durch den Client..

Destination Objekte unterstützen den nebenläufigen Einsatz, es können also gleichzeitig mehrere Benutzer das selbe Destination- Objekt verwenden.



1.4.2.2. ConnectionFactory

Eine *ConnectionFactory* kapselt die Verbindungskonfigurationsparameter, welche vom Administrator definiert wurden. Ein Client verwendet die Factory, um eine Verbindung zu einem JMS Provider zu kreieren.

1.4.3. Connection

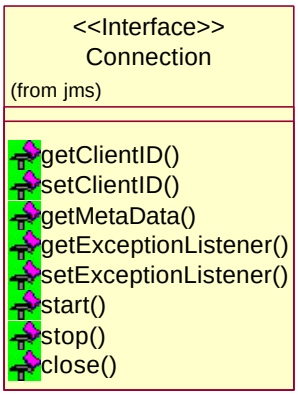
Eine JMS *Connection (Verbindung)* ist die aktive Verbindung eines Clients zu einem JMS Provider. Typischerweise werden damit Provider Resourcen ausserhalb der Java VM alloziert.

Connections unterstützt gleichzeitigen Zugriff..

Eine *Connection* dient mehreren Zwecken:

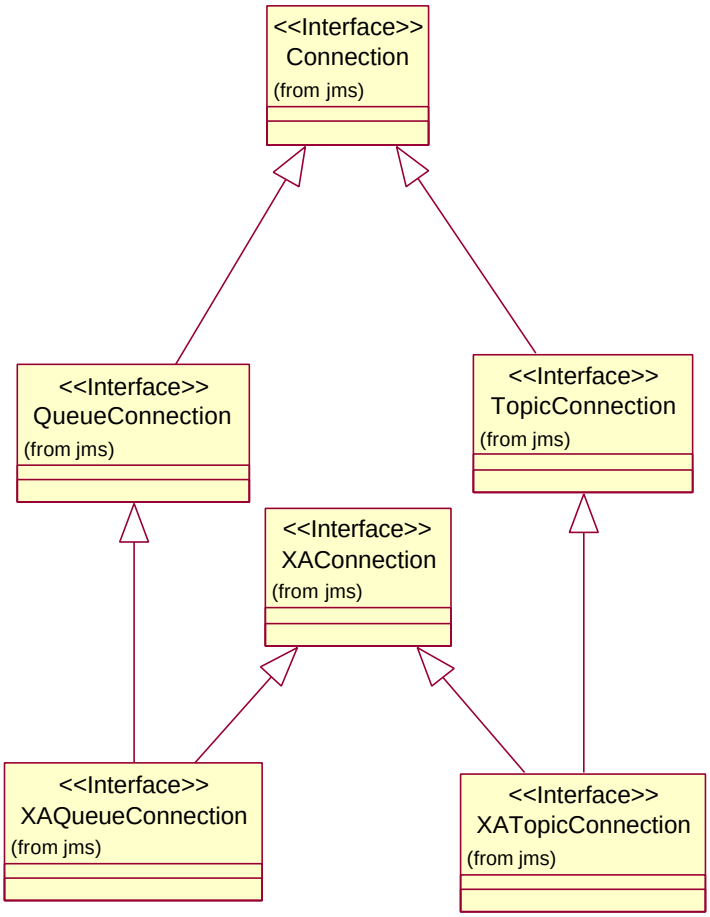
- sie kapselt eine offene Verbindung mit einem JMS Provider. Es handelt sich typischerweise um einen offenen TCP/IP Socket zwischen Client und einem Server Daemon.
- beim Kreieren wird die Authentität des Clients überprüft.
- es kann ein eindeutiger Client Identifier abgegeben werden.

- die Connection kreiert *Session*s.
- die Connection liefert *ConnectionMetaData*.
- sie unterstützt einen optionalen *ExceptionListene*r.



Da während dem Kreieren des Connection Objekts eine Authentisierung und ein Kommunikationsaufbau stattfindet, handelt es sich beim Connection Objekt um ein komplexes JMS Objekt.

Die meisten Clients werden das gesamte Messaging über eine einzige Connection abwickeln.



Komplexere Applikationen können auch mehrere Connection Objekte definieren. Aus Sicht

von JMS gibt es keinen Grund mehrere Connection Objekte zu definieren (eine mögliche Anwendung wäre eine Gateway Funktion zwischen mehreren Providern).

1.4.3.1. Authentisierung

Beim Kreieren eines Connection Objektes kann ein Client seine Authentisierung als Benutzername / Passwort angeben. Falls diese nicht angegeben werden, werden jene des aktuellen Threads verwendet, falls vorhanden.

Im JDK gibt es keine Definition für eine Thread- basierte Authentisierung. Aber vermutlich wird in einem der nächsten Releases diese Möglichkeit eingebaut.

1.4.3.2. Client Identifier

Idealerweise wird der Client Identifier einer Connection in einer Client- spezifischen *ConnectionFactory* gesetzt. Ein Client kann den Client Identifier einer Verbindung auch mit Hilfe eines Provider- spezifischen Wertes setzen. Das sollte aber die Ausnahme sein. Falls ein Wert bereits gesetzt ist und versucht wird ihn nochmals zu setzen, wird eine *IllegalStateException* Ausnahme geworfen. Diese Ausnahme wird auch geworfen, falls der Wert zu spät gesetzt wird, also nicht direkt nach dem Kreieren der Verbindung, bevor irgend einer andern Aktivität.

Der Client Identifier dient dazu, eine Verbindung herstellen zu können, zwischen der Verbindung, dem Connection Objekt und dem Zustand des Clients. Der JMS Provider muss dafür sorgen, dass dieser Client Zustand nur von genau einem Client benutzt werden kann.

1.4.3.3. Connection Setup

Ein JMS Client kreiert typischerweise :

- eine *Connectio*n;
- eine oder mehrere *Session*s;
- und eine Anzahl *MessageProducers* und *MessageConsumer*s.

Eine neu kreierte *Connection* befindet sich nach dem Kreieren im sogenannten *stopped* Modus. Das heisst, dass keine Nachrichten an dieses Objekt abgeliefert werden können. Typischerweise belässt man das Connection Objekt in diesem Zustand, bis der gesamte Setup abgeschlossen ist. Erst dann wird die *Connection. start()* Methode ausgeführt.

Eine *Connection* kann auch sofort gestartet werden. In diesem Falle muss der Client aber vorbereitet sein, asynchrone Nachrichten zu verarbeiten.

.

Ein *MessageProducer* kann Nachrichten auch senden, wenn eine *Connection* gestoppt ist .

1.4.3.4. Unterbrechen eingehender Meldungen

Das Abliefern eintreffender Nachrichten kann temporär mit Hilfe der *stop()* Methode gestoppt werden. Anschliessend kann man das Abliefern mit der *start*() Methode wieder starten.
Im gestoppten Zustand ist eine Ablieferung der Nachrichten an alle *MessageConsumers* der Connection unterbunden: Nachrichten werden nicht an die *MessageListeners* abgeliefert.

Falls ein MessageListeners zum Zeitpunkt des Aufrufes der stop() Methode aktiv ist, wird die Ausführung der stop() Methode verzögert.

1.4.3.5. Schliessen einer Connection

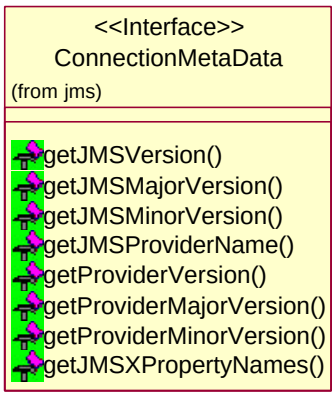
Ein Provider verwendet typischerweise Resourcen ausserhalb der JVM. Damit reicht es in der Regel nicht aus, den Garbage Collector die Arbeit machen zu lassen. Deswegen muss die *close*() Methode aufgerufen werden.

1.4.3.6. Sessions

Eine *Connection* ist eine Factory für *Sessions,* welche die Verbindung zu einem JMS Provider nutzen, um Nachrichten zu produzieren und zu konsumieren. Sessions werden weiter unten besprochen.

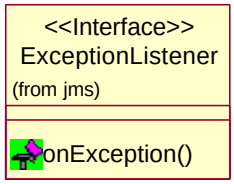
1.4.3.7. ConnectionMetaData

Eine *Connection* liefert ein *ConnectionMetaData* Objekt. Dieses Objekt enthält die Informationen über die JMS Version des Providers.



1.4.3.8. ExceptionListener

Falls ein JMS Provider ein Problem mit der Verbindung feststellt, wird er den Verbindungs-*ExceptionListener* darüber informieren. Dies geschieht mit Hilfe der Methode *onException()*

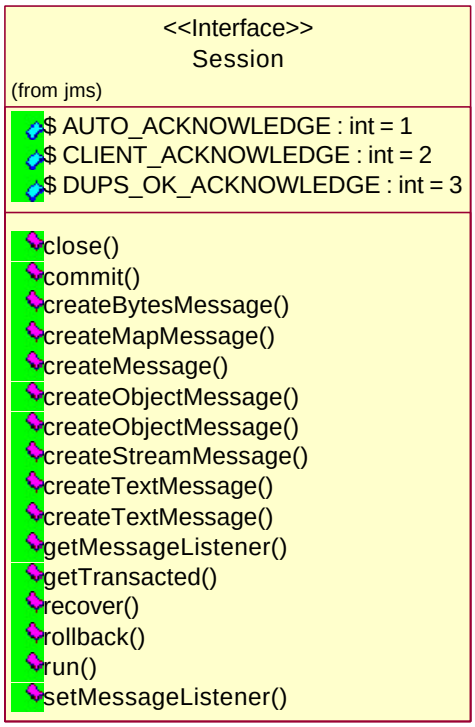


Damit kann der Client asynchron über das Problem informiert werden.

Ein JMS Provider sollte zuerst versuchen die Verbindungsprobleme selbst zu lösen, bevor er den / die Client(s) informiert.

1.4.4. Session

Eine JMS *Session* ist ein Kontext zum produzieren und konsumieren von Meldungen, ausgerichtet auf einen einzelnen [2]Thread. Obschon eine Session auch Resourcen ausserhalb der JVM allozieren kann, wird eine Session als leichtgewichtiges JMS Objekt angesehen.

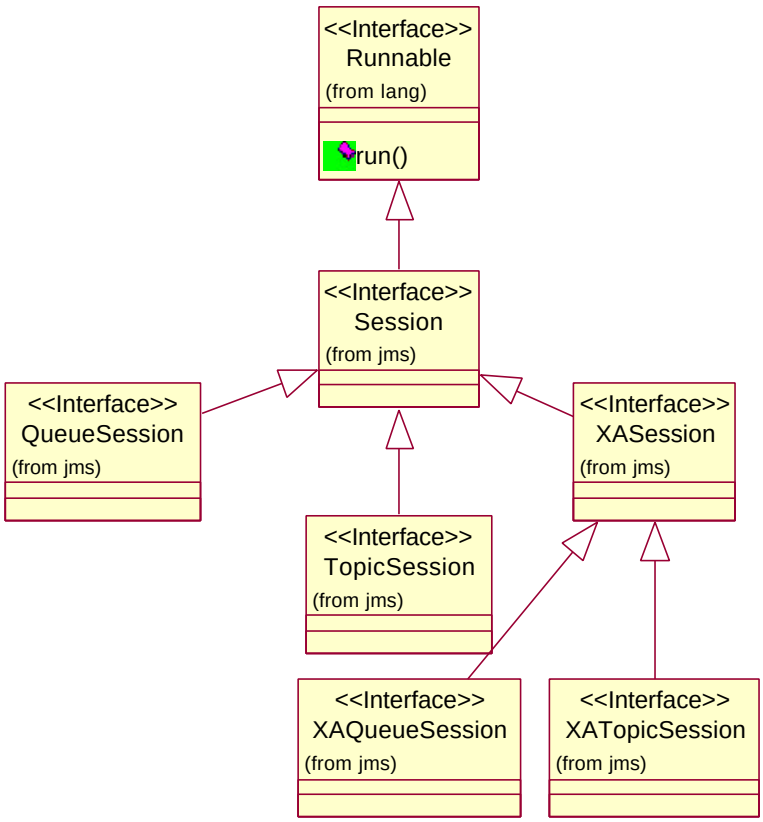


Eine *Session* dient folgenden Zwecken:

- sie liefert eine Factory für *MessageProducers* und *MessageConsumer*s.
- sie ist auch eine Factory für temporäre *Destinations*.
- sie liefert *Destination* Objekte für Clients, welche Provider- spezifische Destinationsnamen manipulieren müssen
- sie stellen Provider- optimierte Message Factories zur Verfügung.
- sie schaffen die Grundlage, Transaktionen, die sowohl Konsumenten als auch Produzenten betreffen, atomar zu kapseln.
- eine *Session* definiert eine Ordnungsrelation für die Messages, welche von ihr produziert und konsumiert werden.
- eine *Session* behält die Meldungen, die sie konsumiert, so lange, bis diese bestätigt wurden.

[2] es gibt keine Schranke betreffend der Anzahl Threads, welche ein Session Objekt benutzen können. Die Einschränkung besagt lediglich, dass nicht mehrere Threads gleichzeitig das Session Objekt benutzen. Der Benutzer ist selbst dafür verantwortlich, dass diese Einschränkung eingehalten wird. Es ist sicher das Einfachste, nur einen einzigen Thread zu verwenden.

- eine *Session* serialisiert die Ausführung der *MessageListeners.*



1.4.4.1. Schliessen einer Session

Da ein Provider Resourcen ausserhalb der JVM für eine Session allozieren kann, sollten Clients Sessions schliessen, sobald diese nicht mehr benötigt werden, unabhängig vom Garbage Collector (der braucht eventuell noch eine Weile bis er feststellt, dass die Session nicht mehr gebraucht wird).

Das gleiche gilt für die *MessageProducers* und *MessageConsumers,* die von der Session kreiert werden.

Beim Schliessen der Session werden alle laufenden bearbeitungen von Nachrichten abgeschlossen. Der Shutdown muss dem Rechnung tragen und allfällige pendente bearbeitungen abschliessen.

Sobald eine Session geschlossen ist, löst jeder Versuch die Session weiter zu benutzen, eine *IllegalStateException* aus.

1.4.4.2. Kreieren von MessageProducer und MessageConsumer

Eine Session kann mehrere *MessageProducers* und *MessageConsumer*s kreieren und bedienen.
Obschon eine Session mehrere Produzenten und Konsumenten kreieren kann, können diese lediglich seriell benutzt werden. Es gibt nur einen Kontrollthread, der sie nutzen kann. Wir kommen später darauf zurück.

1.4.4.3. Kreieren von TemporaryDestination Objekten

Keieren temporärer Destinationen ist nicht zwingend erforderlich und wird lediglich der Einfachheit halber zur Verfügung gestellt. Die Lebensdauer der Destionationen ist so lange wie die Session existiert.

1.4.4.4. Kreieren von Destinationen

Die meisten Clients werden *Destinations* verwenden, die JMS administrierte Objekte sind, und die sie mit Hilfe von JNDI oder einem analogen Mechanismus nachgeschaut haben.

Einige Clients werden eventuell *Destinations* dyamisch kreieren.

1.4.4.5. Optimierte Message Implementationen

Die Methoden zum Kreieren von Messages sind Provider- spezifisch optimiert implementiert. Damit kann der Overhead zum Behandeln von Nachrichten minimiert werden.

1.4.4.6. Konventionen zum Benutzen einer Session

Sessions sind so designed, dass sie seriell benutzt werden können. Einzige Ausnahme sind Shutdowns.

1.4.4.7. Transaktionen

Optional kann man eine Session als *transacted* spezifizieren. Jede transacted Session unterstützt die Zusammenfassung von Nachrichten, produzierten und konsumierten, die eine atomare Einheit bilden.

Beim Commiten einer atomaren Transaktion werden alle Meldungen bestätigt; beim Rollback werden alle produzierten Nachrichten zerstört und alle konsumierten Nachrichten automatisch recovered.

Für diese Operationen stehen die Methoden *commit()* oder *rollback()* zur Verfügung. Nach Abschluss einer Transaktion startet automatisch die nächste. Man hat also laufend eine aktuelle Transaktion.

JTS (Java Transaktion Services) oder irgend ein anderer Transaktions Monitor können eingesetzt werden, um die Transaktionen zusammen zu halten. Ein Aufruf der Methoden *commit* und *rollback* wirft im Falle der JTA eine JMS *TransactionInProgressExceptio*n.

1.4.4.8. Verteilte Transaktionen

JMS verlangt keine Unterstützung für verteilte Transaktionen. Falls diese aber implementiert sind, sollten diese gemäss dem JTA *XAResource* API implementiert sein
.
Ein JMS Provider kann aber auch einen eigenen verteilten Transaktionsmonitor implementieren.

1.4.4.9. Multiple Sessions

Ein Client kann mehrere Sessions kreieren. Jede Session ist ein unabhängiger Produzent und Konsument von Nachrichten.

Im Falle von Pub/Sub:

falls zwei Sessions einen Subscriber für das selbe Topic haben, werden beide die selben Meldungen erhalten.

Im Falle von PTP:
JMS spezifiziert die Sematik gleichzeitiger Empfänger nicht; JMS verbietet die Situation aber nicht.

1.4.4.10. Message Ordnung

JMS Clients müssen wissen, wann sie auf die Reihenfolge der Nachrichten bauen können und wann nicht.

1.4.4.11. Reihenfolge der Message Ankunft

Die Reihenfolge, in der Meldungen konsumiert werden, definiert eine Ordnungsrelation. Diese Ordnungsrelation ist für das Acknowledgment / Bestätigen wichtig.

1.4.4.12. Ordnung gesendeter Messages

Der Client betrachtet Nachrichten, welche seriell produziert werden, als geordnet. Dies trifft aber eigentlich nicht zu, da der Empfang der Nachricht diese Sequenz unter Umständen nicht wiedergibt:

- Nachrichten mit höherer Priorität können andere Nachrichten überholen.
- unter Umständen empfängt ein Client eine NON_PERSISTENT Message wegen einem JMS Provider Ausfall nicht.
- falls PERSISTENT und NON_PERSISTENT Nachrichten zu einer Destination gesandt werden, wird die Reihenfolge nur innerhalb des Modus garantiert.
- beim Gruppieren mehrerer Nachrichten zu einer Transaktion werden die Nachrichten als Einheit, atomar, versandt.

1.4.4.13. Message Acknowledgment

Falls eine Session mit Transaktionen arbeitet, werden Bestätigungen automatisch generiert, immer wenn die *commit* Methode ausgeführt wird; auch Rollbacks werden durch die Methode *rollback* ausgelöst.

Falls eine Session nicht mit Transaktionen arbeitet, gibt es drei Optionen:

• DUPS_OK_ACKNOWLEDGE – diese Option instruiert die Session, die Ablieferung der Meldungen "locker" zu besätigen, im Sinne, dass dublizierte Meldungen zugelassen werden (DUBS_OK...).

• AUTO_ACKNOWLEDGE – mit dieser Option bestätigt die Session automatisch den Enpfang der Nachricht durch den Client.

• CLIENT_ACKNOWLEDGE – in diesem Falle bestätigt eine Client den Empfang, indem die *acknowledge* Methode der Session aufgerufen wird.

1.4.4.14. Mehrfaches Abliefern von Messages

Ein JMS Provider darf nie eine bestätigte Nachricht ein zweitesmal abliefern.

1.4.4.15. Mehrfache Produktion von Messages

JMS Providers sollten Nachrichten nie zweimal produzieren. Ein Client kann sich dabei auf den JMS Provider verlassen, dass dieser dafür besorgt ist, dass dieser Fall nicht eintrift.

1.4.4.16. Serielle Ausführung von Client Code

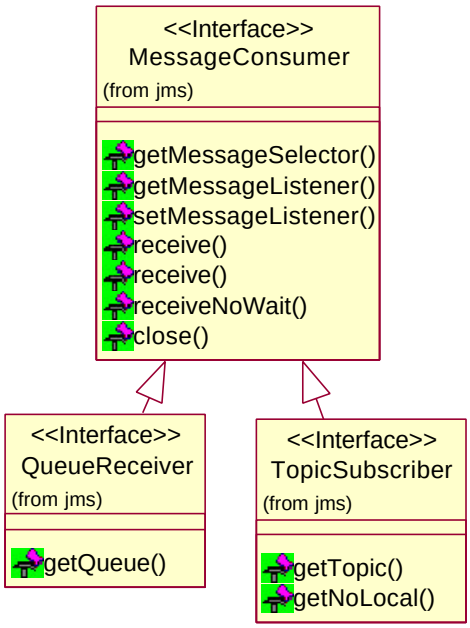
JMS verlangt nicht, dass Client Code nebenläufig / concurrent ausgeführt wird. Der Client kann dies aber verlangen und mehrere Threads gleichzeitig am Laufen haben.
Eine Session nutzt genau einen Thread, um alle MessageListeners ablaufen zu lassen. Dadurch wird eine Sequenzialisierung erzwungen, auf Stufe MessageListener.

1.4.4.17. Concurrent Message Delivery

Falls ein Client concurrent Delivery haben möchte und endsprechend implementiert ist, dann muss er mehrere Sessions benutzen. Und jede Session verfügt dann über einen SessionListener.

1.4.5. MessageConsumer

Der Client benutzt den *MessageConsumer*, um Nachrichten von einer Destination zu empfangen..



1.4.5.1. Synchrone Delivery

Ein Client kann die nächste Meldung von einem *MessageConsumer* mit Hilfe der *receive* Methode empfangen.

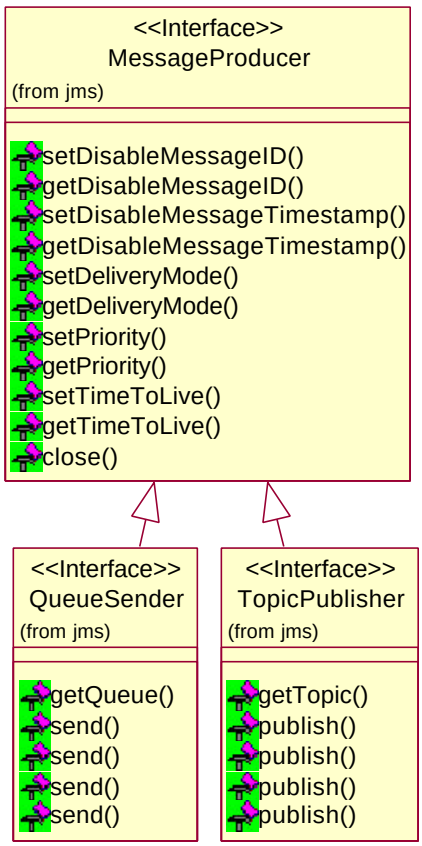
1.4.5.2. Asynchrone Delivery

Ein Client kann ein Objekt registrieren, welches das JMS *MessageListener* Interface mit einem *MessageConsumer* implementiert. Falls Meldungen für den Konsumenten eintreffen, wird der Provider sie abliefern, indem er die Listener *onMessage* Methode aufruft.

1.4.6. MessageProducer

Ein Client benutzt einen *MessageProducer*, um Meldungen an ein *Destinatio*n Objekt zu übermitteln. Dies geschieht mit Hilfe einer der folgenden Methoden:

- QueueSession.createReceiver
- TopicSession.createSubscriber
- QueueSession.createSender
- TopicSession.createPublisher



Der Client kann den Standard Liefermodus, Priorität und Time-to-Live spezifizieren, aber die selben Grössen können auch pro Nachricht spezifiziert werden.

1.4.7. Message Delivery Modus

JMS unterstützt zwei unterschiedliche Nachrichten Liefermodi:

- NON_PERSISTENT Mode :
  Vorteil dieses Modus ist der geringe Overhead. Die Nachrichten werden nicht gelogged oder gespeichert. Falls ein JMS Provider ausfällt, kann eine solche Meldung verloren gehen.
- PERSISTENT Modus instruiert den JMS Provider, dass dieser sich darum kümmern muss, dass die Nachrichten nicht verloren gehen. Der JMS Provider muss solche Nachrichten auch garantiert maximal einmal abliefern. Eine Nachricht kann also verloren gehen, darf dann aber garantiert nur einmal abgeliefert werden. Ein JMS Provider muss eine PERSISTENT Message *einmal-und-nur-einmal* abliefern. Dadurch wird die Performance verschlechtert.

PERSISTENTMessages werden trotzdem nicht garantiert an alle denkbaren Clients abgeliefert.

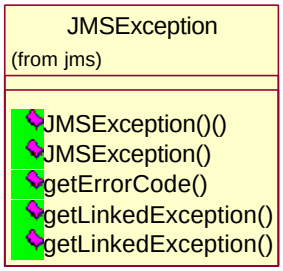
1.4.8. Message Time-To-Live

Der Client kann angeben einen Wert für die time-to-live value in Millisekunden angeben. Der Wert definiert eine Expiration Time, eine Gültigkeitsdauer.

Ein JMS Provider sollte sein bestes tun, um Nachrichten zeitgerecht in den Nachrichtenhimmel zu transferieren. JMS legt aber nicht fest, mit welcher Genauigkeit dies zu geschehen hat. Aber die TTL darf nicht einfach ignoriert werden.

1.4.9. Exceptions

JMSException ist die Basisklasse aller JMS Exceptions. Sie finden eine vollständige Liste aller JMS Exceptions in der API Beschreibung.



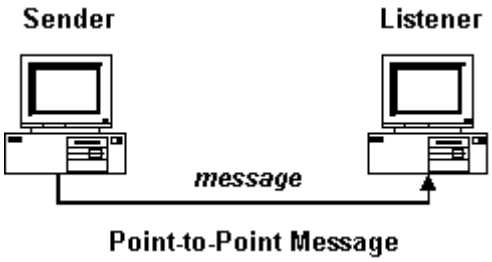
1.4.10. Reliability / Zuverlässigkeit

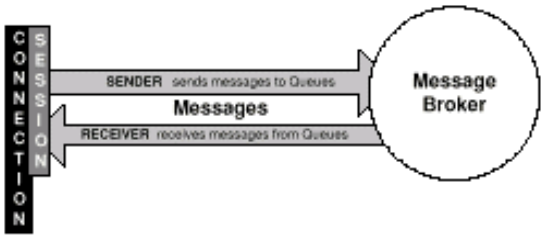
Die meisten Clients sollten Produzenten einsetzen, die PERSISTENT Messages produzieren. Dies garantiert einmal und nur einmalige Ablieferung der Nachrichten.

NON_PERSISTENT Messages auf der andern Seite, haben weniger Overhead, sind also schneller.

Allerdings muss auch im PERSISTENT Modus beachtet werden, dass es eine TTL, eine Time-to-Live gibt, also Nachrichten "verloren" gehen könnten.

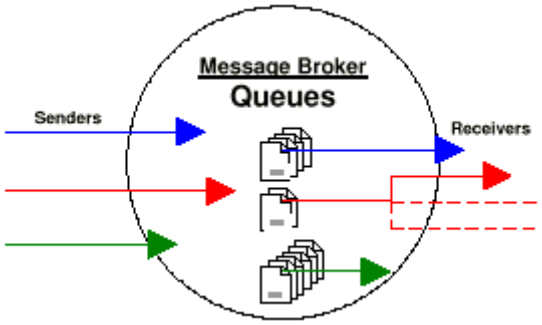
1.5. JMS Point-to-Point Modell





1.5.1. Übersicht

Point-to-Point (PTP) Systeme arbeiten mit Message Queues / Warteschlangen. Man nennt sie PTP, weil ein Client eine Nachricht an eine spezifische Warteschlange sendet. Einige PTP Systeme verwischen den Unterschied zwischen PTP und Pub/Sub (siehe weiter unten) indem sie System Clients zur Verfügung stellen, welche die Meldungen automatisch verteilen.



In der Regel liefert ein Client alle Nachrichten an eine einzige Warteschlange. Eine Warteschlange kann, wie eine generische Mailbox, jegliche Arten Nachrichten aufnehmen. Der Unterhalt der Warteschlangen ist allerdings eher aufwendig. Daher werden die meisten Warteschlangen mit Hilfe irgendwelcher administrativer Werkzeuge erstellt und als statische Grössen von ihren Clients behandelt.

Das JMS PTP Modell definiert, wie ein Client mit den Warteschlangen umzugehen hat:

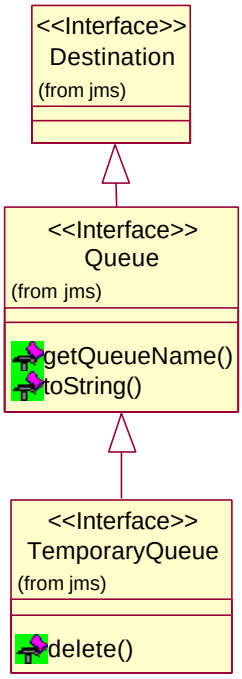
- wie findet der Client die Warteschlange
- wie kann er Nachrichten an die Warteschlange senden
- wie kann er Nachrichten aus der Warteschlange empfangen.

1.5.2. Queue Management

JMS definiert keine Facilities, mit deren Hilfe langlebige Warteschlangen kreiert, adminstriert oder gelöscht werden (für *TemporaryQueues* steht eine solche Schnittstelle zur Verfügung). Da die meisten Clients mit statisch definierten Warteschlangen arbeiten, stellt dies kein Problem dar.

1.5.3. Queue

Ein *Queue* Objekt kapselt Provider- spezifische Warteschlangennamen. Der Client spricht die Warteschlange also über dieses Objekt an. JMS spezifiziert keine Zeitdauer, während der die Nachrichten in der Warteschlange bleiben können. Auch der Overflow ist nicht geregelt.

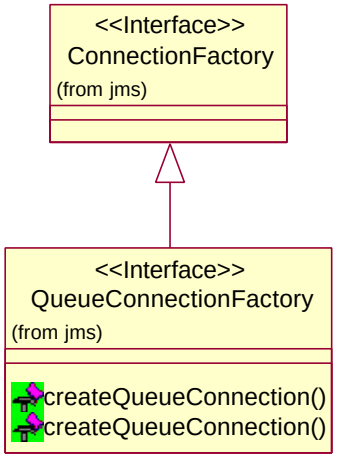


1.5.4. TemporaryQueue

Ein *TemporaryQueue* Objekt ist ein spezielles *Queue* Objekt, welches für die Dauer einer *QueueConnection* kreiert wird. Diese Warteschlange kann lediglich durch die QueueConnection genutzt werden. Sie ist systemseitig definiert.

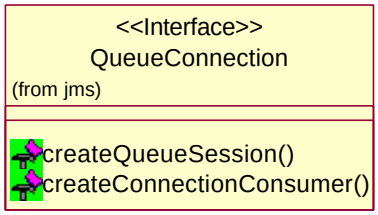
1.5.5. QueueConnectionFactory

Ein Client benutzt eine *QueueConnectionFactory*, um *QueueConnections* mit einem JMS PTP Provider zu kreieren.



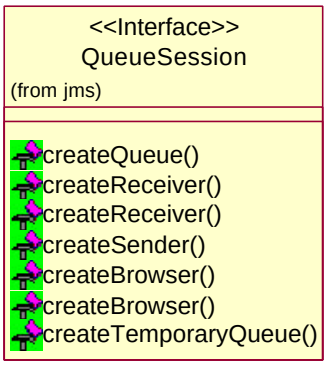
1.5.6. QueueConnection

Eine *QueueConnection* ist eine aktive Verbindung zu einem JMS PTP Provider. Ein Client benutzt eine *QueueConnection* um eine oder mehrere *QueueSessions* zu kreieren, um Nachrichten zu produzieren und zu konsumieren.



1.5.7. QueueSession

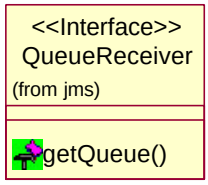
Eine *QueueSession* stellt Methoden zum Kreieren von *QueueReceiver's*, *QueueSender's*, *QueueBrowser's* und *TemporaryQueue*s zur Verfügung.



Falls Meldungen vor dem Abschluss einer *QueueSession* empfangen, aber nicht bestätigt wurden, müssen diese Meldungen erhalten bleiben und neu abgeliefert werden, sobald der Consumer das nächste Mal auf die Warteschlange zugreift.

1.5.8. QueueReceiver

DerClient benutzt ein *QueueReceiver*, um Meldungen aufzunehmen, welche an die Warteschlange gesandt wurden. Es ist denkbar, dass ein *QueueReceiver* mehrere aktive Sessions unterstützt, für ein und die selbe Warteschlange. JMS regelt nicht, wie im Einzelnen in diesem Fall vorgegangen werden soll.

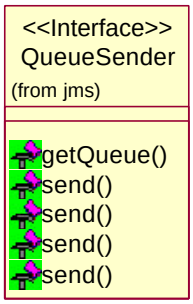


Falls ein *QueueReceiver* einen Message Selektor soezifiziert, werden Nachrichten, welche nicht selektiert werden, in der Warteschlange verbleiben.

Per Definition erlaubt ein Message Selektor einem QueueReceiver Nachrichten zu überspringen. to skip messages. Damit wird auch die Ordnungsrelation auf den Nachrichten gestört. Die Ordnungsrelation, welche vom Message Producer stammt, bleibt nur dann erhalten, falls alle Nachrichten selektiert werden.

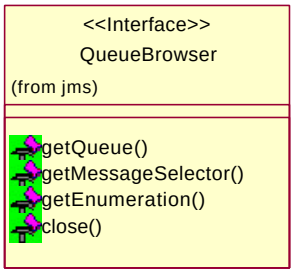
1.5.9. QueueSender

Ein Client verwendet ein *QueueSender* Objekt, um Meldungen an eine Warteschlange zu senden.



1.5.10. QueueBrowser

Der Client verwendet einen *QueueBrowser*, um Messages in einer Warteschlange zu betrachten, ohne die Meldungen zu verändern oder zu löschen. In Wirklichkeit wird eine Aufzählung *java.util.Enumeration* geliefert, mit deren Hilfe die Meldungen in der Warteschlange gescannt werden. Die Aufzählung kann entweder den Inhalt der ganzen Warteschlange umfassen, oder in Falle der Päsenz eines Selektors, des Teils der Nachrichten, welche mit dem Selektor ausgewählt werden können.

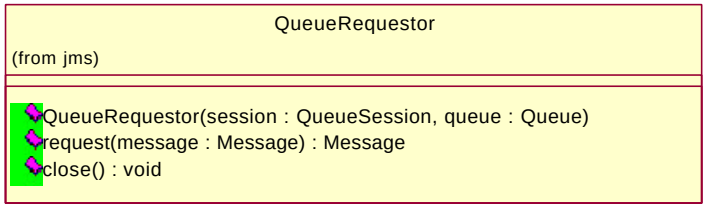


Zum Zeitpunkt des scannens der Meldungen können weitere Meldungen eintreffen, es können Meldungen auf Grund ihres Gültigkeitsstempels vernichtet werden, JMS verlangt nicht, dass der Snapshot statisch ist. Ob diese dynamischen Änderungen sichtbar werden oder nicht, hängt vom Provider ab.

1.5.11. QueueRequestor

JMS stellt eine *QueueRequestor* Helper Klasse zur Verfügung, mit der Service Request vereinfacht realisiert werden können

Dem Konstruktor *QueueRequestor* wird als Parameter eine *QueueSession* und eine Zielwarteschlange übergeben.



Es wird eine temporäre Warteschlange *TemporaryQueue* kreiert, welche die Antworten aufnehmen kann. Die *request*() Methode sendet die request- Nachricht und wartet auf eine Antwort.

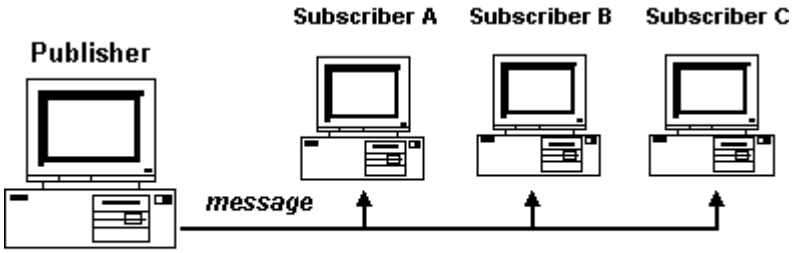
Damit wird die grundsätzliche request/reply Abstraktion definiert. JMS Providers und Clients können komplexere Mechanismen definieren.

1.5.12. Zuverlässigkeit / Reliability

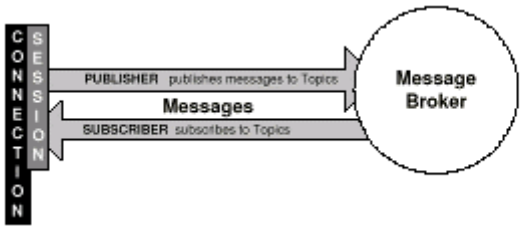
Eine Warteschlange, kann von einem Administrator kreiert werden und existiert für lange Zeit. Sie ist immer verfügbar, Nachrichten aufzunehmen, welche an sie gesandt wurden egal ob der Client aktiv ist oder auch nicht.

Der Client braucht sich also nicht darum zu kümmern, alle für ihn bestimmten Nachrichten zu erhalten; die JMS Mechanismen garantieren dies mit Hilfe der Warteschlangen.

1.6. JMS Publish/Subscribe Modell

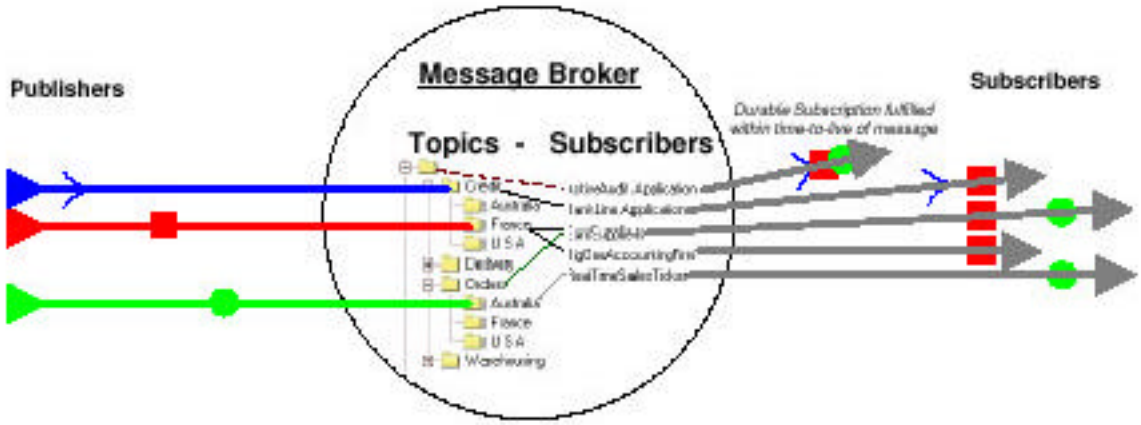


Event-Driven Publish/Subscribe Interaction



1.6.1. Übersicht

Das JMS Pub/Sub definieren, wie ein JMS Client Nachrichten publizieren und Nachrichten aus wohl definierten Knoten in einer inhaltsorientierten Hierarchie.



JMS nennt diese Knoten *topic*s.
Die Begriffe *publish* und *subscribe* sind gleichbedeutend mit den Begriffen *produce* und *consume, Produzent* und *Konsument*, die wir bisher benutzt haben.

Unter einem Topic kann man sich einen mini Message Broker vorstellen, der Nachrichten, die an ihn adressiert sind, sammelt und weiterleitet.

Die Aufteilung *Publisher + Topic* auf der einen Seite, *Topic + Subscriber* auf der andern Seite, reduziert die Komplexität des Systems: der Publisher wird entlastet, er muss sich nicht um die Abonneten kümmern.

Topic übernimmt die Aufgabe wechselnde Publisher und Subscriber im Griff zu halten.

Publishers und Subscribers sind *aktiv*, sobald die Java Objekte, welche diese repräsentieren, instanziert sind.

JMS unterstützt optional die Dauer *durability* einer Subscribtion. Diese bleiben erhalten selbst wenn der Subscriber inaktiv wird.

1.6.2. Pub/Sub Latenz

Typischerweise besteht eine gewisse Latenz in allen Pub/Sub Systemen. Es vergeht eine bestimmte Zeit bis eine Nachricht vom Produzenten zum Konsumenten gelangt.

Es kann auch passieren, dass ein neuer Abonnent / Subscriber Nachrichten empfängt, die beim Provider noch vorhanden waren, aber bereits einige Zeit alt sind.

JMS definiert nicht genau, wie diese Situationen zu handhaben sind. JMS beschreibt die Semantik erst ab dem Zeitpunkt, zu dem ein 'steady state' erreicht ist.

1.6.3. Dauerhafte Subskription

Nicht dauerhafte Subskription besteht solange wie das Subscriber Objekt. Damit kann ein Client die Nachrichten eines Topics solange empfangen, wie er lebt. Falls der Subscriber nicht mehr aktiv ist, verpasst er die Nachrichten.

Auf Kosten der Effizienz kann ein Subscriber als *durable* definiert werden. Ein *durable Subscriber* registriert eine *durable subscription* mit einer eindeutigen Identität, welche von JMS aufrechterhalten wird. Subscriber Objekte mit der selben Identität, werden damit automatisch weitere Nachrichten empfangen.

Falls es keinen durable Sbscriber gibt, bewahrt JMS die Subskriptions Nachrichten bis diese von der Subskription empfangen sind, oder bis sie ungültig werden.

Alle JMS Provider müssen in der Lage sein, dynamisch durable Subskriptionen zu kreieren und zu löschen. Die Verwaltung dieser Subskriptionen ist fakultativ.

Eine *inaktive* durable Subskription ist eine, die zwar existiert, zur Zeit aber keinen Message Consumer besitzt.

1.6.4. Topic Management

Einige Produkte verlangen, dass Topics statisch definiert werden, inklusive Authorisierungsliste, ... ; andere Produkte kennen keine Topic Administration.

JMS selbst legt keine Facilities fest, mit deren Hilfe Topics administriert werden können.

TemporaryTopic sind spezielle Topics, welche zu einer TopicConnection gehören.
Wir kommen gleich darauf zurück.

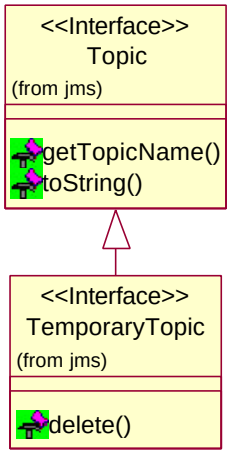
1.6.5. Topic

Ein Topic Object kapselt den Provider-spezifischen Topicnamenen. Mit diesem Namen spezifiziert der Client das Topic an die JMS Methoden.

Viele Pub/Sub Provider gruppieren Topics hierarchisch und erlauben es dem Client selektiv zu abonnieren.

JMS legt nicht fest, was ein Topic Objekt darstellt. Es kann also ein Blatt in einer Hierarchie sein, oder Teil einer grösseren Hierarchie.

Die Organisation und die Granularität der Subskription sind die wichtigsten Architekturthemen einer Pub/Sub Applikation. JMS hielft dabei aber überhaupt nicht.



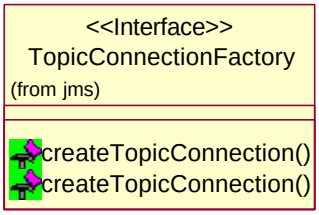
1.6.6. TemporaryTopic

Ein *TemporaryTopic* ist ein eindeutiges *Topic* Objekt, welches für die Dauer einer *TopicConnectio*n existiert. Dieses *Topic* ist systemseitig definiert und kann nur von der *TopicConnection* verwendet werden, welche es kreiert hat.

Natürlich macht es keinen Sinn eine durable Subskription für ein temporäres Topic zu kreieren. JMS zwingt aber den Provider nicht, einen solchenUnsinn zu unterbinden und eine Ausnahme zu werfen.

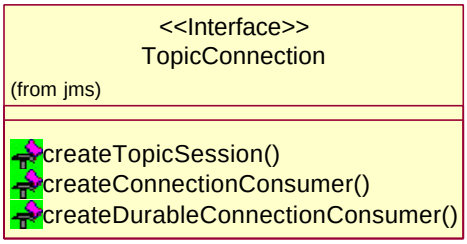
1.6.7. TopicConnectionFactory

Ein Client benutzt eine *TopicConnectionFactory* um eine *TopicConnections* mit einem JMS Pub/Sub Provider herzustellen.
.



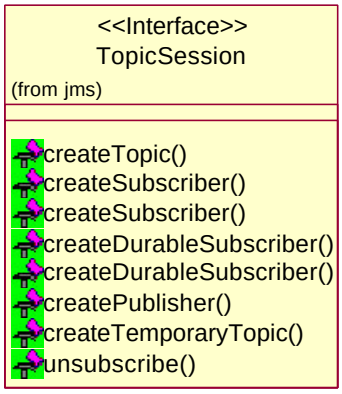
1.6.8. TopicConnection

Eine TopicConnection ist eine aktive Verbindung zu einem JMS Pub/Sub Provider. Ein Client nutzt eine TopicConnection um eine oder mehrere TopicSessions zum Produzieren und Konsumieren von Nachrichten zu kreieren.



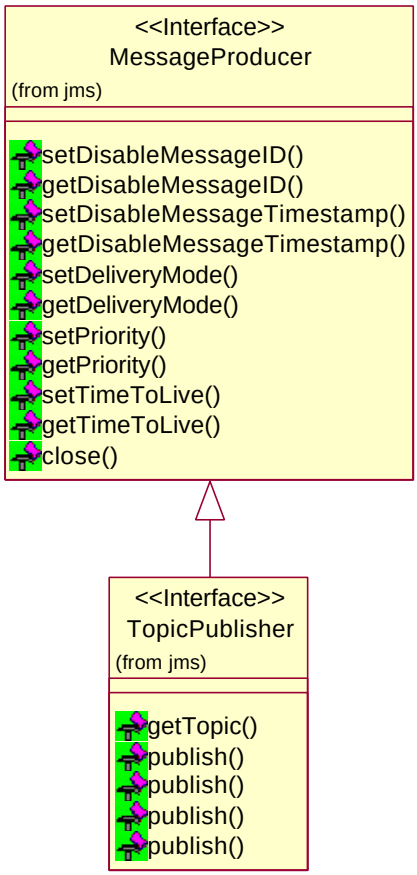
1.6.9. TopicSession

Eine TopicSession stellt Methoden zur Verfügung, um TopicPublisher, TopicSubscriber und TemporaryTopics zu kreieren. TopicSession stellt auch die Methode *unsubscribe* zur Verfügung, mit deren Hilfe durable / dauerhafte Subskriptionen gelöscht werden können.



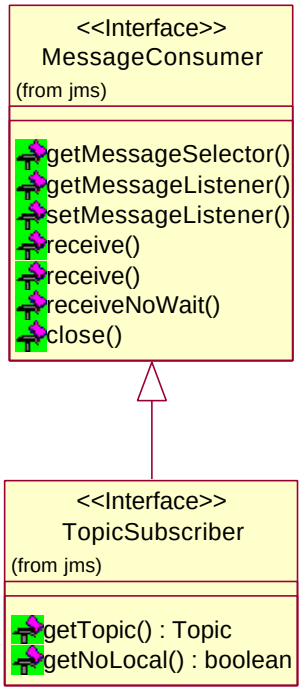
1.6.10. TopicPublisher

Ein Client verwendet einen *TopicPublisher*, um Nachrichten zu einem Topic zu publiziere. *TopicPublisher* ist die Pub/Sub Variante eines JMS *MessageProducer*.



1.6.11. TopicSubscriber

Ein Client benutzt einen *TopicSubscriber* um Nachrichten zu erhalten, welche zu einem Topic publiziert wurden. *TopicSubscriber* ist die Pub/Sub Variante vom JMS *MessageConsumer*.



Reguläre *TopicSubscriber* sind nicht durabel. Das heisst: sie empfangen Nachrichten nur solange sie aktiv sind. Nachrichten, die mit Hilfe eines Selektors aus dem Nachrichtenstrom ausgefiltert wurden, werden nie zum Subscriber geliefert. Aus Sicht des Subscribers existieren diese Nachrichten nicht.

Es gibt Fälle, in denen eine Connection sowohl publiziert als auch subskriert. In diesem Falls kann ein Attribut *NoLocal* gesetzt werden, damit die eigenen Nachrichten nicht wieder abgeliefert werden (an die Entstehungsadresse).

Eine TopicSession kann pro Destination mehrere *TopicSubscribers* definieren. Jede Nachricht wird dann an jeden *TopicSubscriber* geliefert, sofern die Selektionskriterien erfüllt werden.

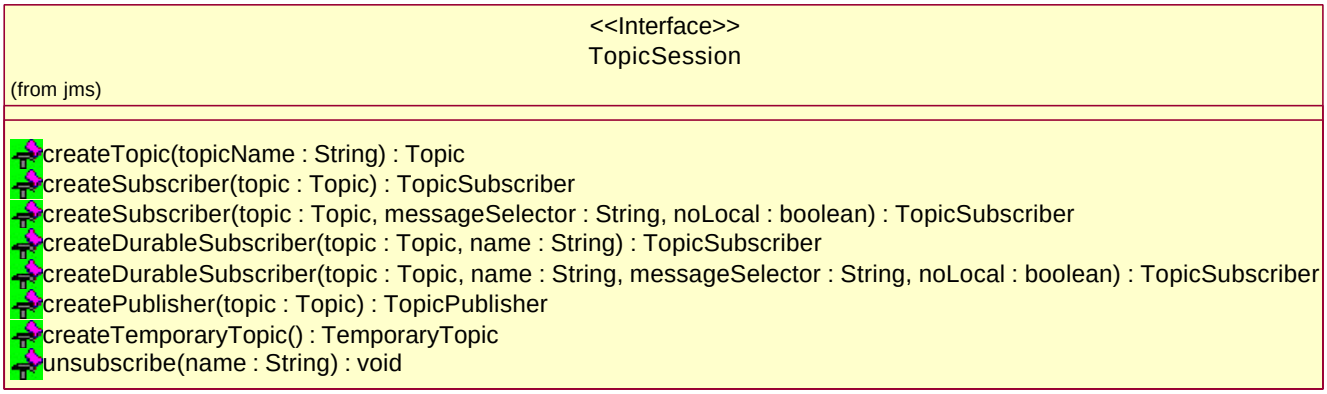
Wenn eine Nachricht an mehrere Empfänger versandt wird, wird jede Kopie als unabhängig angesehen. Änderungen an einer Nachricht an einen bestimmten Empfänger sind somit "lokale" Änderungen. Das betrifft auch die Empfangsbestätigungen, die natürlich pro versandte Kopie unabhängig sind..

1.6.11.1. Durable TopicSubscriber

Falls ein Client auch die Nachrichten empfangen möchte, die in seiner inaktiven Zeit versandt wurden, muss er einen durable *TopicSubscriber* verwenden.

JMS speichert eine Liste dieser durable Subskriptionen und garantiert, dass alle Meldungen der Publisher zu diesem Topic aufbewahrt werden, bis sie entweder vom durable Subscriber bestätigt werden, oder bis ihre Gültigkeit abgelaufen ist.

Der durable Subscriber wird in der TopicSession Klasse kreiert:



Sessions mit durablen Subscribern müssen immer den selben Client Identifier liefern (damit wird die Eindeutigkeit des Clients festgelegt und die Verfolgbarkeit des Zustandes der Nachrichten [geliefert / noch nicht geliefert/ ..] festgehalten).

Ein Client kann eine durable Subscription modifizieren: der Name muss gleich bleiben, aber das Topic und der Selektor kann ändern. In Wirklichkeit wird die Subskription gelöscht und neu angelegt.

Die Methode *unsubscribe* der TopicSessions dient zum Löschen einer dauerhaften Subskription eines Klienten. Damit wird auch der oben erwähnte Zustand gelöscht. Falls die unsubscribe Methode aufgerufen wird, während eine Nachricht übertragen wird, wird eine Fehlermeldung produziert, eine Ausnahme geworfen.

1.6.12. Recovery und Redelivery

Da nicht-durable Subskriptionen zeitlich befrstet sind, kann es passieren, dass beim Versuch eine nicht abgelieferte Nachricht wieder herzustellen, ein Fehler auftritt: die Nachricht ist eventuell schon nicht mehr vorhanden.

Nur durable Subskriptionen garantieren die Zuverlässige Ablieferung der Nachrichten, bis diese bestätigt sind.

1.6.13. Administration von Subskriptionen

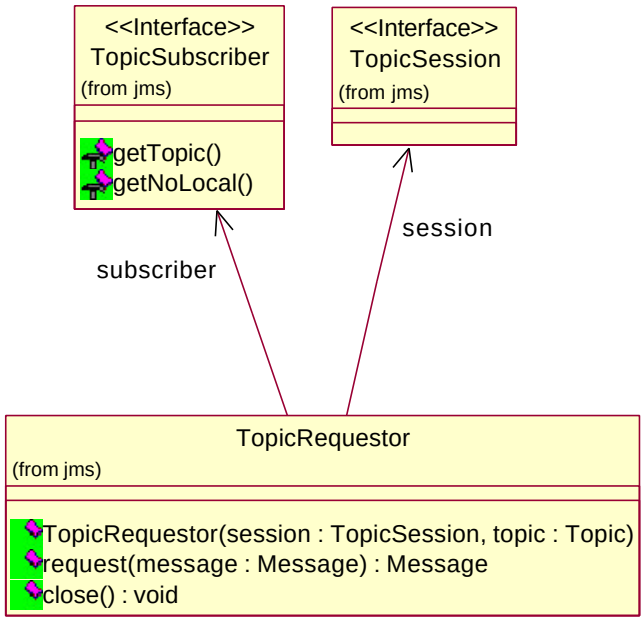
Idealerweise werden Publisher und Subscriber dynamisch registriert. Aus Sicht des Clients ist dies evident. Aus Sicht des Administrators kann man sich bestimmte Verwaltungsfunktionen vorstellen.

JMS legt keinerlei administrative Grundfunktionen für die Verwaltung der Subskriptionen fest.

1.6.14. TopicRequestor

JMS stellt eine *TopicRequestor* Helper Klasse zur Verfügung, mit deren Hilfe Service Requests vereinfacht werden können.

Der *TopicRequestor* Konstruktor hat als Parameter eine *TopicSession* und ein Topic. Der Konstruktor kreiert ein *TemporaryTopic* für die Antworten, für den Empfang von Nachrichten, und stellt eine *request()* Methode zur Verfügung, welche Request Meldungen sendet und auf die Anwort wartet.



Dies ist wieder der grundlegende request/reply Mechanismus, in seiner abstrakten Form.

Für die meisten Anwendungen reicht diese Funktionalität aber aus. JMS Provider und Clients können weitere ausgefeiltere Versionen implementieren.

1.6.15. Reliability / Zuverlässigkeit

Falls alle Nachrichten zu einem Topic empfangen werden sollten, muss ein durabler Subscriber verwendet werden. JMS garantiert in diesem Falle, dass alle Meldungen abgeliefert werden, selbst wenn der Subscriber während einer bestimmten Zeit inaktiv, nicht erreichbar, ... war.

Nicht-durable Subscriber sollten nur dann eingesetzt werden, wenn der Verlust einzelner Nachrichten tolerabel ist, wie zum Beispiel bei Video-, Audio- ... Übertragungen.

1.7. API Übersicht

Package Hierarchien:
javax.jms

1.7.1. Klassen Hierarchie

- class java.lang.Object
  - class javax.jms.**QueueRequestor**
    - class java.lang.Throwable (implements java.io.Serializable)
      - class java.lang.Exception
        - class javax.jms.**JMSException**
          - class javax.jms.**IllegalStateException**
          - class javax.jms.**InvalidClientIDException**
          - class javax.jms.**InvalidDestinationException**
          - class javax.jms.**InvalidSelectorException**
          - class javax.jms.**JMSSecurityException**
          - class javax.jms.**MessageEOFException**
          - class javax.jms.**MessageFormatException**
          - class javax.jms.**MessageNotReadableException**
          - class javax.jms.**MessageNotWriteableException**
          - class javax.jms.**ResourceAllocationException**
          - class javax.jms.**TransactionInProgressException**
          - class javax.jms.**TransactionRolledBackException**
    - class javax.jms.**TopicRequestor**

1.7.2. Interface Hierarchie

- interface javax.jms.**Connection**
  - interface javax.jms.**QueueConnection**
    - interface javax.jms.**XAQueueConnection**(erweitert auch javax.jms.XAConnection)
  - interface javax.jms.**TopicConnection**
- interface javax.jms.**XATopicConnection**(erweitert auch javax.jms.XAConnection)
- interface javax.jms.**ConnectionConsumer**
- interface javax.jms.**ConnectionFactory**
  - interface javax.jms.**QueueConnectionFactory**
    - interface javax.jms.**XAQueueConnectionFactory**(erweitert auch javax.jms.XAConnectionFactory)
  - interface javax.jms.**TopicConnectionFactory**
    - interface javax.jms.**XATopicConnectionFactory**(erweitert auch javax.jms.XAConnectionFactory)
- interface javax.jms.**ConnectionMetaData**
- interface javax.jms.**DeliveryMode**
- interface javax.jms.**Destination**
  - interface javax.jms.**Queue**
    - interface javax.jms.**TemporaryQueue**

    - interface javax.jms.**Topic**
        - interface javax.jms.**TemporaryTopic**
- interface javax.jms.**ExceptionListener**
- interface javax.jms.**Message**
    - interface javax.jms.**BytesMessage**
    - interface javax.jms.**MapMessage**
    - interface javax.jms.**ObjectMessage**
    - interface javax.jms.**StreamMessage**
    - interface javax.jms.**TextMessage**
- interface javax.jms.**MessageConsumer**
    - interface javax.jms.**QueueReceiver**
    - interface javax.jms.**TopicSubscriber**
- interface javax.jms.**MessageListener**
- interface javax.jms.**MessageProducer**
    - interface javax.jms.**QueueSender**
    - interface javax.jms.**TopicPublisher**
- interface javax.jms.**QueueBrowser**
- interface java.lang.Runnable
    - interface javax.jms.**Session**
        - interface javax.jms.**QueueSession**
        - interface javax.jms.**TopicSession**
        - interface javax.jms.**XASession**
            - interface javax.jms.**XAQueueSession**
            - interface javax.jms.**XATopicSession**
- interface javax.jms.**ServerSession**
- interface javax.jms.**ServerSessionPool**
- interface javax.jms.**XAConnection**
    - interface javax.jms.**XAQueueConnection**(erweitert auch javax.jms.QueueConnection)
    - interface javax.jms.**XATopicConnection**(erweitert auch javax.jms.TopicConnection)
- interface javax.jms.**XAConnectionFactory**
    - interface javax.jms.**XAQueueConnectionFactory**(erweitert auch javax.jms.QueueConnectionFactory)
    - interface javax.jms.**XATopicConnectionFactory**(erweitert auch javax.jms.TopicConnectionFactory)

1.8. Kommerzielle Anbieter

Sie finden die aktuelle Liste der JMS Fan Gemeinde unter der Web Adresse:

http://java.sun.com/products/jms/vendors.html

Hier eine kleine Auswahl:
JMS Implementationen sind erhältlich von:

Allaire Corporation - JRun Server
BEA Systems, Inc.,
Fiorano Software, Inc.,
GemStone
IBM
objectCube, Inc.,
Orion
Progress Software,
Push Technologies Ltd.,
Saga Software, Inc.,
Softwired, Inc.
SpiritSoft
Sun Java Message Queue Produkt (Forte)
Venue Software

Folgende Firmen haben ein Committment zum JMS API abgegeben:

Active Software Inc.,
Etsee Soft,
Glotech Solutions, Inc,
IBM MQSeries,
New Era of Networks, Inc. (NEON),
Novell, Inc.,
Oracle Corporation,
Orion,
ProSyst,
Software AG Americas,
Software Technologies Corp. (STC),
Sybase, Inc.,
Talarian Corp.,
TIBCO,
Vitria Technology, Inc.,
Xing (OpenMoM)

1.9. Entwicklung von Applikationen

Die Java Messaging Service (JMS) Spezifikation definiert zwei Messaging Modelle:

- Punkt-zu-Punkt (Warteschlange / Queue) und
- Publish- Subscribe (Topic)

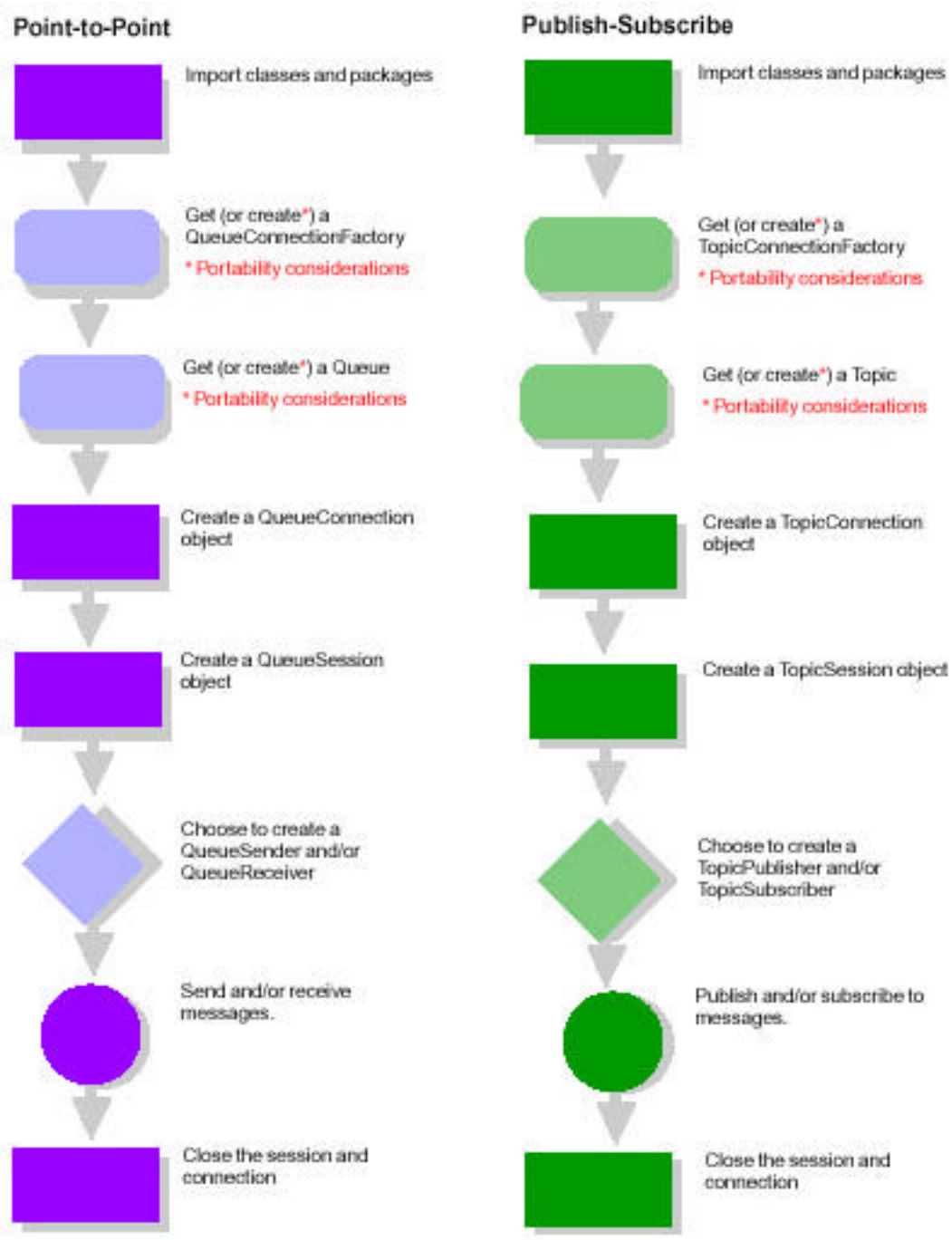
Der wesentliche Unterschied besteht darin:

- Punkt-zu-Punkt implementiert Systeme mit einem Empfänger pro Message. Jeder Sender adressiert eine Message an eine Warteschlange, welche die Nachricht für den bestimmten Empfänger bereit hält. Der Client kann sich die Nachricht bei der Warteschlange abholen.
- Publish-Subscribe implementiert Systeme mit mehreren Empfängern pro Nachricht. Jeder Publisher publiziert Nachrichten zu einem Topic in einer Content Hierarchie. Einer oder mehrere Subscriber registrieren ein Interesse an einem Topic und holen sich Kopien der Nachrichten ab.

Unabhängig vom Modell definiert JMS das Interface über das der Client sich an einem Message System anhängen kann.

I. Programmcode erstellen

Die JMS Produkte implementieren in der Regel beide Messagingmodelle, Point-to-Point und Publish-Subscribe. Als Ergebnis können wir also beide Modelle einsetzen.



II. Publish-Subscribe Client

- Importieren der benötigten Klassen oder Packages.
  für einen Java Message Queue Client:

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

  für die Verbindung zum Java Message Queue Messaging System (entsprechend anders sieht dieser Teil aus, falls ein anderes Produkt eingesetzt wird):

  ```
  import com.sun.messaging.TopicConnectionFactory;
  ```

  Typischerweise wird der Administrator diese benutzen, um administrierte Objekte zu instanzieren und administrierte Objekte zu speichern.

- TopicConnectionFactory.

  Die JMS Spezifikation beschreibt ein TopicConnectionFactory Interface, um eine Verbindung zum Messaging Systems des JMS Providers herzustellen. Es handelt sich um ein Interface, damit der Provider eine passende Implementierung wählen kann..

  i. Provider-unabhängiger Approach
  der Administrator registriert ein TopicConnectionFactory Objekt in JNDI unter einem freien, aber festgelegten Namen. In unseren Beispielen nennen wir das Objekt MyConnectionFactory. Wir könnten es auch Greifensee nennen. Wichtig ist, dass der Administrator einen Namen verwendet, der vomClient eingesetzt werden kann.

  ```
  TopicConnectionFactory factory =(TopicConnectionFactory)
  context.lookup("MyTopicConnectionFactory");
  ```

  ii. Provider-abhängiger Approach
  Clients können die Schnittstellen und Klassen auch selber instanzieren. Das hätte aber den Nachteil, dass die Client Applikation nicht mehr portabel wäre.

  ```
  TopicConnectionFactory myjmqTopicConnectionFactory;
  myjmqTopicConnectionFactory = new
  com.sun.messaging.TopicConnectionFactory();
  ```

- Get, oder create ein Message Topic.

  i. Provider-unabhängiger Approach
  Wie im obigen Abschnitt bei der TopicConnectionFactory ist Topic als Interface definiert und muss vom JMS imlementiert werden. Der Messaging System Administrator kreiert und speichert es im JNDI Namespace unter einem bekannten Namen. Der Client kann damit das Objekt nachschlagen.

  Beispiel:

  ```
  Topic topicA;
  topicA=(Topic) context.lookup("MyTopic");
  ```

ii. Provider-abhängiger Approach
Clients können die Interfaces direkt implementieren, sofern Portabilität kein Thema ist.

```
Topic myjmqTopic;
myjmqTopic = new com.sun.messaging.Topic();
```

- mit Hilfe der `TopicConnectionFactory` wird eine `TopicConnection` kreiert.

  Zum Beispiel:

  ```
  TopicConnection connection;
  connection = factory.createTopicConnection();
  ```

- mitHilfe der `TopicConnection` wird eine `TopicSession` kreiert, unter Angabe, ob es sich um eine Transaktions-orientierte Verbindung handelt.

  Zum Beispiel:

  ```
  TopicSession session;
  session = connection.createTopicSession(false,
  Session.DUPS_OK_ACKNOWLEDGE);
  ```

- mit Hilfe der `TopicSession` wird ein `TopicPublisher`, ein `TopicSubscriber` oder beides kreiert:

  Das Topic wird bei der Subskription angegeben.

  ```
  TopicSubscriber subscriber;
  subscriber=session.createSubscriber(topicA);
  ```

  Standardmässig werden diese Nachrichten synchron abgeliefert. Falls man das nicht möchte, muss ein MessageListener für das Topic kreiert werden und für den Subsciber registriert werden. Siehe das `connections` Beispiel für eine Anwendung dieser Technik.

- Sobald eine Session abgeschlossen ist, muss sie auch geschlossen werden.

  Damit werden Systemresourcen frei. Der Programmcode sieht in etwa folgendermassen aus:

  ```
  try {
     session.close();
     connection.close(); }
     catch (JMSException jmse) {
          jmse.printStackTrace();  }
  ```

3. Übersetzen des Programmcodes
   Je nach Produkt (JMS Implementation) müssen bestimmte Umgebungsvariablen gesetzt werden.

   Eine der Variablen ist in der Regel JAVA_HOME, also zum Beispiel C:\JDK1.2.2 . JAVA_HOME wird in der Regel ergänzt, wie zum Beispiel in %JAVA_HOME%\bin .

1.10. Einführende Beispiele

Als erstes wollen wir möglichst einfache, aber transparente Beispiele kennen lernen. Weiter hinten finden Sie Beispiele, welche mit der alten Referenzimplementation von Sun Forte mitgeliefert wurden.

In unseren Beispielen benutzen wir die J2EE, die Enterprise Edition von Java. Die Applikationen (Sender und Empfänger) sind einfache Applikationen, welche grundsätzliches illustrieren sollen.

Die zwei grundlegenden Beispiele, die wir besprechen, sind:
1) ein Point to Point (PTP, P2P) Beispiel
2) ein Publish / Subscribe Beispiel mit einem Message Listener

Bevor die Beispiele getestet werden können, müssen Sie die Umgebung passend setzen. Dazu müssen mehrere Umgebungsvariablen definiert werden:
1) %JAVA_HOME% : zeigt auf das Java 2 SDK,
   Beispiel: set JAVA_HOME = C:\JBuilder4\jdk1.3
2) %J2EE_HOME% : zeigt auf das Java 2 Enterprise Edition SDK
   Beispiel: set J2EE_HOME = c:\j2sdkee1.3
3) %CLASSPATH% : dieser muss
   a) das Archiv %J2EE_HOME%\lib\j2ee.jar
   b) das Verzeichnis %J2EE_HOME%\lib\locale
   umfassen
4) %PATH% : muss %J2EE_HOME%\bin enthalten

Sie können die obigen Variablen auch in den Batch Skripten setzen. Aber irgendwo müssen die Variablen definiert sein. Classpath und Pfad Angaben sind in der Regel besser bei der Ausführung anzugeben!

1.10.1. Ein einfaches Point-to-Point Beispiel

Das Beispiel zeigt, wie eine Meldung generiert und versandt (Provider) und von einem Client wieder gelesen werden kann. Dazu verwenden wir J2EE 1.3 SDK.

Dabei gehen wir schrittweise vor:
1) schreiben des Client Programms
2) übersetzen des Clients
3) starten des JMS Providers
4) kreieren des JMS administrierten Objekts (Warteschlange)
5) starten des Clients
6) löschen der Warteschlange

1.10.1.1. Schreiben des Client Programms

Der Sender, das Programm `EinfacherQueueSender`, besitzt folgende Struktur:
1. mit Hilfe des Java Naming and Directory Interface (JNDI) wird die `QueueConnectionFactory` und die Warteschlange gesucht, mit der / über die kommuniziert werden soll.
   Hinweis: die Warteschlange muss *vorgängig* mit dem J2EE Administrations Tool generiert worden sein. Diesen Schritt besprechen wir weiter unten!
2. Aufbau einer Verbindung und einer Session
3. kreieren eines `QueueSender` Objekts
4. kreieren einer `TextMessage`
5. senden von einer oder mehreren Meldungen an die Warteschlange
6. senden einer Kontrollnachricht, welche das Ende der Übertragung anzeigt
7. schliessen der Verbindung, der Session und abschliessen des `QueueSender`

Der Empfänger, das Programm `EinfacherQueueReceiver`, besitzt folgende Struktur:
1. zuerst wird auch mit JNDI eine Verbindung zur QueueConnectionFactory und der Warteschlange hergestellt, natürlich auf die selbe wie beim Sender!
2. Aufbau einer Verbindung und einer Session
3. kreieren eines `QueueReceiver` Objekts
4. starten der Verbindung - die Lieferung der Meldungen beginnt!
5. empfangen der Meldungen, welche an die Warteschlange gesandt wurden, solange, bis die Kontrollmeldung empfangen wird.
6. schliessen der Verbindung, der Session und Abschluss des `QueueReceiver`.

Hier noch einige Bemerkungen zur `Message.receive()` Methode:
Sie können bei der receive() Methode ein oder kein Argument angeben.
Falls Sie keines angeben, oder als Argument die Integer Zahl 0, dann blockiert die Methode solange, bis schliesslich eine Meldung eintrifft:

```
Message msg = queueReceiver.receive();
// oder
Message msg = queueReceiver.receive(0);
```

In unserem einfachen Beispiel spielt dies keine Rolle. Aber falls Sie JMS zusammen mit Enterprise Beans oder Servlets einsetzen, kann dieses Argument wichtig werden!

In produktiven Servern können Sie folgendermassen vorgehen:

- rufen Sie die `receive()` Methode mit einem *Timeout* Parameter (grösser als 0) auf:

  ```
  Message msg = queueReceiver.receive(2); // 2 Millisekunden Timeout
  ```

- rufen Sie die `receiveNoWait()` Methode auf. Diese sorgt dafür, dass Meldungen nur dann empfangen werden, falls bereits eine oder mehrere auf Verarbeitung warten (...`NoWait`)

  ```
  Message msg = queueReceiver.receiveNoWait();
  ```

Das Programm `EinfacherQueueReceiver` verwendet eine Endlosschleife (`while(true)` ...) um Meldungen zu empfangen und verwendet `Message m = queueReceiver.receive(1);` mit einem TimeOut Argument. Genauso könnten wir die Methode `receiveNoWait()` verwenden.

1.10.1.2. Einfaches Queue Sender Programm

```
package einfacherqueuesender;

/**
 * Title:
 * Description:
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss GmbH
 * @author J.M.Joller
 * @version 1.0
 */

/**
 * Der EinfacherQueueSender besteht aus einer main Methode
 * die Nachrichten sendet, an eine Warteschlange
 *
 * Das Gegenstück ist der Receiver EinfacherQueueReceiver
 * Parameter sind die Warteschlange, die bereits in J2EE erfasst sein
sollte
 * und die Anzahl Messages,die gesendet werden sollten
 */
import javax.jms.*;
import javax.naming.*;

public class EinfacherQueueSender {

    /**
     * Main method.
     *
     * @param args     Warteschlange
     *                 optional eine Anzahl Messages, die gesandt werden
     */
    public static void main(String[] args) {
        String                  queueName = null;
        Context                 jndiContext = null;
        QueueConnectionFactory  queueConnectionFactory = null;
        QueueConnection         queueConnection = null;
        QueueSession            queueSession = null;
        Queue                   queue = null;
        QueueSender             queueSender = null;
        TextMessage             message = null;
```

```
        final int               NUM_MSGS;
        final String    MSG_TEXT = new String("Message Nummer ");

        if ( (args.length < 1) || (args.length > 2) ) {
            System.out.println("Usage: java EinfacherQueueSender " +
                "<queue-name> [<Anzahl-messages>]");
            queueName = "meineWarteschlange";
            //System.exit(1);
        } else {
        queueName = new String(args[0]); }
        System.out.println("Namen der Warteschlange " + queueName);
        if (args.length == 2){
            NUM_MSGS = (new Integer(args[1])).intValue();
        } else {
            NUM_MSGS = 10;
        }
        System.out.println("Anzahl Messages " + NUM_MSGS);

        /*
         * kreiere ein JNDI InitialContext Objekt, falls noch keines
existiert
         */
        try {
            jndiContext = new InitialContext();
        } catch (NamingException e) {
            System.out.println("Der JNDI Context konnte nicht kreiert
werden " +
                 e.toString());
            System.exit(1);
        }

        /*
         * Look up der Connection Factory und Queue.
         * Falls eine oder beide nicht existieren, wird abgebrochen.
         */
        try {
            queueConnectionFactory = (QueueConnectionFactory)
                jndiContext.lookup("QueueConnectionFactory");
            queue = (Queue) jndiContext.lookup(queueName);
        } catch (NamingException e) {
            System.out.println("JNDI Lookup schlug fehl: " +
                e.toString());
            System.exit(1);
        }

        /*
         * Programmlogik:
         * Kreiere connection Objekt
         * Kreiere session Objekt aus dem connection Objekt;
         *      false Parameter: keine Transaktion
         * Kreiere Sender und Text Message.
         * Sende Messages
         * Sende end-of-messages Message.
         * close connection Objekt.
         */
        try {
            queueConnection =
                queueConnectionFactory.createQueueConnection();
            queueSession =
                queueConnection.createQueueSession(false,
                    Session.AUTO_ACKNOWLEDGE);
            queueSender = queueSession.createSender(queue);
```

```
            message = queueSession.createTextMessage();
            for (int i = 0; i < NUM_MSGS; i++) {
                message.setText(MSG_TEXT + " " + (i + 1));
                System.out.println("Message " + i +" : "+
                    message.getText());
                queueSender.send(message);
            }

            /*
             * Senden einer non-text control message zeigt das Ende der
             * Messages.
             */
            queueSender.send(queueSession.createMessage());
        } catch (JMSException e) {
            System.out.println("Es trat eine Exception auf: " +
                e.toString());
        } finally {
            if (queueConnection != null) {
                try {
                    queueConnection.close();
                } catch (JMSException e) {}
            }
        }
    }
}
```

1.10.1.3. Einfacher Message Empfänger

```
package einfacherqueuereceiver;

//
env.put(Context.INITIAL_CONTEXT_FACTORY,"com.sun.enterprise.naming.SerialIn
itContextFactory");
import java.util.*;
import javax.jms.*;
import javax.naming.*;
import com.sun.enterprise.naming.SerialInitContextFactory;

/**
 * Title:
 * Description:
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss GmbH
 * @author J.M.Joller
 * @version 1.0
 */

/**
 * EinfacherQueueReceiver Klasse bestehend aus main()
 * Diese liest eine oder mehrere Messages aus einer Warteschlange
 * synchron.
 * Das Partner Programm ist EinfacherQueueSender.
 * Die Warteschlange wird als Parameter übergeben
 */
public class EinfacherQueueReceiver {

    /**
     * Main Methode.
     *
     * @param args     die Warteschlange
     */
```

```
    public static void main(String[] args) {
        String                  queueName = null;
        Context                 jndiContext = null;
        QueueConnectionFactory  queueConnectionFactory = null;
        QueueConnection         queueConnection = null;
        QueueSession            queueSession = null;
        Queue                   queue = null;
        QueueReceiver           queueReceiver = null;
        TextMessage             message = null;

        /*
         * Lies den queue Namen und zeig ihn an.
         */
        if (args.length != 1) {
            System.out.println("Usage: java EinfacherQueueReceiver " +
                "<queue-name>");
            queueName = "meineWarteschlange";
            //System.exit(1);
        } else {
        queueName = new String(args[0]);}
        System.out.println("[EinfacherQueueReceiver]Der Queue Name ist " +
queueName);

        /*
         * Kreiere ein JNDI InitialContext Objekt falls noch keiner
existiert.
         */
        try {
          jndiContext = new InitialContext();
        } catch (NamingException e) {
            System.out.println("[EinfacherQueueReceiver]JNDI Context konnte
nicht generiert werden: "
                               + e.toString());
            System.exit(1);
        }

        /*
         * Look up Connection Factory Und Queue.
         * Falls eines oder beide nicht existieren wird das Programm
abgebrochen
         */
        try {

System.out.println("[EinfacherQueueReceiver]queueConnectionFactory");
            queueConnectionFactory =
(QueueConnectionFactory)jndiContext.lookup("QueueConnectionFactory");

            System.out.println("[EinfacherQueueReceiver]queue");
            queue = (Queue) jndiContext.lookup(queueName);
        } catch (NamingException e) {
            System.out.println("[EinfacherQueueReceiver]JNDI Lookup
fehlgeschlagen: "
                + e.toString());
            System.exit(1);
        }

        /*
         * Kreiere Connection.
         * Kreiere Session aus der Connection; false als Parameter
bedeutet:
         *                                                    keine
Transaktion.
```

```
         * Kreiere Receiver,
         *      dann kann die Message Lieferung starten
         * Empfange alle Text Messages aus der Queue
         *      eine nicht-text Message zeigt das Ende des Message Stromes
an
         * Close connection.
         */
        try {
            queueConnection =
                queueConnectionFactory.createQueueConnection();
            queueSession =
                queueConnection.createQueueSession(false,
                    Session.AUTO_ACKNOWLEDGE);
            queueReceiver = queueSession.createReceiver(queue);
            queueConnection.start();
            while (true) {
                Message m = queueReceiver.receive(1);
                if (m != null) {
                    if (m instanceof TextMessage) {
                        message = (TextMessage) m;
                        System.out.println("[EinfacherQueueReceiver]Lesen
der Message: " +
                            message.getText());
                    } else {
                        break;
                    }
                }
            }
        } catch (JMSException e) {
            System.out.println("[EinfacherQueueReceiver]Es trat eine
Exception auf: " +
                e.toString());
        } finally {
            if (queueConnection != null) {
                try {
                    queueConnection.close();
                } catch (JMSException e) {}
            }
        }
    }
}
```

1.10.1.4. Übersetzen der Clients

Damit wir unser PTP Beispiel starten können, müssen wir den Sender und den Empfänger (beides Clients) übersetzen. Dazu muss die entsprechende Library eingebunden sein:

In meinem Fall ist dies eine Library im JBuilder, die ich J2EE getauft habe, und die folgende Archive und Verzeichnisse umfasst:

```
%J2EE_HOME%\lib\locale
%J2EE_HOME%\lib\j2ee.jar
%J2EE_HOME%\lib\system
```

Das Übersetzen geschieht bei den Beispielen auf dem Server / der CD im JBuilder.

1.10.1.5. Starten des JMS Providers

Falls Sie J2EE 1.3 SDK einsetzen, wird Ihnen JMS im SDK mitgeliefert (siehe Bibliotheken oben). Sie können den Administrator mit dem Batch Skript starten:

```
@echo off
Rem
Rem   Starten der J2EE
Rem
set J2EE_HOME=c:\j2sdkee1.3
c:
cd %J2EE_HOME%\bin
@echo Starten des Java Enterprise Servers
@j2ee.bat -verbose
pause
D:
```

Das Starten kann eine Weile dauern, da beispielsweise ein Apache Web Server und die Cloudscape Datenbank gestartet wird!J2EE meldet sich mit einer Nachricht wie:

```
Starten des Java Enterprise Servers

J2EE server listen port: 1050
Naming service started:1050
Binding DataSource, name = jdbc/InventoryDB, url =
jdbc:cloudscape:rmi:CloudscapeDB;create=true
Binding DataSource, name = jdbc/DB2, url =
jdbc:cloudscape:rmi:CloudscapeDB;create=true
Binding DataSource, name = jdbc/EstoreDB, url =
jdbc:cloudscape:rmi:CloudscapeDB;create=true
Binding DataSource, name = jdbc/Cloudscape, url =
jdbc:cloudscape:rmi:CloudscapeDB;create=true
Binding DataSource, name = jdbc/DB1, url =
jdbc:cloudscape:rmi:CloudscapeDB;create=true
Binding DataSource, name = jdbc/XACloudscape, url = jdbc/XACloudscape__xa
Binding DataSource, name = jdbc/XACloudscape__xa, dataSource =
COM.cloudscape.core.RemoteXaDataSourc
e@505c13
Starting JMS service ... Initialization complete - waiting for client
requests
Binding : < JMS Destination : jms/Topic , javax.jms.Topic >
Binding : < JMS Destination : jms/Queue , javax.jms.Queue >
Binding : < JMS Cnx Factory : QueueConnectionFactory , Queue , No
properties >
Binding : < JMS Cnx Factory : jms/TopicConnectionFactory , Topic , No
properties >
Binding : < JMS Cnx Factory : TopicConnectionFactory , Topic , No
properties >
Binding : < JMS Cnx Factory : jms/QueueConnectionFactory , Queue , No
properties >
Starting web service at port:8000
Starting secure web service at port:7000
Apache Tomcat/4.0-b4-dev
Starting web service at port:9191
Apache Tomcat/4.0-b4-dev
J2EE server startup complete.
```

1.10.1.6. Kreieren der JMS administrierten Objekte

Die Warteschlange, über die / mit Hilfe derer wir kommunizieren wollen, bzw. der Sender und der Receiver Meldungen austauschen wollen, muss zuerst im JNDI registriert werden. Dies geschieht mit dem J2EE Administrations Tool, oder dem Batch Skript auf dem Server / der CD:

```
@echo off
Rem
Rem   Starten der J2EE
Rem
set J2EE_HOME=c:\j2sdkee1.3
c:
cd %J2EE_HOME%\bin
@echo Eintrag der Warteschlange
@j2eeadmin -addJmsDestination meineWarteschlange queue
pause
```

Zur Sicherheit überprüfen wir auch gleich, ob der Eintrag erfolgte und korrekt ist, speziell falls Sie ohne Skripts arbeiten!

```
@echo off
Rem
Rem   Starten der J2EE
Rem
set J2EE_HOME=c:\j2sdkee1.3
c:
cd %J2EE_HOME%\bin
@echo Ueberpruefen des Contexts
j2eeadmin -listJmsDestination
pause
```

mit der Ausgabe:

```
JmsDestination
--------------
< JMS Destination : meineWarteschlange , javax.jms.Queue >
< JMS Destination : jms/Topic , javax.jms.Topic >
< JMS Destination : jms/Queue , javax.jms.Queue >
```

1.10.1.7. Starten der PTP Clients

Wenn wir von Clients sprechen, denken wir an den Sender und den Receiver. Der Server ist in diesem Sinne der provider, in unserem Falle also J2EE.

```
@echo off
echo Starten des JMS Beispiels. Bitte warten ...
Rem
set J2EE_HOME=c:\j2sdkee1.3
Rem
cd EinfacherQueueSender
%JAVA_HOME%\bin\java -classpath
c:\jbuilder4\jdk1.3\lib;%J2EE_HOME%\lib\j2ee.jar;%J2EE_HOME%\lib\locale;.;.
.  -Djms.properties=c:\j2sdkee1.3\config\jms_client.properties
einfacherqueuesender.EinfacherQueueSender meineWarteschlange 30
pause
```

Die Property Dateien werden mit J2EE mitgeliefert!

```
#comments jms properties file
com.sun.jms.internal.java.naming.factory.initial=com.sun.enterprise.naming.
SerialInitContextFactory
com.sun.jms.internal.java.naming.provider.url=

com.sun.jms.client.transport_preference=IIOP

#Possible values are SEVERE,WARNING,INFO, FINE, FINER, FINEST.
com.sun.jms.default.loglevel=WARNING
```

Der Sender liefert folgende Ausgabe:

```
Starten des JMS Beispiels. Bitte warten ...
Namen der Warteschlange meineWarteschlange
Anzahl Messages 30
Java(TM) Message Service 1.0.2 Reference Implementation (build b13)
Message 0 : Message Nummer  1
Message 1 : Message Nummer  2
Message 2 : Message Nummer  3
Message 3 : Message Nummer  4
Message 4 : Message Nummer  5
Message 5 : Message Nummer  6
Message 6 : Message Nummer  7
Message 7 : Message Nummer  8
Message 8 : Message Nummer  9
Message 9 : Message Nummer  10
Message 10 : Message Nummer  11
Message 11 : Message Nummer  12
Message 12 : Message Nummer  13
Message 13 : Message Nummer  14
Message 14 : Message Nummer  15
Message 15 : Message Nummer  16
Message 16 : Message Nummer  17
Message 17 : Message Nummer  18
Message 18 : Message Nummer  19
Message 19 : Message Nummer  20
Message 20 : Message Nummer  21
Message 21 : Message Nummer  22
Message 22 : Message Nummer  23
Message 23 : Message Nummer  24
Message 24 : Message Nummer  25
Message 25 : Message Nummer  26
Message 26 : Message Nummer  27
Message 27 : Message Nummer  28
Message 28 : Message Nummer  29
Message 29 : Message Nummer  30
Taste drücken, um fortzusetzen . . .
```

Nun können wir den Empfänger Client starten:

```
@echo off
echo Starten des JMS Beispiels. Bitte warten ...
Rem
set J2EE_HOME=c:\j2sdkee1.3
Rem
cd EinfacherQueueReceiver
%JAVA_HOME%\bin\java -classpath
c:\jbuilder4\jdk1.3\lib\;%J2EE_HOME%\lib\j2ee.jar;%J2EE_HOME%\lib\locale;
-Djms.properties=%J2EE_HOME%\config\jms_client.properties
einfacherqueuereceiver.EinfacherQueueReceiver meineWarteschlange
pause
```

Der Start dieses Skripts liefert, auch noch nachdem der Sender schon lange seine Tätigkeiten abgeschlossen hat:

```
Starten des JMS Beispiels. Bitte warten ...
[EinfacherQueueReceiver]Der Queue Name ist meineWarteschlange
[EinfacherQueueReceiver]queueConnectionFactory
[EinfacherQueueReceiver]queue
Java(TM) Message Service 1.0.2 Reference Implementation (build b13)
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  1
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  2
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  3
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  4
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  5
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  6
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  7
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  8
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  9
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  10
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  11
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  12
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  13
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  14
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  15
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  16
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  17
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  18
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  19
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  20
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  21
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  22
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  23
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  24
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  25
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  26
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  27
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  28
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  29
[EinfacherQueueReceiver]Lesen der Message: Message Nummer  30
Taste drücken, um fortzusetzen . . .
```

1.10.1.8. Löschen der Warteschlange

Das administrierte Objekt 'Warteschlange' (queue) muss oder sollte nach Abschluss wieder aus dem Enterprise Server entfernt werden.

```
@echo off
Rem
Rem   Stoppen der J2EE : Queue löschen
Rem
set J2EE_HOME=c:\j2sdkee1.3
c:
cd %J2EE_HOME%\bin
@j2eeadmin -removeJmsDestination meineWarteschlange
pause
```

Damit sind wir am Ende unseres einfachsten Punkt-zu-Punkt Beispiels angelangt.

1.10.2. Ein einfaches Publish / Subscribe Beispiel

Nun wollen wir mit Topics und damit mit Publishern und Subscribern arbeiten.
Dazu entwickeln wir wieder einen einfachen 'Sender', den Publisher und einen 'Receiver', den Subscriber. Daneben brauchen wir auch in diesem Beispiel ein administriertes Objekt, das Topic und natürlich den Message Broker / Provider, in Form der J2EE.

Wir gehen wieder schrittweise vor:
1. schreiben der publish und subscribe Programme
2. übersetzen dieser Programme
3. starten des JMS Providers
4. kreieren der JMS administrierten Objekte
5. starten von Publisher und Subscriber
6. löschen des Topics und herunterfahren des Servers

1.10.2.1. Schreiben der Publisher und Subscriber Programme

Unser *Publisher* funktioniert folgendermassen:
1. mit JNDI die `TopicConnectionFactory` und das Topic bestimmen
2. eine Connection und eine Session kreieren
3. den `TopicPublisher` kreieren
4. die `TextMessage` kreieren
5. eine oder mehrere Messages zum Topic publizieren
6. die Connection schliessen, und damit automatisch auch schliessen der Session und des `TopicPublisher`.

Unser *Subscriber* funktioniert folgendermassen:
1. mit JNDI die `TopicConnectionFactory` und das Topic bestimmen
2. eine Connection und eine Session kreieren
3. den `TopicSubscriber` kreieren
4. eine Instanz der `TextListener` Klasse kreieren und beim `TopicSubscriber` registrieren
5. starten der Connection und damit lesen der Meldungen
6. falls der Benutzer ein `q` oder `Q` eingibt, wird die Client beendet.

Der `TextListener` funktioniert folgendermassen:
1. falls eine Meldung eintrifft wird die onMessage() Methode ausgeführt.
2. die `onMessage()` Methode konvertiert die eintreffende Nachricht in eine `TextMessage` und zeigt deren Inhalt an.

¨

1.10.2.1.1. Der Publisher

```
package einfachertopicpublisher;

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

/**
 * Title:
 * Description:
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss GmbH
 * @author J.M.Joller
 * @version 1.0
 */

/**
 * EInfacherTopicPublisher : besteht aus einer main Methode, die
 * Messages an ein Topic publiziert
 *
 * Partnetprogramm: EInfacherTopicSubscriber.
 *
 * Das Topic wird als Parameter übergeben. Es muss zuerst registriert
werden
 * Standardmässig sendet das Programm eine einfache Meldung
 * Sie können aber auch eine Anzahl Messages als Parameter 2 angeben
 */
public class EinfacherTopicPublisher {

    /**
     * Main Methode.
     *
     * @param args     topic
     *                 optional: Anzahl Messages
     */
    public static void main(String[] args) {
        String                  topicName = null;
        Context                 jndiContext = null;
        TopicConnectionFactory  topicConnectionFactory = null;
        TopicConnection         topicConnection = null;
        TopicSession            topicSession = null;
        Topic                   topic = null;
        TopicPublisher          topicPublisher = null;
        TextMessage             message = null;
        final int               NUM_MSGS;
        final String    MSG_TEXT = new String("Einfache Text Message");

        if ( (args.length < 1) ) {
            System.out.println("Usage: java " +
                "EinfacherTopicPublisher <topic-name> " +
                "[<anzahl-messages>]");
            topicName = "meinTopic";
            //System.exit(1);
        } else {
        topicName = new String(args[0]); }
        System.out.println("Der Name des Topics ist " + topicName);
        if (args.length == 2){
            NUM_MSGS = (new Integer(args[1])).intValue();
        } else {
            NUM_MSGS = 1;
```

```
        }
        System.out.println("Kreieren des JDNI Initial Contexts");
        /*
         * Kreieren eines JNDI InitialContext Objekt falls noch keines
existiert
         */
        try {
            jndiContext = new InitialContext();
        } catch (NamingException e) {
            System.out.println("JNDI Context konnte nicht kreiert werden: "
+
                               e.toString());
            System.exit(1);
        }

        /*
         * Look up der Connection Factory und des Topics.
         * Falls keines existiert, wird abgebrochen
         */
        System.out.println("Lookup der Connection Factory und des Topics");
        try {
            topicConnectionFactory = (TopicConnectionFactory)
                jndiContext.lookup("TopicConnectionFactory");
            topic = (Topic) jndiContext.lookup(topicName);
        } catch (NamingException e) {
            System.out.println("JNDI Lookup schlug fehl: " +
                e.toString());
            System.exit(1);
        }

        /*
         * Ablauf:
         * kreiere connection Objekt.
         * kreiere session Objekt aus der connection;
         *        'false' bedeutet: keine Transaktion
         * kreiere Publisher und Text Message.
         * sende Messages
         * schliesse connection Objekt.
         */
        try {
            System.out.println("createTopicConnection");
            topicConnection =
                topicConnectionFactory.createTopicConnection();
            System.out.println("createTopicSession");
            topicSession =
                topicConnection.createTopicSession(false,
                    Session.AUTO_ACKNOWLEDGE);
            System.out.println("createPublisher");
            topicPublisher = topicSession.createPublisher(topic);
            System.out.println("createTextMessage");
            message = topicSession.createTextMessage();
            for (int i = 0; i < NUM_MSGS; i++) {
                message.setText(MSG_TEXT + " " + (i + 1));
                System.out.println("Message wird publiziert: " +
                    message.getText());
                System.out.println("publish()");
                topicPublisher.publish(message);
            }
        } catch (JMSException e) {
            System.out.println("Es trat eine Exception auf: " +
                e.toString());
        } finally {
```

```
            if (topicConnection != null) {
                try {
                    topicConnection.close();
                } catch (JMSException e) {}
            }
        }
    }
}
```

1.10.2.1.2. Der Subscriber

```
package einfachertopicsubscriber;

import javax.jms.*;
import javax.naming.*;
import java.io.*;

/**
 * Title:
 * Description:
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss GmbH
 * @author J.M.Joller
 * @version 1.0
 */

 /**
 * EinfacherTopicSubscriber
 * empfängt eine oder mehrere Text Messages aus dem Topic
 * mittels asynchroner Kommunikation.
 * Es wird ein Listener eingesetzt: TextListener.
 * Das Programm gehört zu : EinfacherTopicPublisher
 * und der Hilfsklasse TextListener
 *
 * Parameter ist das Topic
 *
 * Abbruch / Beenden des Programms mit Q oder q
 */
public class EinfacherTopicSubscriber {

    /**
     * Main Methode.
     *
     * @param args     Topic
     */
    public static void main(String[] args) {
        String                  topicName = null;
        Context                 jndiContext = null;
        TopicConnectionFactory  topicConnectionFactory = null;
        TopicConnection         topicConnection = null;
        TopicSession            topicSession = null;
        Topic                   topic = null;
        TopicSubscriber         topicSubscriber = null;
        TextListener            topicListener = null;
        TextMessage             message = null;
        InputStreamReader       inputStreamReader = null;
        char                    answer = '\0';

        /*
         * Lies den Namen aus der Kommandozeile
         */
        if (args.length != 1) {
            System.out.println("Usage: java EinfacherTopicSubscriber "
                + "<topic-name>");
            //System.exit(1);
            topicName = "MeinTopic";
        } else {   topicName = new String(args[0]);}
        System.out.println("Name des Topics : " + topicName);
```

```
        /*
         * kreiere einen JNDI InitialContext falls noch keiner existiert
         */
        try {
            jndiContext = new InitialContext();
        } catch (NamingException e) {
            System.out.println("Der JNDI Context konnte nicht kreiert
werden: " +
                    e.toString());
            System.exit(1);
        }

        /*
         * Look Up des Connection Factory Objekts und des Topic.
         */
        try {
            topicConnectionFactory = (TopicConnectionFactory)
                jndiContext.lookup("TopicConnectionFactory");
            topic = (Topic) jndiContext.lookup(topicName);
            //NamingEnumeration ne = jndiContext.getNameInNamespace();
            System.out.println("Inhalt des
Contexts:"+jndiContext.listBindings()
        } catch (NamingException e) {
            System.out.println("JNDI Lookup schlug fehl: "
                + e.toString());
            System.exit(1);
        }

        /*
         * Ablauf:
         * kreiere connection.
         * kreiere session mit der connection;
         *    'false' bedeutet: keine Transaktion
         * kreiere subscriber.
         * registriere message listener (TextListener).
         * empfange Text Messages vom Topic.
         * Nachdem alle Meldungen emfangen wurden: Q oder quit.
         * schliesse connection.
         */
        try {
            topicConnection =
                topicConnectionFactory.createTopicConnection();
            ConnectionMetaData cmd = topicConnection.getMetaData();
            System.out.println("Comnnection Metadata : ");
            System.out.println("
JMSMajorVersion="+cmd.getJMSMajorVersion());
            System.out.println("
JMSMinorVersion="+cmd.getJMSMinorVersion());
            System.out.println("
JMSProviderName="+cmd.getJMSProviderName());
            System.out.println("
JMSVersion="+cmd.getJMSVersion());
            System.out.println("
JMSXPropertyVersion="+cmd.getJMSXPropertyNames());
            System.out.println("
JMSXProviderVersion="+cmd.getJMSXPropertyNames());

            topicSession =
                topicConnection.createTopicSession(false,
                    Session.AUTO_ACKNOWLEDGE);
            topicSubscriber = topicSession.createSubscriber(topic);
            topicListener = new TextListener();
```

```
            topicSubscriber.setMessageListener(topicListener);
            topicConnection.start();
            System.out.println("Beenden des Programms mit Q oder q, "
                + "und <return>");
            inputStreamReader = new InputStreamReader(System.in);
            while (!((answer == 'q') || (answer == 'Q'))) {
                try {
                    answer = (char) inputStreamReader.read();
                } catch (IOException e) {
                    System.out.println("I/O Exception: "
                        + e.toString());
                }
            }
        } catch (JMSException e) {
            System.out.println("Es trat eine Exception auf: "
                + e.toString());
        } finally {
            if (topicConnection != null) {
                try {
                    topicConnection.close();
                } catch (JMSException e) {}
            }
        }
    }
}
```

mit dem `TextListener`:

```
package einfachertopicsubscriber;

/**
 * Title:
 * Description:
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss GmbH
 * @author J.M.Joller
 * @version 1.0
 */

 /**
 * TextListener implementiert das MessageListener
 * Interface mittels einer onMessage Methode
 * Diese zeigt den Inhalt einer Text Meldung an
 *
 * Diese Klasse ist für die Anwendung EinfacherTopicSubscriber
 * bestimmt.
 */
import javax.jms.*;

public class TextListener implements MessageListener {

    /**
     * Casten der Message in eine Text Message und Ausgabe des Textes
     *
     * @param message     eintreffende Message
     */
    public void onMessage(Message message) {
        TextMessage msg = null;

        try {
```

```
            if (message instanceof TextMessage) {
                msg = (TextMessage) message;
                System.out.println("Message wird gelesen: " +
                    msg.getText());
            } else {
                System.out.println("Keine Text Message: "
                    + message.getClass().getName());
            }
        } catch (JMSException e) {
            System.out.println("JMSException in onMessage(): " +
                e.toString());
        } catch (Throwable te) {
            System.out.println("Exception in onMessage():"
                + te.getMessage());
        }
    }
}
```

1.10.2.2. Übersetzen der Pub/Sub Clients

Auch hier ist es wichtig, dass Sie die im vorigen Beispiel erwähnten Archive und Bibliotheken eingebunden haben!

Das Übersetzen geschieht wieder im JBuilder!

1.10.2.3. Starten des Providers - J2EE

Dieser Schritt geschieht genau gleich wie im vorangehenden Beispiel, mit der selben Ausgabe. Falls der Server noch am Laufen ist, können Sie natürlich diesen Schritt überspringen.

1.10.2.4. Kreieren des administrierten Objekts (Topic)

Dieses geschieht am Besten mit dem Batch Skript:

```
@echo off
Rem
Rem   Starten der J2EE
Rem
set J2EE_HOME=c:\j2sdkee1.3
c:
cd %J2EE_HOME%\bin
@echo Eintrag des Topics
@j2eeadmin -addJmsDestination MeinTopic topic
pause
```

und gleich überprüfen, ob der Eintrag korrekt ausgeführt wurde:

```
@echo off
Rem
Rem   Starten der J2EE
Rem
set J2EE_HOME=c:\j2sdkee1.3
c:
cd %J2EE_HOME%\bin
@echo Ueberpruefen des Contexts
start j2eeadmin -listJmsDestination
D:
```

mit der Ausgabe (ich habe die Warteschlange nach dem ersten Beispiel nicht gelöscht):

```
JmsDestination
--------------
< JMS Destination : meineWarteschlange , javax.jms.Queue >
< JMS Destination : MeinTopic , javax.jms.Topic >
< JMS Destination : jms/Topic , javax.jms.Topic >
< JMS Destination : jms/Queue , javax.jms.Queue >
```

1.10.2.5. Starten des Subscribers

Da wir nicht angegeben haben, dass die Meldungen gespeichert werden sollen, müssen wir den Subscriber Client vor dem Publisher Client starten, da sonst einfach alle Meldungen verloren gehen würden, die vor dem Start des Subscribers im Topic eingetragen werden!

Der Subscriber wird mit folgendem Script gestartet:

```
@echo off
echo Starten des JMS Beispiels. Bitte warten ...
Rem
set J2EE_HOME=c:\j2sdkee1.3
Rem
cd EinfacherTopicSubscriber
%JAVA_HOME%\bin\java -classpath
c:\jbuilder4\jdk1.3\lib;%J2EE_HOME%\lib\j2ee.jar;%J2EE_HOME%\lib\locale;%J2
EE_HOME%\lib\;.;..  -
Djms.properties=%J2EE_HOME%\config\jms_client.properties
einfachertopicsubscriber.EinfacherTopicSubscriber MeinTopic
cd ..
pause
```

mit folgender Ausgabe:

```
Starten des JMS Beispiels. Bitte warten ...
Name des Topics : MeinTopic
Inhalt des Contexts:
Java(TM) Message Service 1.0.2 Reference Implementation (build b13)
Comnnection Metadata :
                       JMSMajorVersion=1
                       JMSMinorVersion=0
                       JMSProviderName=Sun Microsystems
                       JMSVersion=1.0
                       JMSXPropertyVersion=java.util.Vector$1@45f743
                       JMSXProviderVersion=java.util.Vector$1@6c8909
Beenden des Programms mit Q oder q, und <return>
```

Nach dem Starten des Publishers geht's folgendermassen weiter:

```
Beenden des Programms mit Q oder q, und <return>
Message wird gelesen: Einfache Text Message 1
Message wird gelesen: Einfache Text Message 2
Message wird gelesen: Einfache Text Message 3
..
Message wird gelesen: Einfache Text Message 9
Message wird gelesen: Einfache Text Message 10
Message wird gelesen: Einfache Text Message 18
Message wird gelesen: Einfache Text Message 19
Message wird gelesen: Einfache Text Message 20
```

1.10.2.6. Starten des Publishers

Dazu verwenden wir das folgende Skript, damit alle Properties korrekt gesetzt werden:

```
@echo off
echo Starten des JMS Beispiels. Bitte warten ...
Rem
set J2EE_HOME=c:\j2sdkee1.3
Rem
cd EinfacherTopicPublisher
%JAVA_HOME%\bin\java -classpath
c:\jbuilder4\jdk1.3\lib;%J2EE_HOME%\lib\j2ee.jar;%J2EE_HOME%\lib\locale;%J2
EE_HOME%\lib\system\tool.jar;.;..  -
Djms.properties=%J2EE_HOME%\config\jms_client.properties
einfachertopicpublisher.EinfacherTopicPublisher MeinTopic 20
cd ..
pause
```

Dies produziert folgende Ausgabe:

```
Starten des JMS Beispiels. Bitte warten ...
Der Name des Topics ist MeinTopic
Kreieren des JDNI Initial Contexts
Lookup der Connection Factory und des Topics
createTopicConnection
Java(TM) Message Service 1.0.2 Reference Implementation (build b13)
createTopicSession
createPublisher
createTextMessage
Message wird publiziert: Einfache Text Message 1
publish()
Message wird publiziert: Einfache Text Message 2
publish()
Message wird publiziert: Einfache Text Message 3
publish()
...
Message wird publiziert: Einfache Text Message 10
publish()
Message wird publiziert: Einfache Text Message 11
publish()
...
Message wird publiziert: Einfache Text Message 18
publish()
Message wird publiziert: Einfache Text Message 19
publish()
Message wird publiziert: Einfache Text Message 20
publish()
Taste drücken, um fortzusetzen . . .
```

1.10.2.7. Löschen des Topics

Dieses geschieht wie im vorangehenden Beispiel mit einem vorbereiteten Skript:

```
@echo off
Rem   Stoppen der J2EE : Topic löschen
Rem
set J2EE_HOME=c:\j2sdkee1.3
c:
cd %J2EE_HOME%\bin
start j2eeadmin -removeJmsDestination MeinTopic
```

1.11. Weitere Beispiele

Die folgenden Beispiele stammen aus der JMS Implementation von Sun (Forte) , eine eher schlechte Implementierung.

Die folgenden Beispiele zeigen spezifische Funktionalitäten, wie sie ein JMS System typischerweise zur Verfügung stellen:

- "Connections"
- "Durability"
- "Persistence"
- "Producer-Consumer"
- "Selectors"
- "Simplechat"
- "Transactions"

1.11.1. Connections

Diese Beispiel demonstriert den grundsätzlichen Verbindungsaufbau mit anschliessendem Senden und Empfangen von Nachrichten mit Hilfe von Topics und Queues. Ein Client sendet eine Nachricht an ein bestimmtes Topic oder an eine bestimmte Warteschlange.

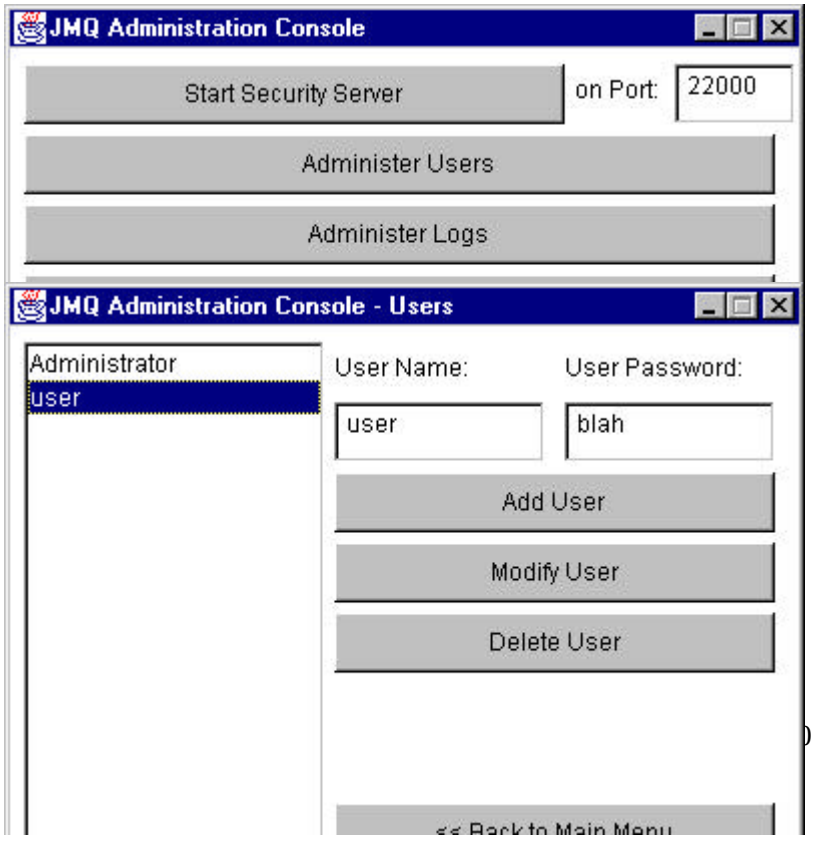
In diesem Beispiel wird zuerst ein Topic Beispiel kreiert, anschliessend ein Queue Beispiel.

Als nächstes wird TopicConnections und TopicSessions kreiert. Eine TopicConnection ist eine aktive Verbindung zum JMS Publish-Subscribe Provider.

Ein Client kreiert eine TopicConnection um eine oder mehrere TopicSessions zum Produzieren und Konsumieren von Nachrichten zur Verfügung zu haben.

Das Beispiel zeigt auch eine einfache Administrator-Konsolenapplikation und einen Security Server. Die Verbindung wird für den Benutzer user mit dem Passwort blah kreiert.

Der Security Server muss an Port 24000 gestartet werden (Default : 22000). Der Benutzer kann mit Hilfe des Administrators kreiert werden.



Administratorkonsole für die Verwaltung der Benutzer unseres Demo Systems.

Ausführungsprotokoll:

```
Creating topic connection
Creating topic session: not transacted, auto ack
Creating topic, publisher and subscriber...
Creating and sending message...
Receiving message...
Topic TextMessage test successful!
Closing publisher and subscriber...
Closing topic session and topic connection
Socket read exception occurred: java.net.SocketException: JVM_recv in
socket input stream read (code=10004) IAReadChannel(SocketRead) cannot read
socket
Creating queue connection
Creating queue session: not transacted, auto ack
Creating queue, sender and receiver...
Creating and sending message...
Receiving message...
Queue TextMessage test successful!
Closing sender and receiver...
Closing queue session and queue connection
Socket read exception occurred: java.net.SocketException: JVM_recv in
socket input stream read (code=10004) IAReadChannel(SocketRead) cannot read
socket
```

Programmcode

```
/**
 * This example is designed to demonstrate how to set up a basic
 * connection to send and receive messages using topics and queues.
 *
 * This test also demonstrates how to use the JMSSecurityServer.
 *
 * To do this using Topics or Queues
 * involves the following 10 steps:
 *          Start the router
 *          Create a *ConnectionFactory
 *          Create a *Connection
 *          Create a *Session
 *          Create a Topic (Queue) to Publish / Subscribe to
 *          Create a Publisher (Sender)
 *          Create a Subscriber (Receiver)
 *          Create a Message
 *          Publish (Send) the Message
 *          Subscribe (Receive) the Message
 *
*/


import javax.jms.*;


public class Connections {

  TopicConnectionFactory     topicConnectionFactory;
  TopicConnection            topicConnection;
  TopicSession               topicSession;
  TopicPublisher             topicPublisher;
  TopicSubscriber            topicSubscriber;
  Topic                            topic;

  QueueConnectionFactory     queueConnectionFactory;
  QueueConnection            queueConnection;
  QueueSession               queueSession;
  QueueSender                      queueSender;
  QueueReceiver              queueReceiver;
  Queue                            queue;

  String[]                   arguments;
  String                     testString1;
  Thread                     mythread;


  /**
   * Default Constructor
   */

  public Connections() {
  }


  /** Create the connection factories. Begin the example.
   *
   * @param args JMQ startup properties
   */

  public Connections(String[] args) {
    queueConnectionFactory = new
com.sun.messaging.QueueConnectionFactory(args);
```

```
    //The second Argument tells the ConnectionFactory to create connections
that
    //will try to access the JMSSecurityServer on port 24000
    topicConnectionFactory = new
com.sun.messaging.TopicConnectionFactory(args, 24000);

    example();
  }


  /** Create the program.
   *
   * @param args JMQ startup properties
   */

  public static void main(String args[]) {
    Connections c = new Connections(args);
  }


  /**
   * Sleep for a specified time.
   *
   * @param time Time in milliseconds to wait.
   */

  public void sleep(int time) {
    try {
      Thread.sleep(time);
    }
    catch(Exception e) {
    }
  }


  /**
   * Complete the topic example, follwed by the queue example, then exit
the program.
   */

  public void example() {
    //Create a TopicConnection, TopicSession
    t11();
    //Demonstrate Sending and Receiving Topic Messages
    tmessage();
    //Close the TopicConnection, TopicSession
    tc();

    //Create a QueueConnection, QueueSession
    q11();
    //Demonstrate Sending and Receiving Queue Messages
    qmessage();
    //Close the QueueConnection, QueueSession
    qc();

    System.exit(0);
  }


  /**
   * Create the TopicConnection and TopicSession
```

```
   */

  public void t11() {
    try {
      //Create a Connection with the username "user" and password "blah"
      //This username and password are defaults in the README file
      //Which is used by the JMSSecurityServer for validating users and
their passwords.
      topicConnection =
topicConnectionFactory.createTopicConnection("user", "blah");
      topicConnection.start();
      System.out.println("Creating topic connection");
      //creating Topic Session
      //    Transaction Mode: None
      //    Acknowledge Mode: Automatic
      topicSession = topicConnection.createTopicSession(false,1);
      System.out.println("Creating topic session: not transacted, auto
ack");
    }
    catch(Exception e) {
      System.out.println("Exception, could not create topic connection or
session: " +
                  e.getMessage() + "");
      e.printStackTrace();
    }
  }


  /**
   * Send a TextMessage using Topics
   */

  public void tmessage() {
    try {
      System.out.println("Creating topic, publisher and subscriber...");
      // Create a topic, a publisher, a subscriber
      topic = topicSession.createTopic("ConnectionsExampleJTopic");
      topicPublisher = topicSession.createPublisher(topic);
      topicSubscriber = topicSession.createSubscriber(topic);

      System.out.println("Creating and sending message...");
      // Use the Session to create a message
      TextMessage textmsg1 = topicSession.createTextMessage();
      // Set the message properties
      testString1 = "Topic Message Test";
      textmsg1.setText(testString1);
      // Send the message
      topicPublisher.publish(textmsg1);

      System.out.println("Receiving message...");
      // Manually receive the message
      TextMessage textmsg2 = (TextMessage)topicSubscriber.receive();
      // Test if the message is same as the one sent
      if(textmsg2.getText().equals(testString1)) {
      System.out.println("Topic TextMessage test successful!");
      }
      else {
      System.out.println("Topic TextMessage test failed!");
      }

      // Close the publisher, the subscriber
      System.out.println("Closing publisher and subscriber...");
```

```
      topicPublisher.close();
      topicSubscriber.close();
    }
    catch(JMSException e) {
      System.out.println("JMSException in tmessage(): " + e + "");
      e.printStackTrace();
    }
  }


  /**
   *  Create the QueueConnection and QueueSession
   */

  public void q11() {
    try {
      queueConnection = queueConnectionFactory.createQueueConnection();
      queueConnection.start();
      System.out.println("Creating queue connection");
      //creating Queue Session
      //    Transaction Mode: None
      //    Acknowledge Mode: Automatic
      queueSession = queueConnection.createQueueSession(false,1);
      System.out.println("Creating queue session: not transacted, auto
ack");
    }
    catch(Exception e) {
      System.out.println("Exception, could not create queue connection or
session: " +
                   e.getMessage() + "");
      e.printStackTrace();
    }
  }


  /**
   *  Send a TextMessage using Queues
   */

  public void qmessage() {
    try {
      System.out.println("Creating queue, sender and receiver...");
      // Create a queue, a sender, a receiver
      queue = queueSession.createQueue("ConnectionsExampleJQueue");
      queueSender = queueSession.createSender(queue);
      queueReceiver = queueSession.createReceiver(queue);

      System.out.println("Creating and sending message...");
      // Use the Session to create a message
      TextMessage textmsg1 = queueSession.createTextMessage();
      // Set the message properties
      testString1 = "Queue Message Test";
      textmsg1.setText(testString1);
      // Send the message
      queueSender.send(textmsg1);

      System.out.println("Receiving message...");
      // Manually receive the message
      TextMessage textmsg2 = (TextMessage)queueReceiver.receive();
      // Test if the message is same as the one sent
      if(textmsg2.getText().equals(testString1)) {
      System.out.println("Queue TextMessage test successful!");
```

```
      }
      else {
      System.out.println("Queue TextMessage test failed!");
      }

      // Close the sender, the receiver
      System.out.println("Closing sender and receiver...");
      queueSender.close();
      queueReceiver.close();
    }
    catch(JMSException e) {
      System.out.println("JMSException in qmessage(): " + e + "");
      e.printStackTrace();
    }
  }


  /**
   *  Close the topic session and topic connection
   */

  public void tc() {
    try {
      System.out.println("Closing topic session and topic connection");
      topicSession.close();
      topicConnection.close();
    }
    catch(Exception e) {
      System.out.println("Exception, could not close topic session or topic
connection: "+
                  e.getMessage() + "");
      e.printStackTrace();
    }
  }


  /**
   *  Close the queue session and queue connection
   */

  public void qc() {
    try {
      System.out.println("Closing queue session and queue connection");
      queueSession.close();
      queueConnection.close();
    }
    catch(Exception e) {
      System.out.println("Exception, could not close queue session or queue
connection: "+
                  e.getMessage() + "");
      e.printStackTrace();
    }
  }
}
```

1.11.2. Durability

In diesem Beispiel publizieren und abonnieren Clients Nachrichen zu einem Topic Knoten.

Subscribers können auch als dauerhaft spezifiziert werden, so dass auch Nachrichten abgeliefert werden, die eintrafen, als der Client nicht aktiv war.

Prozessverlauf des Beispiels:

- starten des Router
- kreieren eines Topics, Publishers, Subscribers und Messages
- publizieren der Messages
- durable Subscriber starten
- disconnect und reconnect vom Router

Der Subscriber erhält weiter Nachrichten.

Ausführungsprotokoll:

```
Creating topic connection
Creating topic session: not transacted, auto ack
Creating durable subscriber
Sending 3 messages!
Receive first message!
ObjectMessage1 test successful!
-----------------------
Forcing disconnect!
Closing topic session and topic connection
Socket read exception occurred: java.net.SocketException: JVM_recv in
socket input stream read (code=10004) IAReadChannel(SocketRead) cannot read
socket
-----------------------
Attempting to reconnect!
Creating durable subscriber
Attempting to receive last 2 messages!
ObjectMessage2 test successful!
ObjectMessage3 test successful!
Closing sender and receiver...
Unsubscribing Durable Interest...
Closing topic session and topic connection
Socket read exception occurred: java.net.SocketException: JVM_recv in
socket input stream read (code=10004) IAReadChannel(SocketRead) cannot read
socket
```

Programmcode:

```
/**
 * This example is designed to demonstrate durability
 *
 * To do this using Topics
 * we will demonstrated using the following steps:
 *          Start the router
 *          Create a ConnectionFactory
 *          Create a Connection
 *          Create a Session
 *          Create a Topic (Queue) to Publish / Subscribe to
 *          Create a Publisher (Sender)
 *          Create a Durable Subscriber (Receiver)
 *          Create a Message
 *          Publish (Send) 3 Messages
 *          Subscribe (Receive) the first Message
 *          Fully DISCONNECT from the router
 *          Reconnect to the router
 *                creating a connection, session, topic, durable subscriber
 *          Subscribe (Receive) the last 2 Messages
 *
 * Note that Queues are durable by default
 *
*/


import javax.jms.*;


public class Durability {

  TopicConnectionFactory     topicConnectionFactory,
SecondtopicConnectionFactory;
  TopicConnection       topicConnection;
  TopicSession               topicSession;
  TopicPublisher             topicPublisher;
  TopicSubscriber            topicSubscriber;
  Topic                           topic;

  String[]                   arguments;
  String                     testString1;
  Thread                     mythread;


  /**
   *  Default Constructor
   */
  public Durability() {
  }


  /**
   * Create the connection factories. Begin the example.
   *
   * @param args JMQ startup properties
   */

  public Durability(String[] args) {
    topicConnectionFactory = new
com.sun.messaging.TopicConnectionFactory(args);
```

```
    SecondtopicConnectionFactory = new
com.sun.messaging.TopicConnectionFactory(args);
    example();
  }


  /**
   * Create the program.
   *
   * @param args JMQ startup properties
   */

  public static void main(String args[]) {
    Durability d = new Durability(args);
  }


  /**
   * Sleep for a specified time.
   *
   * @param time Time in milliseconds to wait.
   */

  public void sleep(int time) {
    try {
      Thread.sleep(time);
    }
    catch(Exception e) {
    }
  }


  /**
   * Complete the topic example, follwed by the queue example, then exit
the program.
   */

  public void example() {
    //Create a TopicConnection, TopicSession
    t11();
    //Demonstrate Sending and Receiving Topic Messages
    tmessage();
    //Close the TopicConnection, TopicSession
    tc();

    System.exit(0);
  }


  /**
   * Create the TopicConnection and TopicSession
   */

  public void t11() {
    try {
      topicConnection = topicConnectionFactory.createTopicConnection();
      topicConnection.start();
      System.out.println("Creating topic connection");
      //creating Topic Session
      //    Transaction Mode: None
      //    Acknowledge Mode: Automatic
      topicSession = topicConnection.createTopicSession(false,1);
```

```
      System.out.println("Creating topic session: not transacted, auto
ack");
    }
    catch(Exception e) {
      System.out.println("Exception, could not create topic connection or
session: " +
                   e.getMessage() + "");
      e.printStackTrace();
    }
  }


  /**
   * Send a TextMessage using Topics
   */

  public void tmessage() {
    String testString1 = "String Object Message1";
    String testString2 = "String Object Message2";
    String testString3 = "String Object Message3";
    int xyz;
    xyz = (int)(Math.floor(Math.random()*100));
    //  System.out.println("Random is " +xyz);

    try {
      // JMS pub-sub init
      Topic topic = topicSession.createTopic("DurabilityExampleJTopic"
+xyz);
      topicPublisher = topicSession.createPublisher(topic);
      System.out.println("Creating durable subscriber");
      topicSubscriber = topicSession.createDurableSubscriber(topic,
                                              "DurabilityExamplename"+xyz);

      // Send 3
      System.out.println("Sending 3 messages!");
      ObjectMessage objmsg1 = topicSession.createObjectMessage();
      ObjectMessage objmsg2 = topicSession.createObjectMessage();
      ObjectMessage objmsg3 = topicSession.createObjectMessage();
      objmsg1.setObject(testString1);
      objmsg2.setObject(testString2);
      objmsg3.setObject(testString3);
      topicPublisher.publish(objmsg1);
      topicPublisher.publish(objmsg2);
      topicPublisher.publish(objmsg3);

      // Recv 1
      System.out.println("Receive first message!");
      ObjectMessage objmsg4 = (ObjectMessage)topicSubscriber.receive();
      if (objmsg4.getObject().equals(testString1)){
      System.out.println("ObjectMessage1 test successful!");
      }
      else{
      System.out.println("ObjectMessage test failed!");
      }

      // Force Disconnect
      System.out.println("------------------------");
      System.out.println("Forcing disconnect!");
      tc();
      System.out.println("------------------------");

      // Pause
```

```
      sleep(1000);

      // Reconnect
      System.out.println("Attempting to reconnect!");
      topicConnection =
SecondtopicConnectionFactory.createTopicConnection();
      topicConnection.start();
      topicSession = topicConnection.createTopicSession(false,1);
      topic = topicSession.createTopic("DurabilityExampleJTopic" +xyz);
      System.out.println("Creating durable subscriber");
      topicSubscriber = topicSession.createDurableSubscriber(topic,
                                              "DurabilityExamplename"+xyz);

      // Recv Last 2
      System.out.println("Attempting to receive last 2 messages!");
      ObjectMessage objmsg5 = (ObjectMessage)topicSubscriber.receive();
      if (objmsg5.getObject().equals(testString2)){
      System.out.println("ObjectMessage2 test successful!");
      }
      else{
      System.out.println("ObjectMessage2 test failed!");
      }
      ObjectMessage objmsg6 = (ObjectMessage)topicSubscriber.receive();
      if (objmsg6.getObject().equals(testString3)){
      System.out.println("ObjectMessage3 test successful!");
      }
      else{
      System.out.println("ObjectMessage3 test failed!");
      }

      // Close the sender, the receiver
      System.out.println("Closing sender and receiver...");
      topicPublisher.close();
      topicSubscriber.close();

      // Unsubscribe the durable interest
      System.out.println("Unsubscribing Durable Interest...");
      topicSession.unsubscribe("DurabilityExamplename");
    }
    catch(JMSException e) {
      System.out.println("JMSException in tmessage(): " + e + "");
      e.printStackTrace();
    }
  }


  /**
   * Close the topic session and topic connection
   */

  public void tc() {
    try {
      System.out.println("Closing topic session and topic connection");
      topicSession.close();
      topicConnection.close();
    }
    catch(Exception e) {
      System.out.println("Exception, could not close topic session or topic
connection: "+
                   e.getMessage() + "");
      e.printStackTrace();
    }
```

```
   }
}
```

1.11.3. Persistence

Dieses Beispiel demonstriert die Persistenz einer gesendeten Nachricht.

Per Default sind JMS Messages persistent und der Java Message Router speichert sie auf Disk.

Ausführungsprotokoll:

```
Creating topic connection
Creating topic session: not transacted, auto ack
Creating Durable Subscriber
Sending 3 messages!
Receive first message!
ObjectMessage1 test successful!
Forcing disconnect!
Closing topic session and topic connection
Socket read exception occurred: java.io.EOFException
IAReadChannel(SocketRead) cannot read socket
------------------------

Waiting 30 seconds - STOP & RESTART the irouter now!

If you type fast enough, look in the resource directory and see the 2
messages
es handelt sich um das Verzeichnis ...res\sys\ap...
die Messages werden serialisiert abgespeichert zum Beispiel als 18.dat
------------------------
Attempting to reconnect!
Creating Durable Subscriber
Attempting to receive last 2 messages!
ObjectMessage2 test successful!
ObjectMessage3 test successful!
Closing sender and receiver...
Unsubscribing Durable Interest...
Closing topic session and topic connection
Socket read exception occurred: java.net.SocketException: JVM_recv in
socket input stream read (code=10004) IAReadChannel(SocketRead) cannot read
socket
```

Programmcode

```
/**
 * This example is designed to demonstrate persistence
 *
 * To do this using Topics,
 * we will demonstrate using the following steps:
 *          Start the router
 *          Create a ConnectionFactory
 *          Create a Connection
 *          Create a Session
 *          Create a Topic (Queue) to Publish / Subscribe to
 *          Create a Publisher (Sender)
 *          Create a Subscriber (Receiver)
 *          Create a Message
 *          Publish (Send) 3 Messages
 *          Subscribe (Receive) the first Message
 *          Fully DISCONNECT from the router
 *
 *          At this point, the user must stop the router
 *          then restart it.  There will be a 20 second delay then...
 *
 *          This program will attempt to reconnect to the router
 *                creating a connection, session, topic, subscriber
 *          Subscribe (Receive) the last 2 Messages
 *
 * Note that all outgoing messages are persistent by default
 * and that under the resource directory (/res) there will be
 * messages stored until the client acknowledges to the router
 * that it has received them.
*/


import javax.jms.*;


public class Persistence {

  TopicConnectionFactory     topicConnectionFactory,
SecondtopicConnectionFactory;
  TopicConnection      topicConnection;
  TopicSession                    topicSession;
  TopicPublisher             topicPublisher;
  TopicSubscriber            topicSubscriber;
  Topic                           topic;

  String[]                     arguments;
  String                     testString1;
  Thread               mythread;


  /**
   * Default Constructor
   */

  public Persistence() {
  }


  /**
   * Create the connection factories. Begin the example.
```

```
   *
   * @param args JMQ startup properties
   *
   */
  public Persistence(String[] args) {
    topicConnectionFactory = new
com.sun.messaging.TopicConnectionFactory(args);
    SecondtopicConnectionFactory = new
com.sun.messaging.TopicConnectionFactory(args);
    example();
  }


  /**
   * Create the program.
   *
   * @param args JMQ startup properties
   */

  public static void main(String args[]) {
    Persistence p = new Persistence(args);
  }


  /**
   * Sleep for a specified time.
   *
   * @param time Time in milliseconds to wait.
   */

  public void sleep(int time) {
    try {
      Thread.sleep(time);
    }
    catch(Exception e) {
    }
  }


  /**
   *  Complete the topic example, follwed by the queue example, then exit
the program.
   */

  public void example() {
    //Create a TopicConnection, TopicSession
    t11();
    //Demonstrate Sending and Receiving Topic Messages
    tmessage();
    //Close the TopicConnection, TopicSession
    tc();

    System.exit(0);
  }


  /**
   * Create the TopicConnection and TopicSession
   */

  public void t11() {
    try {
```

```
      topicConnection = topicConnectionFactory.createTopicConnection();
      topicConnection.start();
      System.out.println("Creating topic connection");
      //creating Topic Session
      //    Transaction Mode: None
      //    Acknowledge Mode: Automatic
      topicSession = topicConnection.createTopicSession(false,1);
      System.out.println("Creating topic session: not transacted, auto
ack");
    }
    catch(Exception e) {
      System.out.println("Exception, could not create topic connection or
session: " +
                   e.getMessage() + "");
      e.printStackTrace();
    }
  }


  /**
   * Send a TextMessage using Topics
   */

  public void tmessage() {
    String testString1 = "String Object Message1";
    String testString2 = "String Object Message2";
    String testString3 = "String Object Message3";
    int xyz;
    xyz = (int)(Math.floor(Math.random()*100));
    //System.out.println("Random is " +xyz);

    try {
      // JMS pub-sub init
      Topic topic = topicSession.createTopic("PersistenceExampleJTopic" +
xyz);
      topicPublisher = topicSession.createPublisher(topic);
      System.out.println("Creating Durable Subscriber");
      topicSubscriber = topicSession.createDurableSubscriber(topic,

"PersistenceExamplename"+xyz);

      // Send 3
      System.out.println("Sending 3 messages!");
      ObjectMessage objmsg1 = topicSession.createObjectMessage();
      ObjectMessage objmsg2 = topicSession.createObjectMessage();
      ObjectMessage objmsg3 = topicSession.createObjectMessage();
      objmsg1.setObject(testString1);
      objmsg2.setObject(testString2);
      objmsg3.setObject(testString3);
      topicPublisher.publish(objmsg1);
      topicPublisher.publish(objmsg2);
      topicPublisher.publish(objmsg3);

      // Recv 1
      System.out.println("Receive first message!");
      ObjectMessage objmsg4 = (ObjectMessage)topicSubscriber.receive();
      if (objmsg4.getObject().equals(testString1)){
      System.out.println("ObjectMessage1 test successful!");
      }
      else{
      System.out.println("ObjectMessage test failed!");
      }
```

```
      // Force Disconnect/Stop Router/Restart Router
      System.out.println("Forcing disconnect!");
      tc();
      System.out.println("------------------------");
      System.out.println("\nWaiting 30 seconds - STOP & RESTART the irouter
now!\n");
      System.out.println("If you type fast enough, look in the resource
directory and see the 2 messages\n");
      System.out.println("------------------------");
      sleep(30000);

      // Reconnect
      System.out.println("Attempting to reconnect!");
      topicConnection =
SecondtopicConnectionFactory.createTopicConnection();
      topicConnection.start();
      topicSession = topicConnection.createTopicSession(false,1);
      topic = topicSession.createTopic("PersistenceExampleJTopic" + xyz);
      System.out.println("Creating Durable Subscriber");
      topicSubscriber = topicSession.createDurableSubscriber(topic,

"PersistenceExamplename"+xyz);

      // Recv Last 2
      System.out.println("Attempting to receive last 2 messages!");
      ObjectMessage objmsg5 = (ObjectMessage)topicSubscriber.receive();
      if (objmsg5.getObject().equals(testString2)){
      System.out.println("ObjectMessage2 test successful!");
      }
      else{
      System.out.println("ObjectMessage2 test failed!");
      }
      ObjectMessage objmsg6 = (ObjectMessage)topicSubscriber.receive();
      if (objmsg6.getObject().equals(testString3)){
      System.out.println("ObjectMessage3 test successful!");
      }
      else{
      System.out.println("ObjectMessage3 test failed!");
      }

      // Close the sender, the receiver
      System.out.println("Closing sender and receiver...");
      topicPublisher.close();
      topicSubscriber.close();

      // Unsubscribe the durable interest
      System.out.println("Unsubscribing Durable Interest...");
      topicSession.unsubscribe("PersistenceExamplename");
    }
    catch(JMSException e) {
      System.out.println("JMSException in tmessage(): " + e + "");
      e.printStackTrace();
    }
  }


  /**
   * Close the topic session and topic connection
   */

  public void tc() {
```

```
    try {
      System.out.println("Closing topic session and topic connection");
      topicSession.close();
      topicConnection.close();
    }
    catch(Exception e) {
      System.out.println("Exception, could not close topic session or topic
connection: "+
                  e.getMessage() + "");
      e.printStackTrace();
    }
  }
}
```

1.11.4. Producer-Consumer

Diese Beispiel zeigt die Unterstützung von Warteschlangen.

Ein Point-to-point (queue) Distributionsmodell wird verwendet. Das Beispiel zeigt, wie Properties eingesetzt werden können und wie damit Nachrichten ausgwählt / selektiert werden können.

Das Point-to-point Model funktioniert grob folgendermassen:

- Sender produzieren Messages
- Empfänger konsumieren Messages
- Warteschlangen dienen als Briefkasten
- Listeners, welche einenEmpfänger darüber informieren, dass neue Nachrichten in der Warteschlange sind

Der Empfänger muss den Listener kreieren, mit Hilfe des MessageListener Interfaces.
Der Receiver kann immer noch entscheiden, ob er über neu angekommene Nachrichten informiert werden möchte.

Sobald der Listener seinen Kunden benachrichtigt hat holt der Kunde die Nachricht mit der Methode onMessage(Message) ab.

Das Beispiel implementiert dieses System mit zwei Threads.

Ausführungsprotokoll

```
>> Creating a new connection factory
>> Creating a new connection
       JMS Major Version = 1
       JMS Minor Version = 0
       JMS Provider Name = Java Message Queue, Sun Microsystems, Inc.
       JMS Version = 1.0.1
       JMS Provider Major Ver = 1
       JMS Provider Minor Ver = 0
       JMS Provider Ver = 1.0.1
Support for JMSX properties is Optional according to JMS Spec
JMSX Properties Supported here are ...
      JMSXRcvTimestamp = true
      JMSXConsumerTXID = true
      JMSXProducerTXID = true
      JMSXAppID = true
      JMSXUserID = true
>>> Creating a new queue session
>>> Creating a new Q - Example JMQ
Producer>>> Message 1
Consumer<<< Message # n Received message body: Producer Message # 0
JMS Message Header Properties ...
       Message Destination   = Example JMQ
       Message Delivery mode = 2
       Message ID            = ID:PHYSICAL:ztnw293/DEF_CLASS:[-818783025,
0][3476184781, 1]
       Message TimeStamp     = 961253892400
       Correlation ID        = null
       Reply To this message @ null
       Message Redelivered ? = false
```

```
       Message Expiration    = 0
       Message Priority      = 4
       Message Type          = null
       Property Names :
            PT_TAG, JMSXRcvTimestamp, JMSXAppID, PT_DATACLASS, PT_DELIVERY,
PT_SRCDOMAIN, PT_LIFETIME, PT_ACKNDX, PT_SEQUENCE, PT_INDPATH, PT_NOTIFY,
       Values of Properties are :
             PT_NOTIFY    = 61440
             PT_ACKNDX    = [7, 0]
             PT_DELIVERY  = modulus.iagent.IAObjectDelivery@21cc40
             PT_INDPATH   = [1, 1]:ztnw293/DEF_CLASS:[277, 0]
             PT_SEQUENCE  = 1
             PT_DATACLASS = [3439186197, 139]
             PT_SRCDOMAIN = PHYSICAL:ztnw293/DEF_CLASS:[-818783025, 0]
             PT_LIFETIME  = modulus.iagent.IAObjectLifetime@6893df
             PT_TAG       = [3476184781, 1]
             JMSXUserID        = null
             JMSXAppID         = PHYSICAL:ztnw293/DEF_CLASS:[-818783025, 0]
             JMSXDeliveryCount = null
             JMSXGroupID       = null
             JMSXGroupSeq      = null
             JMSXProducerTXID  = null
             JMSXConsumerTXID   = null
             JMSXRcvTimestamp  = 961253892721
             JMSXState         = null
Consumer Message Ack sent
Sleep for 3 secs
Done Sleeping
Closing Producer queue session
```

Programmcode

QPC.java

```
/**
* This is a Simple Producer-Consumer Example.
 * Here 2 threads are created - one for producer and one for consumer.
 * The Producer produces a message and the consumer using the
MessageListener
 * property consumes the message.
 * Also demonstrated : <BR>
 * <UL>
 * <LI>The ability to enable JMS Defined Properties (JMSX)</LI>
 * <LI>The way to read the JMSX properties</LI>
 * <LI>Getting JMS version information</LI>
 * <LI>Creating Queues, Starting and Closing QueueConnection and
Session.</LI>
 *</UL>
*/

import javax.jms.*;

public class QPC {

  public static void main(String[] args) {
    try {
      System.out.println (">> Creating a new connection factory");
      com.sun.messaging.QueueConnectionFactory aQConnectionFactory =
      new com.sun.messaging.QueueConnectionFactory (args);

      // No JMSX Property is enabled by default.  This is to reduce packet
size.
      // Enable the properties that you wish.
      aQConnectionFactory.enableJMSX("JMSXUserID");
      aQConnectionFactory.enableJMSX("JMSXAppID");
      aQConnectionFactory.enableJMSX("JMSXDeliveryCount");
      aQConnectionFactory.enableJMSX("JMSXGroupID");
      aQConnectionFactory.enableJMSX("JMSXGroupSeq");
      aQConnectionFactory.enableJMSX("JMSXProducerTXID");
      aQConnectionFactory.enableJMSX("JMSXConsumerTXID");
      aQConnectionFactory.enableJMSX("JMSXRcvTimestamp");
      aQConnectionFactory.enableJMSX("JMSXState");

      // create a connection
      System.out.println (">> Creating a new connection ");
      QueueConnection aQConnection =
aQConnectionFactory.createQueueConnection ();
      // All connections must be started,
      // otherwise you will get errors while running
      aQConnection.start();

      // get the ConnectionMetaData details so as to grab some important
information.
      ConnectionMetaData qcmd =  aQConnection.getMetaData();

      // Print the JMS version strings
      System.out.println("\t JMS Major Version = " +
qcmd.getJMSMajorVersion());
      System.out.println("\t JMS Minor Version = " +
qcmd.getJMSMinorVersion());
```

```
      System.out.println("\t JMS Provider Name = " +
qcmd.getJMSProviderName());
      System.out.println("\t JMS Version = " + qcmd.getJMSVersion());
      System.out.println("\t JMS Provider Major Ver = " +
                  qcmd.getProviderMajorVersion());
      System.out.println("\t JMS Provider Minor Ver = " +
                  qcmd.getProviderMinorVersion());
      System.out.println("\t JMS Provider Ver = " +
qcmd.getProviderVersion());

      String jmsx_prop[] = qcmd.getJMSXPropertyNames();
      if (jmsx_prop.length > 0) {
      System.out.println("Support for JMSX properties is Optional according
to JMS Spec");
      System.out.println("JMSX Properties Supported here are ... ");
      for (int k=0; k < jmsx_prop.length; k++) {
        System.out.println("\t"+jmsx_prop[k] + " = " +
aQConnectionFactory.getJMSX(jmsx_prop[k]));
      }
      }
      else {
      System.out.println("No JMSX Properties Found !!");
      }

      System.out.println (">>> Creating a new queue session ");
      // Here we use the CLIENT_ACKNOWLEDGE property so that we can
demostrate the
      // receiver's MessageListener propety.
      QueueSession aQSession =
      aQConnection.createQueueSession(false,
                              QueueSession.CLIENT_ACKNOWLEDGE );
      aQSession.recover();

      //create a queue with a Queue Name
      String Q_Name = new String("Example JMQ");
      System.out.println (">>> Creating a new Q - " + Q_Name);
      Queue aQ = aQSession.createQueue (Q_Name);

      QProducer p1 = new QProducer(aQSession, aQ);
      QConsumer c1 = new QConsumer(aQSession, aQ);

      p1.start();
      c1.start();
      // I am waiting on when the producer closes the session, so that I
can
      // close the consumer session, queue and the connection - free
resources.
      while (p1.SESSION_STATUS)  {
      //    continue ;
      }
      if (! p1.SESSION_STATUS)  {
      c1.close();
      aQConnection.close();
      }
    }
    catch (JMSException jmse) {
      jmse.printStackTrace();
    }
  }
}
```

QProducer.java

```
import javax.jms.*;

public class QProducer extends Thread {

  public static boolean SESSION_STATUS = true;

  //Change the below number if you want to send more messages everytime.
  public static final int NO_MSG_TO_SEND = 1;

  private QueueSession             theSession;
  private QueueSender         theQSender;

  public QProducer(QueueSession aqs, Queue aq) {
    theSession = aqs;
    try {
      theQSender = theSession.createSender(aq);
    }
    catch (JMSException jmse) {
      System.out.println("Producer Constructor " + jmse);
    }
  }

  public void run() {
    StreamMessage aMessage;
    int messageCount;
    try {
      for (int i = 0; i < NO_MSG_TO_SEND; i++) {
      int j = i+1;
      System.out.println ("Producer>>> Message " + j);
      aMessage = theSession.createStreamMessage();
      aMessage.writeString("Producer Message # ");
      aMessage.writeInt(i);
      //    System.out.println ("Producer>>> Sending message " +
      //i + " to Q " +theQSender);
      theQSender.send (aMessage);
      System.out.println("Sleep for 3 secs");
      sleep(3000); // sleep so that I can make sure that I get a Ack from
consumer
      System.out.println("Done Sleeping");
      }
      close();
    }
    catch (Exception e) {
      System.out.println("Exception in Producer.run()");
      e.printStackTrace();
    }
  }

  /**
   *  Close the queue session
   */

  public void close() {
    try {
      System.out.println("Closing Producer queue session");
      theSession.close();
      SESSION_STATUS = false;
    }
    catch(Exception e) {
```

```
      System.out.println("Exception, could not close Producer queue
session"+
                   e.getMessage());
      e.printStackTrace();
    }
  }

}
```

QConsumer.java

```
import javax.jms.*;
import java.util.*;

public class QConsumer extends Thread implements MessageListener{

  private QueueSession             theSession;
  private QueueReceiver       theQReceiver;

  public QConsumer(QueueSession aqs, Queue aq) {
    theSession = aqs;
    try {
      theQReceiver = theSession.createReceiver(aq);
    }
    catch (JMSException jmse) {
      System.out.println ("Error creating Consumer: " + jmse);
    }
  }

  public void run() {
    //  System.out.println("Consumer Q set to " + theQReceiver);
    try {
      theQReceiver.setMessageListener (this);
    }
    catch (JMSException jmse) {
      System.out.println ("Consumer.run(): " + jmse);
    }
  }

  public void onMessage(Message m) {

    try {
      StreamMessage aMessage = (StreamMessage) m;

      String s = aMessage.readString();
      int i = aMessage.readInt();

      System.out.println ("Consumer<<< Message # n" +
                   " Received message body: " + s + i);

      System.out.println("JMS Message Header Properties ... ");
      System.out.println("\t Message Destination   = " +
m.getJMSDestination());
      System.out.println("\t Message Delivery mode = " +
m.getJMSDeliveryMode());
      System.out.println("\t Message ID            = " +
m.getJMSMessageID());
      System.out.println("\t Message TimeStamp     = " +
m.getJMSTimestamp());
      System.out.println("\t Correlation ID        = " +
m.getJMSCorrelationID());
```

```
      System.out.println("\t Reply To this message @ " +
m.getJMSReplyTo());
      System.out.println("\t Message Redelivered ? = " +
m.getJMSRedelivered());
      System.out.println("\t Message Expiration    = " +
m.getJMSExpiration());
      System.out.println("\t Message Priority      = " +
m.getJMSPriority());
      System.out.println("\t Message Type          = " + m.getJMSType());
      System.out.print("\t Property Names : \n\t\t");
      for (Enumeration  e = m.getPropertyNames(); e.hasMoreElements();) {
      try {
      System.out.print(e.nextElement() + ", ");
      }
      catch (Exception ex) {
        ex.printStackTrace();
      }
      }

      System.out.println("\n\t Values of Properties are :");
      System.out.println("\t\t PT_NOTIFY    = " +
m.getObjectProperty("PT_NOTIFY"));
      System.out.println("\t\t PT_ACKNDX    = " +
m.getObjectProperty("PT_ACKNDX"));
      System.out.println("\t\t PT_DELIVERY  = " +
m.getObjectProperty("PT_DELIVERY"));
      System.out.println("\t\t PT_INDPATH   = " +
m.getObjectProperty("PT_INDPATH"));
      System.out.println("\t\t PT_SEQUENCE  = " +
m.getObjectProperty("PT_SEQUENCE"));
      System.out.println("\t\t PT_DATACLASS = " +
m.getObjectProperty("PT_DATACLASS"));
      System.out.println("\t\t PT_SRCDOMAIN = " +
m.getObjectProperty("PT_SRCDOMAIN"));
      System.out.println("\t\t PT_LIFETIME  = " +
m.getObjectProperty("PT_LIFETIME"));
      System.out.println("\t\t PT_TAG       = " +
m.getObjectProperty("PT_TAG"));

      System.out.println("\t\t JMSXUserID        = " +
m.getObjectProperty("JMSXUserID"));
      System.out.println("\t\t JMSXAppID         = " +
m.getObjectProperty("JMSXAppID"));
      System.out.println("\t\t JMSXDeliveryCount = " +
m.getObjectProperty("JMSXDeliveryCount"));
      System.out.println("\t\t JMSXGroupID       = " +
m.getObjectProperty("JMSXGroupID"));
      System.out.println("\t\t JMSXGroupSeq      = " +
m.getObjectProperty("JMSXGroupSeq"));
      System.out.println("\t\t JMSXProducerTXID  = " +
m.getObjectProperty("JMSXProducerTXID"));
      System.out.println("\t\t JMSXConsumerTXID   = " +
m.getObjectProperty("JMSXConsumerTXID"));
      System.out.println("\t\t JMSXRcvTimestamp  = " +
m.getObjectProperty("JMSXRcvTimestamp"));
      System.out.println("\t\t JMSXState         = " +
m.getObjectProperty("JMSXState"));


      m.acknowledge ();
      System.out.println("Consumer Message Ack sent");
```

```
    }
    catch (Exception jmse) {
      System.out.println ("Consumer: " + jmse);
    }
  }

  public void close() {
    try {
      System.out.println("Closing Consumer queue session");
      theSession.close();
    }
    catch(Exception e) {
      System.out.println("Exception, could not close Consumer queue
session" +
                   e.getMessage());
      e.printStackTrace();
    }
  }

}
```

1.11.5. Selectors

In diesem Beispiel werden Nachrichten selektiv weiter geleitet, mit Hilfe eines Selektors. Dieer wird als Property angegeben.

Ausführungsprotokoll

```
Creating topic connection
Creating topic session: not transacted, auto ack
Creating topic, publisher...
Creating and sending message...
Receiving message...
Topic selector test successful!
Topic TextMessage test successful!
Closing publisher and subscriber...
Closing topic session and topic connection
Socket read exception occurred: java.net.SocketException: JVM_recv in
socket input stream read (code=10004) IAReadChannel(SocketRead) cannot read
socket
Creating queue connection
Creating queue session: not transacted, auto ack
Creating queue, sender...
Queue Created
Queue Session Created
Creating and sending message...
Receiving message...
Queue selector test successful!
Queue TextMessage test successful!
Closing sender and receiver...
Closing queue session and queue connection
Socket read exception occurred: java.net.SocketException: JVM_recv in
socket input stream read (code=10004) IAReadChannel(SocketRead) cannot read
socket
```

Programmcode

```
/**
 * This example is designed to use property selectors to filter the
 * messages a program wants to receive.
 *
 * To do this using Topics or Queues
 * involves the following 10 steps with
 *    a small difference in creating the Subscriber:
 *          Start the router
 *          Create a *ConnectionFactory
 *          Create a *Connection
 *          Create a *Session
 *          Create a Topic (Queue) to Publish / Subscribe to
 *          Create a Publisher (Sender)
 *          Create a Subscriber (Receiver) with a selector
 *          Create a Message
 *          Publish (Send) the Message
 *          Subscribe (Receive) the Message
 *
 *
 */


import javax.jms.*;


public class Selectors {

  TopicConnectionFactory     topicConnectionFactory;
  TopicConnection            topicConnection;
  TopicSession               topicSession;
  TopicPublisher             topicPublisher;
  TopicSubscriber            topicSubscriber;
  Topic                            topic;

  QueueConnectionFactory     queueConnectionFactory;
  QueueConnection            queueConnection;
  QueueSession               queueSession;
  QueueSender                      queueSender;
  QueueReceiver              queueReceiver;
  Queue                            queue;

  String[]                   arguments;
  String                     testString1, selector;
  Thread                     mythread;


  /**
   * Default Constructor
   *
   */

  public Selectors() {
  }


  /**
   * Create the connection factories. Begin the example.
   *
   * @param args JMQ startup properties
```

```
   */

  public Selectors(String[] args) {
    topicConnectionFactory = new
com.sun.messaging.TopicConnectionFactory(args);
    queueConnectionFactory = new
com.sun.messaging.QueueConnectionFactory(args);
    selectors_example();
  }


  /**
   *  Create the program.
   *
   * @param args JMQ startup properties
   *
   */

  public static void main(String args[]) {
    Selectors s = new Selectors(args);
  }


  /**
   *  Sleep for a specified time.
   *
   *  @param time Time in milliseconds to wait.
   *
   */

  public void sleep(int time) {
    try {
      Thread.sleep(time);
    }
    catch(Exception e) {
    }
  }


  /**
   * Complete the topic example, followed by the queue example,
   * then exit the program.
   *
   */

  public void selectors_example() {
    //Create a TopicConnection, TopicSession
    t11();
    //Demonstrate Sending and Receiving Topic Messages using property
selectors
    tmessage();
    //Close the TopicConnection, TopicSession
    tc();

    //Create a QueueConnection, QueueSession
    q11();
    //Demonstrate Sending and Receiving Queue Messages using property
selectors
    qmessage();
    //Close the QueueConnection, QueueSession
    qc();
```

```
    System.exit(0);
  }


  /**
   * Create the TopicConnection and TopicSession
   *
   */

  public void t11() {
    try {
      topicConnection = topicConnectionFactory.createTopicConnection();
      topicConnection.start();
      System.out.println("Creating topic connection");
      //creating Topic Session
      //    Transaction Mode: None
      //    Acknowledge Mode: Automatic
      topicSession = topicConnection.createTopicSession(false,1);
      System.out.println("Creating topic session: not transacted, auto
ack");
    }
    catch(Exception e) {
      System.out.println("Exception, could not create topic connection or
session: " +
                  e.getMessage() + " ");
      e.printStackTrace();
    }
  }


  /**
   *  Demonstrates receiving a message using selectors with topics
   *
   */

  public void tmessage() {
    try {
      System.out.println("Creating topic, publisher...");
      // Create a topic, a publisher
      topic = topicSession.createTopic("SelectorsExampleJTopic");
      topicPublisher = topicSession.createPublisher(topic);

      //Set Selector, then create Receiver
      selector = "topicselectortests";
      topicSubscriber = topicSession.createSubscriber(topic,
                                        "myselector = '"+selector+"'",
                                        false);

      System.out.println("Creating and sending message...");
      // Use the Session to create a message
      TextMessage textmsg1 = topicSession.createTextMessage();
      // Set the message properties
      testString1 = "Topic Message Test";
      textmsg1.setText(testString1);
      textmsg1.setStringProperty("myselector", selector);
      // Send the message
      topicPublisher.publish(textmsg1);

      System.out.println("Receiving message...");
      // Manually receive the message
      TextMessage textmsg2 = (TextMessage)topicSubscriber.receive();
      // The selector should be equal to the one set to be received
```

```
      if(selector.equals(textmsg2.getStringProperty("myselector"))) {
      System.out.println("Topic selector test successful!");
      // Now test if the message is same as the one sent
      if(textmsg2.getText().equals(testString1)) {
        System.out.println("Topic TextMessage test successful!");
      }
      else {
        System.out.println("Topic TextMessage test failed!");
      }
      }
      else {
      System.out.println("Topic selector test failed!");
      }

      // Close the publisher, the subscriber
      System.out.println("Closing publisher and subscriber...");
      topicPublisher.close();
      topicSubscriber.close();
    }
    catch(JMSException e) {
      System.out.println("JMSException in tmessage(): " + e + "");
      e.printStackTrace();
    }
  }


  /**
   *  Create the QueueConnection and QueueSession
   *
   */

  public void q11() {
    try {
      queueConnection = queueConnectionFactory.createQueueConnection();
      queueConnection.start();
      System.out.println("Creating queue connection");
      //creating Queue Session
      //    Transaction Mode: None
      //    Acknowledge Mode: Automatic
      queueSession = queueConnection.createQueueSession(false,1);
      System.out.println("Creating queue session: not transacted, auto
ack");
    }
    catch(Exception e) {
      System.out.println("Exception, could not create queue connection or
session: " +
                  e.getMessage() + " ");
      e.printStackTrace();
    }
  }


  /**
   *  Demonstrates receiving a message using selectors with queues
   *
   */

  public void qmessage() {
    try {
      System.out.println("Creating queue, sender...");
      // Create a queue, a sender
      queue = queueSession.createQueue("SelectorsExampleJQueue");
```

```
      System.out.println("Queue Created");
      queueSender = queueSession.createSender(queue);
      System.out.println("Queue Session Created");
      //Set Selector, then create Receiver
      selector = "queueselectortests";
      queueReceiver = queueSession.createReceiver(queue,
                                    "myselector = '"+selector+"'");

      System.out.println("Creating and sending message...");
      // Use the Session to create a message
      TextMessage textmsg1 = queueSession.createTextMessage();
      // Set the message properties
      testString1 = "Queue Message Test";
      textmsg1.setText(testString1);
      textmsg1.setStringProperty("myselector", selector);
      // Send the message
      queueSender.send(textmsg1);

      System.out.println("Receiving message...");
      // Manually receive the message
      TextMessage textmsg2 = (TextMessage)queueReceiver.receive();
      // The selector should be equal to the one set to be received
      if(selector.equals(textmsg2.getStringProperty("myselector"))) {
      System.out.println("Queue selector test successful!");
      // Now test if the message is same as the one sent
      if(textmsg2.getText().equals(testString1)) {
        System.out.println("Queue TextMessage test successful!");
      }
      else {
        System.out.println("Queue TextMessage test failed!");
      }
      }
      else {
      System.out.println("Queue selector test failed!");
      }

      // Close the sender, the receiver
      System.out.println("Closing sender and receiver...");
      queueSender.close();
      queueReceiver.close();
    }
    catch(JMSException e) {
      System.out.println("JMSException in qmessage(): " + e + "");
      e.printStackTrace();
    }
  }


  /**
   *  Close the topic session and topic connection
   *
   */

  public void tc() {
    try {
      System.out.println("Closing topic session and topic connection");
      topicSession.close();
      topicConnection.close();
    }
    catch(Exception e) {
      System.out.println("Exception, could not close topic session or topic
connection: "+
```

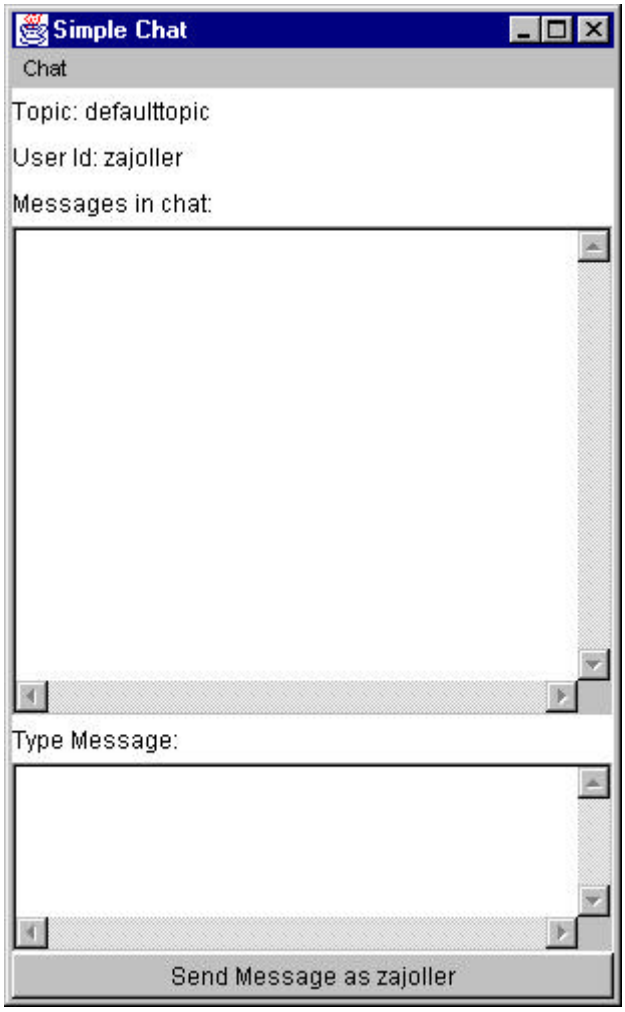
```
                    e.getMessage() + " ");
      e.printStackTrace();
    }
  }


  /**
   *  Close the queue session and queue connection
   *
   */

  public void qc() {
    try {
      System.out.println("Closing queue session and queue connection");
      queueSession.close();
      queueConnection.close();
    }
    catch(Exception e) {
      System.out.println("Exception, could not close queue session or queue
connection: "+
                    e.getMessage() + " ");
      e.printStackTrace();
    }
  }
}
```

1.11.6. Simplechat

Dieses Beispiel implementiert ein einfaches Chat System. Testen Sie es einfach aus. Es funktioniert! Die Oberfläche ist so banal, dass jeder damit zu Rande kommen sollte.

Die Chat Applikation ist ein Topic Publisher und Subscriber. mit Hilfe der onMessage(Message) Method des javax.jms.MessageListener Interface werden die Meldungen jeweils übermittelt.

Der grösste Teil des Programmes befasst sich mit der grafischen Oberfläche. Der eigentliche Nachrichtenteil ist eher banal.



1.11.7. Transactions

Das Beispiel zeigt eine verteilte Transaktionsverwaltung mit Hilfe von JMS.

Eine Transaktion bündelt das Senden und Empfangen mehrerer Nachrichten zu einer atomaren Einheit.

Mit Hilfe des commit Befehls können die Transaktionen bestätigt werden; mit Hilfe von Rollbacks werden sie rückgängig gemacht.

Ausführungsprotokoll

```
Creating topic connection
Creating topic session: transacted, auto ack
Creating topic, publisher, subscriber, messagelistener...
Creating and Sending 2 transacted messages...
Committing transaction...
MessageListener: message received
MessageListener: message received
Demonstrating Transaction-Rollback...
Creating and Sending 2 transacted messages...
Rollback the transaction...
MessageListener: message received
MessageListener: message received
Commit the transaction...
Creating and Sending 2 transacted messages...
Committing transaction...
MessageListener: message received
MessageListener: message received
Closing publisher and subscriber...
Closing topic session and topic connection
Socket read exception occurred: java.io.EOFException
IAReadChannel(SocketRead) cannot read socket
Creating queue connection
Creating queue session: transacted, auto ack
Creating queue, sender, receiver, messagelistener...
Creating and Sending 2 transacted messages...
Committing transaction...
MessageListener: message received
MessageListener: message received
Demonstrating Transaction-Rollback...
Creating and Sending 2 transacted messages...
Rollback the transaction...
MessageListener: message received
MessageListener: message received
Commit the transaction...
Creating and Sending 2 transacted messages...
Committing transaction...
MessageListener: message received
MessageListener: message received
Closing sender and receiver...
Closing queue session and queue connection
Socket read exception occurred: java.io.EOFException
IAReadChannel(SocketRead) cannot read socket
```

Programmcode

```
/**
 * @(#)Transactions.java     1.3 99/10/21
*/


/**
 *  This example is designed to demonstrate how to use transactions.
 *
 * A transaction represents a set of messages that are to be sent and
 * received as a group.
 *
 * To do this using Topics or Queues
 * involves the following steps
 *          Start the router
 *          Create a *ConnectionFactory
 *          Create a *Connection
 *          Create a Transacted *Session
 *          Create a Topic (Queue) to Publish / Subscribe to
 *          Create a Publisher (Sender)
 *          Create a Subscriber (Receiver)
 *          Create a MessageListener to receive the messages asynchronously
 *          Create the Message(s)
 *          Publish (Send) the Message(s)
 *          Commit the Transaction (Send all the messages)
 *
 * In addition, we will demonstrate how to execute a 'rollback' to reset a
transaction.
 *
 * Also, the MessageListener should receive all the messages on it's own
without any
 * manual calls to receive()
 *
*/


import javax.jms.*;


public class Transactions {

  TopicConnectionFactory     topicConnectionFactory;
  TopicConnection       topicConnection;
  TopicSession               topicSession;
  TopicPublisher        topicPublisher;
  TopicSubscriber       topicSubscriber;
  Topic                      topic;

  QueueConnectionFactory     queueConnectionFactory;
  QueueConnection            queueConnection;
  QueueSession               queueSession;
  QueueSender                      queueSender;
  QueueReceiver              queueReceiver;
  Queue                            queue;

  messagelistener            transactedlistener;

  String[]                   arguments;
  String                     testString1, selector;
```

```
  Thread                mythread;


  /**
   *Default Constructor
   */

  public Transactions() {
  }


  /**
   * Create the connection factories. Begin the example.
   *
   * @param args JMQ startup properties
   */

  public Transactions(String[] args) {
    topicConnectionFactory = new
com.sun.messaging.TopicConnectionFactory(args);
    queueConnectionFactory = new
com.sun.messaging.QueueConnectionFactory(args);
    selectors_example();
  }


  /**
   * Create the program.
   *
   * @param args JMQ startup properties
   */

  public static void main(String args[]) {
    Transactions t = new Transactions(args);
  }


  /**
   * Sleep for a specified time.
   *
   * @param time Time in milliseconds to wait.
   */

  public void sleep(int time) {
    try {
      Thread.sleep(time);
    }
    catch(Exception e) {
    }
  }


  /**
   * Complete the topic example, follwed by the queue example, then exit
the program.
   */

  public void selectors_example() {
    //Create a TopicConnection, TopicSession
    t11();
    //Topic Transactions
    tmessage();
```

```
    //Close the TopicConnection, TopicSession
    tc();

    //Create a QueueConnection, QueueSession
    q11();
    //Queue Transactions
    qmessage();
    //Close the QueueConnection, QueueSession
    qc();

    System.exit(0);
  }


  /**
   * Create the TopicConnection and TopicSession
   */

  public void t11() {
    try {
      topicConnection = topicConnectionFactory.createTopicConnection();
      topicConnection.start();
      System.out.println("Creating topic connection");
      //creating Topic Session
      //    Transaction Mode: True
      //    Acknowledge Mode: Automatic
      topicSession = topicConnection.createTopicSession(true,1);
      System.out.println("Creating topic session: transacted, auto ack");
    }
    catch(Exception e) {
      System.out.println("Exception, could not create topic connection or
session: " +
                   e.getMessage() + " ");
      e.printStackTrace();
    }
  }


  /**
   * Demonstrates Topic Transactions
   */

  public void tmessage() {
    try {
      System.out.println("Creating topic, publisher, subscriber,
messagelistener...");
      // Create a topic, a publisher
      topic = topicSession.createTopic("TransactionsExampleJTopic");
      topicPublisher = topicSession.createPublisher(topic);
      topicSubscriber = topicSession.createSubscriber(topic);
      transactedlistener = new messagelistener();
      topicSubscriber.setMessageListener(transactedlistener);

      System.out.println("Creating and Sending 2 transacted messages...");
      // Use the Session to create a message
      TextMessage textmsg1 = topicSession.createTextMessage();
      TextMessage textmsg2 = topicSession.createTextMessage();
      // Set the message properties
      testString1 = "Transacted Topic Message Test 1";
      textmsg1.setText(testString1);
      testString1 = "Transacted Topic Message Test 2";
      textmsg2.setText(testString1);
```

```
      // Send the messages
      topicPublisher.publish(textmsg1);
      topicPublisher.publish(textmsg2);

      System.out.println("Committing transaction...");
      // Commit the Transaction -
      //    The messagelistener should now receive
      //    messages 1 and 2 as a group.
      topicSession.commit();

      //-----------

      System.out.println("Demonstrating Transaction-Rollback...");
      System.out.println("Creating and Sending 2 transacted messages...");
      // Use the Session to create a message
      TextMessage textmsg3 = topicSession.createTextMessage();
      TextMessage textmsg4 = topicSession.createTextMessage();
      // Set the message properties
      testString1 = "Transacted Topic Message Test 3";
      textmsg3.setText(testString1);
      testString1 = "Transacted Topic Message Test 4";
      textmsg4.setText(testString1);
      // Send the messages
      topicPublisher.publish(textmsg3);
      topicPublisher.publish(textmsg4);

      System.out.println("Rollback the transaction...");
      // Rollback the Transaction -
      //    'Resets' the transaction by removing
      //    all the messages previously sent.
      topicSession.rollback();
      System.out.println("Commit the transaction...");
      // Commit the Transaction -
      //    The messagelistener should NOT
      //    receive any messages because of the rollback.
      topicSession.commit();

      //-----------

      // Demonstrate that previous rollback no longer affects this
transaction
      System.out.println("Creating and Sending 2 transacted messages...");
      // Use the Session to create a message
      TextMessage textmsg5 = topicSession.createTextMessage();
      TextMessage textmsg6 = topicSession.createTextMessage();
      // Set the message properties
      testString1 = "Transacted Topic Message Test 5";
      textmsg5.setText(testString1);
      testString1 = "Transacted Topic Message Test 6";
      textmsg6.setText(testString1);
      // Send the messages
      topicPublisher.publish(textmsg5);
      topicPublisher.publish(textmsg6);

      System.out.println("Committing transaction...");
      // Commit the Transaction -
      //    The messagelistener should now receive
      //    messages 5 and 6 as a group.
      topicSession.commit();

      // Close the publisher, the subscriber
      System.out.println("Closing publisher and subscriber...");
```

```
      topicPublisher.close();
      topicSubscriber.close();
    }
    catch(JMSException e) {
      System.out.println("JMSException in tmessage(): " + e + "");
      e.printStackTrace();
    }
  }


  /**
   * Create the QueueConnection and QueueSession
   */

  public void q11() {
    try {
      queueConnection = queueConnectionFactory.createQueueConnection();
      queueConnection.start();
      System.out.println("Creating queue connection");
      //creating Queue Session
      //    Transaction Mode: True
      //    Acknowledge Mode: Automatic
      queueSession = queueConnection.createQueueSession(true,1);
      System.out.println("Creating queue session: transacted, auto ack");
    }
    catch(Exception e) {
      System.out.println("Exception, could not create queue connection or
session: " +
                   e.getMessage() + " ");
      e.printStackTrace();
    }
  }


  /**
   *  Demonstrates Queue Transactions
   */

  public void qmessage() {
    try {
      System.out.println("Creating queue, sender, receiver,
messagelistener...");
      // Create a queue, a Sender
      queue = queueSession.createQueue("TransactionsExampleJqueue");
      queueSender = queueSession.createSender(queue);
      queueReceiver = queueSession.createReceiver(queue);
      transactedlistener = new messagelistener();
      queueReceiver.setMessageListener(transactedlistener);

      System.out.println("Creating and Sending 2 transacted messages...");
      // Use the Session to create a message
      TextMessage textmsg1 = queueSession.createTextMessage();
      TextMessage textmsg2 = queueSession.createTextMessage();
      // Set the message properties
      testString1 = "Transacted queue Message Test 1";
      textmsg1.setText(testString1);
      testString1 = "Transacted queue Message Test 2";
      textmsg2.setText(testString1);
      // Send the messages
      queueSender.send(textmsg1);
      queueSender.send(textmsg2);
```

```
      System.out.println("Committing transaction...");
      // Commit the Transaction -
      //    The messagelistener should now receive
      //    messages 1 and 2 as a group.
      queueSession.commit();

      //-----------

      System.out.println("Demonstrating Transaction-Rollback...");
      System.out.println("Creating and Sending 2 transacted messages...");
      // Use the Session to create a message
      TextMessage textmsg3 = queueSession.createTextMessage();
      TextMessage textmsg4 = queueSession.createTextMessage();
      // Set the message properties
      testString1 = "Transacted queue Message Test 3";
      textmsg3.setText(testString1);
      testString1 = "Transacted queue Message Test 4";
      textmsg4.setText(testString1);
      // Send the messages
      queueSender.send(textmsg3);
      queueSender.send(textmsg4);

      System.out.println("Rollback the transaction...");
      // Rollback the Transaction -
      //    'Resets' the transaction by removing
      //    all the messages previously sent.
      queueSession.rollback();
      System.out.println("Commit the transaction...");
      // Commit the Transaction -
      //    The messagelistener should NOT
      //    receive any messages because of the rollback.
      queueSession.commit();

      //-----------

      // Demonstrate that previous rollback no longer affects this
transaction
      System.out.println("Creating and Sending 2 transacted messages...");
      // Use the Session to create a message
      TextMessage textmsg5 = queueSession.createTextMessage();
      TextMessage textmsg6 = queueSession.createTextMessage();
      // Set the message properties
      testString1 = "Transacted queue Message Test 5";
      textmsg5.setText(testString1);
      testString1 = "Transacted queue Message Test 6";
      textmsg6.setText(testString1);
      // Send the messages
      queueSender.send(textmsg5);
      queueSender.send(textmsg6);

      System.out.println("Committing transaction...");
      // Commit the Transaction -
      //    The messagelistener should now receive
      //    messages 5 and 6 as a group.
      queueSession.commit();

      // Close the sender, the receiver
      System.out.println("Closing sender and receiver...");
      queueSender.close();
      queueReceiver.close();
    }
    catch(JMSException e) {
```

```
      System.out.println("JMSException in qmessage(): " + e + "");
      e.printStackTrace();
    }
  }


  /**
   * Close the topic session and topic connection
   */

  public void tc() {
    try {
      System.out.println("Closing topic session and topic connection");
      topicSession.close();
      topicConnection.close();
    }
    catch(Exception e) {
      System.out.println("Exception, could not close topic session or topic
connection: "+
                  e.getMessage() + " ");
      e.printStackTrace();
    }
  }


  /**
   * Close the queue session and queue connection
   */

  public void qc() {
    try {
      System.out.println("Closing queue session and queue connection");
      queueSession.close();
      queueConnection.close();
    }
    catch(Exception e) {
      System.out.println("Exception, could not close queue session or queue
connection: "+
                  e.getMessage() + " ");
      e.printStackTrace();
    }
  }

  /**
   * Message Listener
   */

  public class messagelistener implements MessageListener {
    public void onMessage(Message message){
      System.out.println("MessageListener: message received");
    }
  }
}
```