JMS

JAVA MESSAGE SERVICE

INTRODUCTION TO JMS, JAVA'S MESSAGE SERVICE API

Peter R. Egli
INDIGOO.COM

Contents

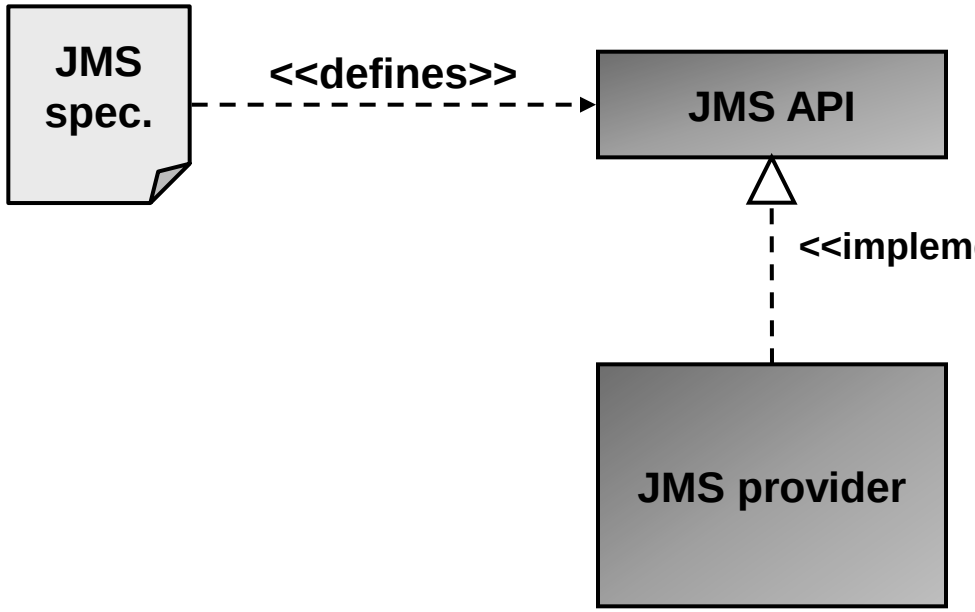
1. What is JMS?

JMS is an API for asynchronous message exchange between Java applications.

JMS implementations (instances that implement the JMS API) are called JMS provider.

Different JMS providers are not directly interoperable due to the lack of a defined wire protocol in JMS. JMS vendors usually provide bridges to connect to other JMS providers.

JMS is standardized as JSR-914 (Java Specification Request).



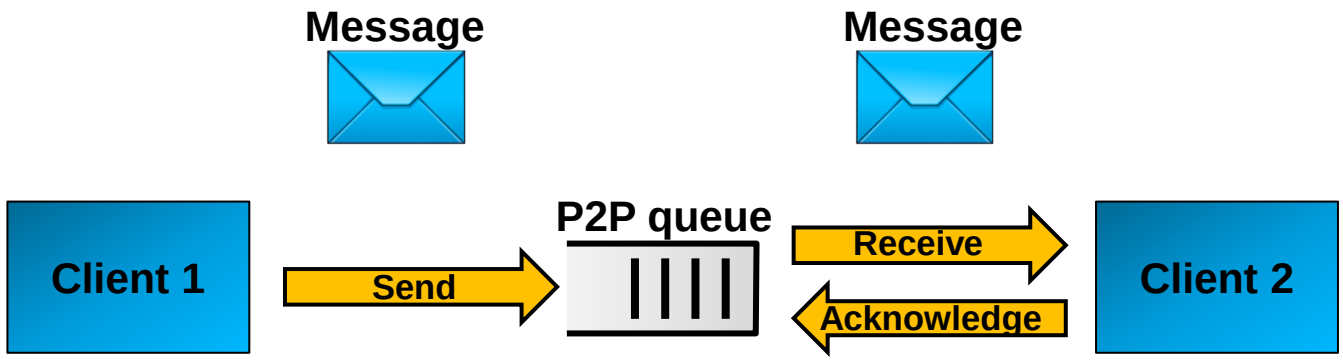
Example JMS providers (implementations):
*** IBM WebSphere MQ**
*** Tibco EMS™ (Enterprise Message Service)**
*** Progress Software SonicMQ**
*** Oracle AQ (Advanced Queueing)**
*** OpenMQ (part of Glassfish ref. impl.)**
*** ActiveMQ™ (Apache project)**
*** JBossMQ**

2. JMS messaging domains (1/2)

JMS supports the 2 messaging modes P2P (point-to-point) and PubSub (publish-subscribe). These message modes are also called messaging domain in JMS-speak.

P2P messaging domain:

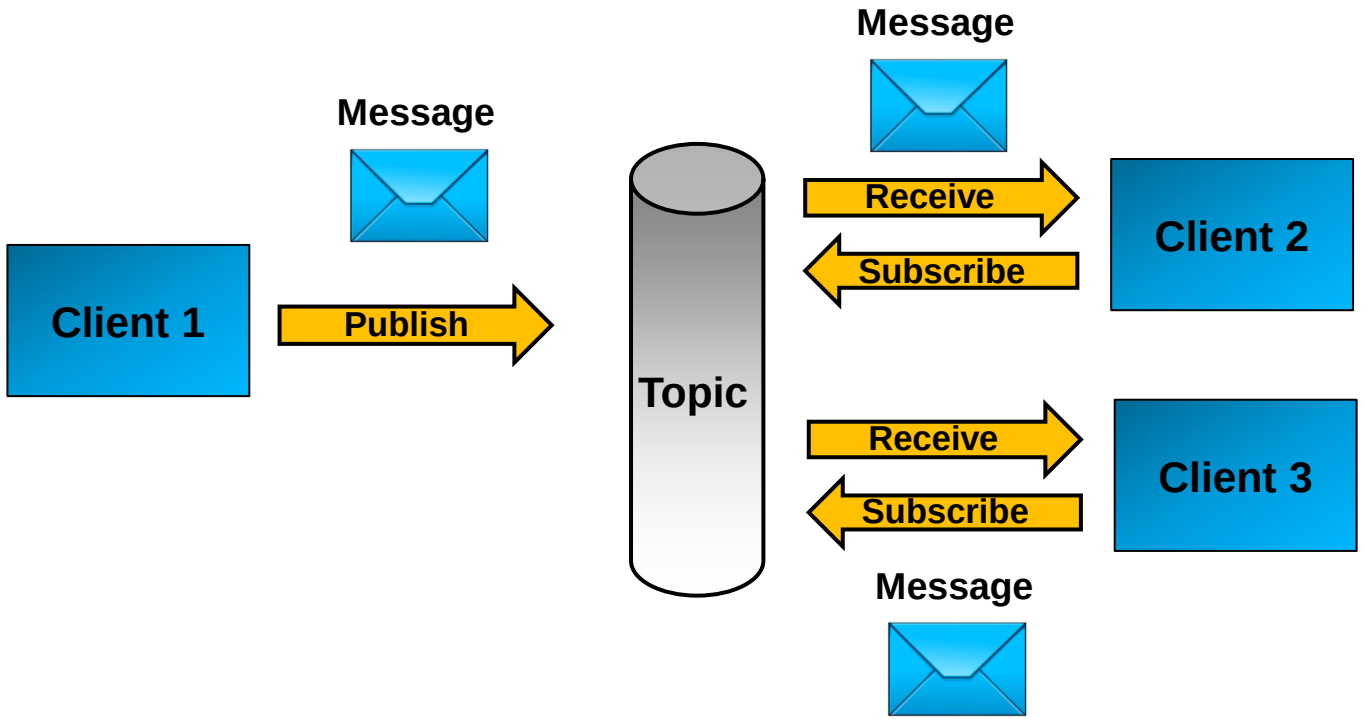
- **Each message has only 1 consumer.**
- **There is no timing relation between sender and receiver (the sender, client 1, may send before the receiver, client 2, is started).**



3. JMS messaging domains (2/2)

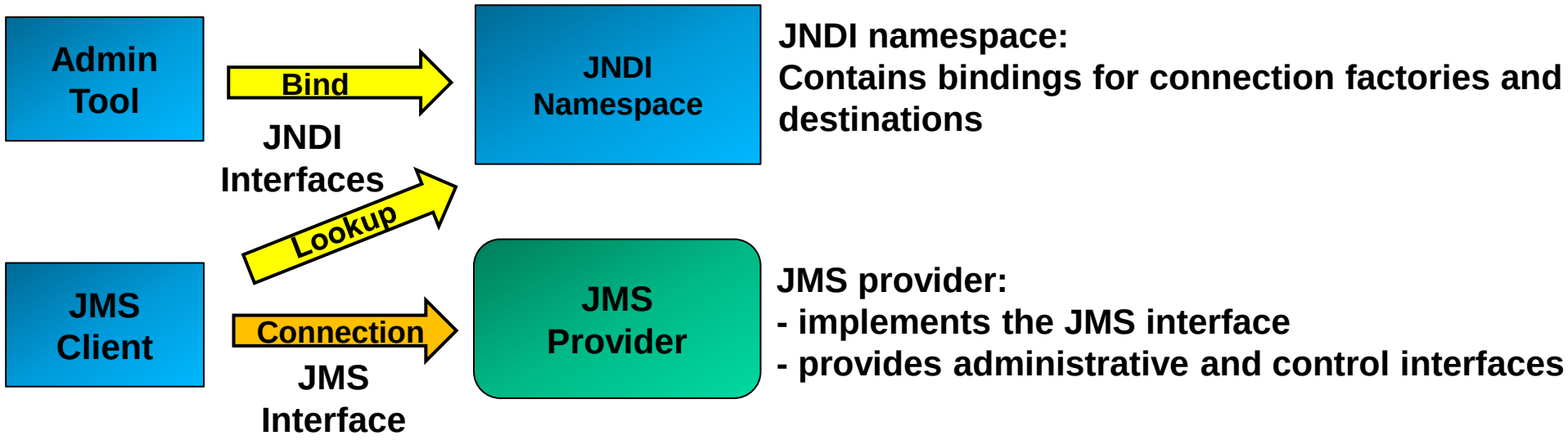
PubSub messaging domain (topic):

- PubSub queues are called topic.
- Each message may have multiple consumers / receivers.
- There is a weak timing relation: subscribers (client 2 and 3 in the figure below) will only receive messages received by the topic after the subscription.
- Option *Durable subscription*: consumer can register a permanent subscription that survives a reboot.



4. JMS architecture

JMS is an interface specification. JMS providers implement the messaging services (basically implement a queue or topic).
JMS makes heavy use of JNDI for lookup / discovery.
Queues, topics and connection factories are administratively created through an administrative tool.

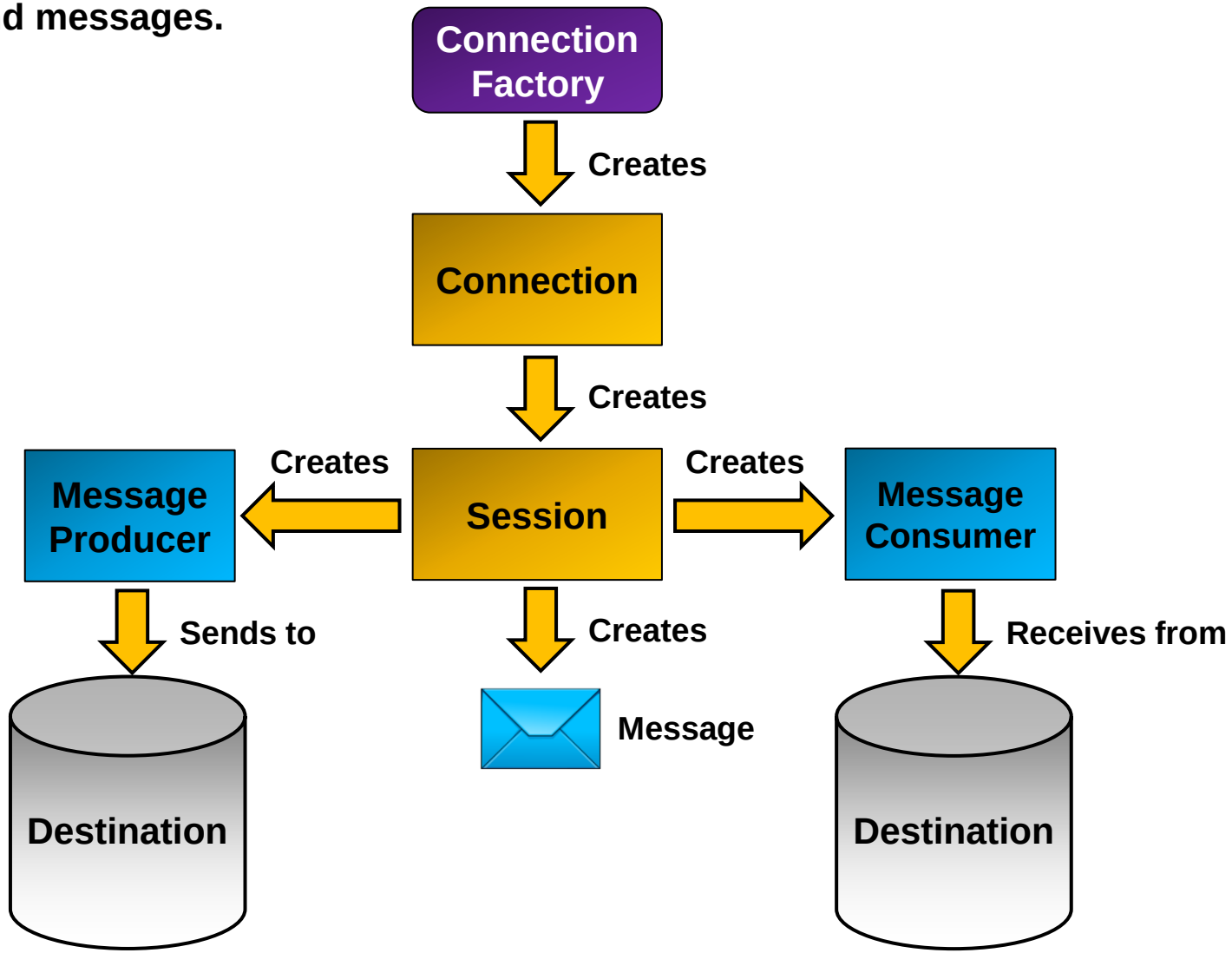


JNDI namespace:
Contains bindings for connection factories and destinations

JMS provider:
- implements the JMS interface
- provides administrative and control interfaces

5. JMS programming model (1/5)

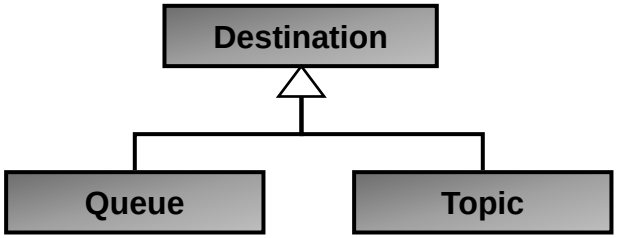
The building blocks of JMS are factories, connections, sessions, message producers and consumers and messages.



5. JMS programming model (2/5)

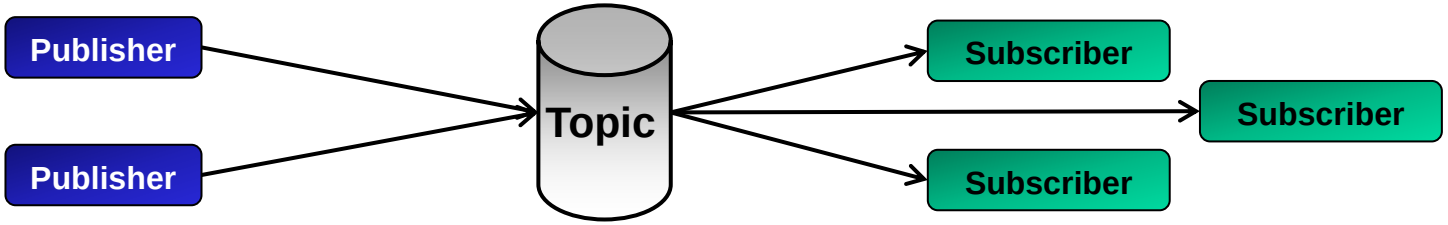
Destination:

A JMS destination is a queue to send to / receive from.
Destinations may be P2P queues or topics (pubsub model).



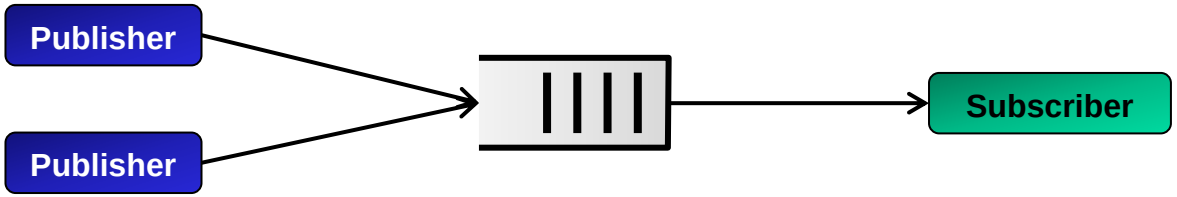
Topic:

A topic is a PubSub (publish / subscribe) queue (mailbox).
Multiple senders may send to a topic and multiple subscribers may receive the message.



Queue:

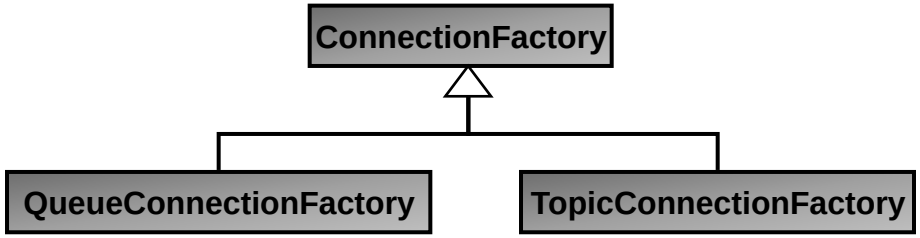
A queue is a P2P-queue (Point To Point) where multiple senders send to the queue but only 1 receiver receives the messages. Once a message is received, it is removed from the queue.



5. JMS programming model (3/5)

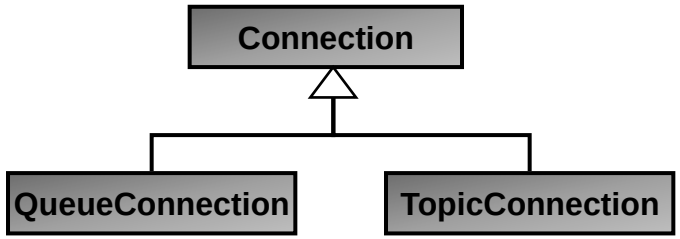
Connection factory:

Connection factories are used for creating queue and topic connections.



Queue connection:

A queue connection is a connection from JMS a client to a JMS provider.
Connections are created through connection factories.
A connection is either a QueueConnection (P2P model) or a TopicConnection (PubSub model).



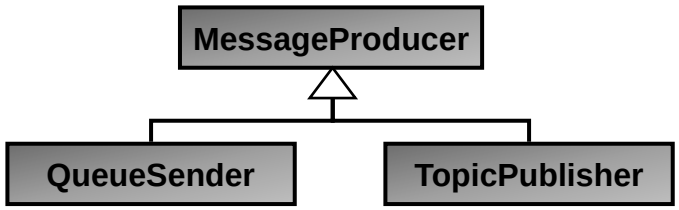
5. JMS programming model (4/5)

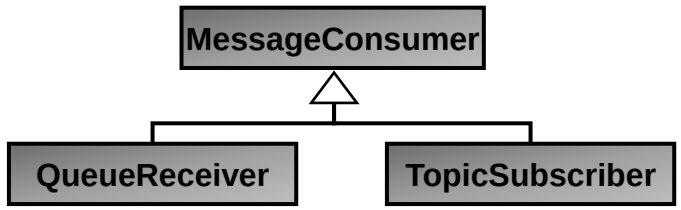
Session:

A session is a context to deliver and consume messages (defined lifecycle with a start and stop).
Sessions are created from connections (connection is a factory).

Message producer / message consumer:

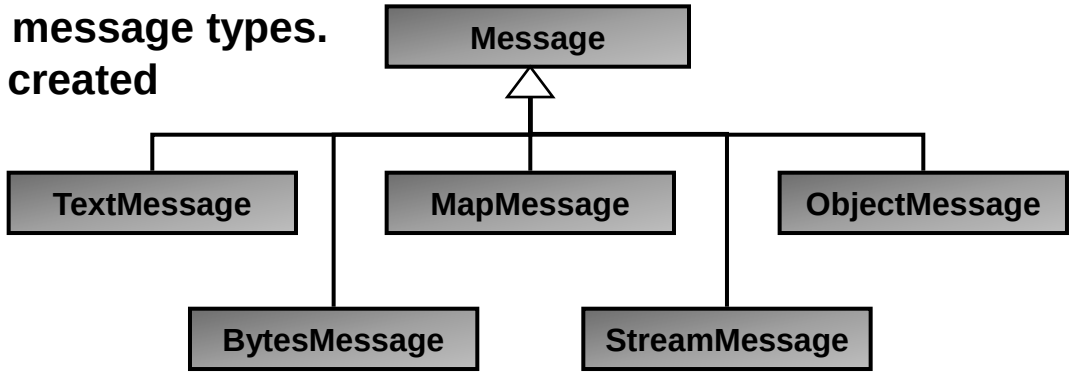
Message producer and consumer are objects for sending / publishing and receiving messages.
Message producer / consumer are created from sessions.





JMS message types:

JMS defines a couple of standard message types.
Application specific types can be created through subclassing.



5. JMS programming model (5/5)

JMS provider:

A JMS provider implements the JMS specification and provides a queue and a queue manager that receives and forwards messages.

JMS client:

A JMS client is either a producer (sender) or consumer (reciever) of a message.

JNDI provider:

JMS is tightly coupled to JNDI, the Java Naming and Directory Interface, for queue name lookups.
A JNDI provider is an instance that implements the JNDI interface specification and services name lookups (returns answers to name lookup requests).

JNDI initial context:

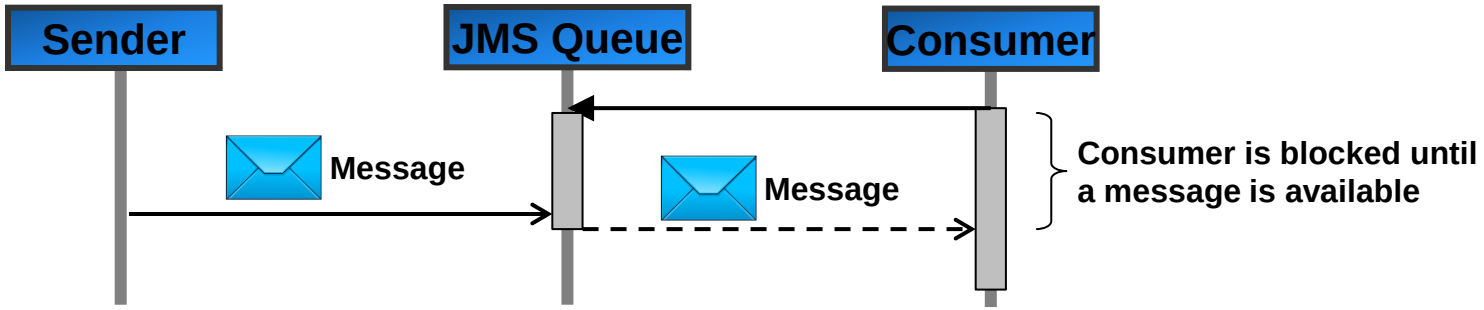
A JNDI initial context is the starting point for name lookups.

6. JMS receive modes

JMS supports both synchronous and asynchronous receive.

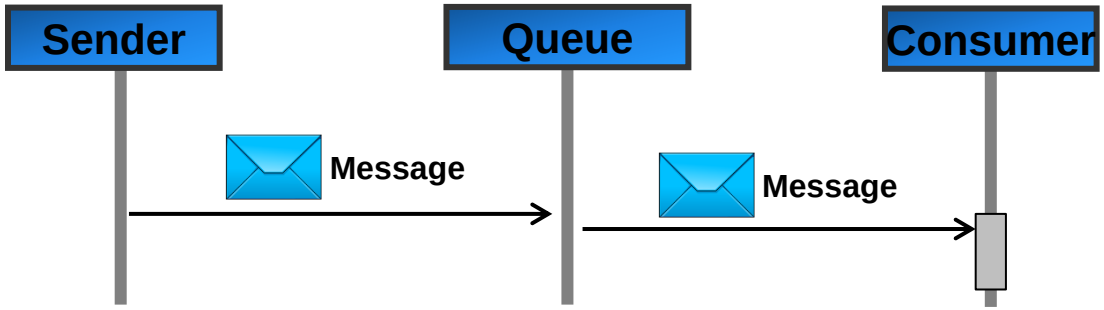
Synchronous receive:

The consumer calls the receive() operation. The receive() operation is blocking. If there is no message in the queue or topic, then the consumer is blocked.



Asynchronous receive:

The consumer registers a message listener. Upon reception of a message, the destination (message queue or topic) calls the message listener.



6. JMS robustness features (1/7)

JMS allows to implement reliable messaging between applications by providing the following robustness features:

a. Message acknowledgment
b. Persistent delivery mode
c. Message priorities
d. Control of message expiration
e. Durable subscriptions
f. Message transactions

N.B.: These features should not be turned on or used by default as they consume additional processing power. The pros and cons of these features need to be carefully weighed against each other.

6. JMS robustness features (2/7)

a. Controlling message acknowledgment:

In non-transacted sessions, a message is processed in 3 steps:

1. **Client receives the message.**
2. **Client processes the message.**
3. **Message is acknowledged using 1 of 3 message acknowledge modes.**

a. Auto-acknowledgment (`Session.AUTO_ACKNOWLEDGE`)

→ The message is acknowledged when the client has successfully returned from a call to receive() or when a message listener has successfully returned.

b. Client acknowledgment (`Session.CLIENT_ACKNOWLEDGE`)

→ The client explicitly acknowledges a message after it has successfully processed it by calling the message's acknowledge() method.

c. Lazy client acknowledgment (`Session.DUPS_OK_ACKNOWLEDGE`)

→ The session lazily acknowledges the delivery of a message. This mode is similar to auto-acknowledgment, but reduces the overhead on the JMS provider in that it does not need to prevent duplicate messages. Clients must be prepared to receive message duplicates in case the JMS provider fails.

Transacted sessions are auto-acknowledgment, i.e. messages are acknowledged upon completion (commit) of the transaction.

6. JMS robustness features (3/7)

b. PERSISTENT delivery mode:

When using persistent messages, JMS stores the message in persistent storage until the message is successfully delivered.

- **Failsafe operation (message can not be lost and survive crashes)**
- **Needs more performance**
- **Needs more storage (for the messages)**

Message persistence is not always the best solution. It introduces additional processing overhead and storage requirements for every message.
Usually message loss only occurs in exceptional circumstances (JMS provider crashes).
Depending on the application it may make sense to handle failures in the application and run JMS in NON_PERSISTENT mode (faster, less processing overhead, optimization of the normal case, handle exceptions in special application code).

c. Message priorities:

Message priorities can be used to have urgent messages delivered first. There are 10 priorities to choose from:

0 = lowest
4 = default
9 = highest

6. JMS robustness features (4/7)

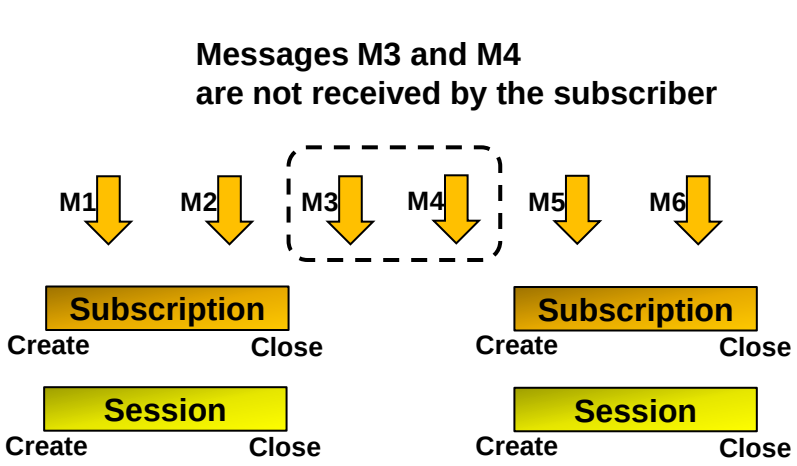
d. Message expiration:

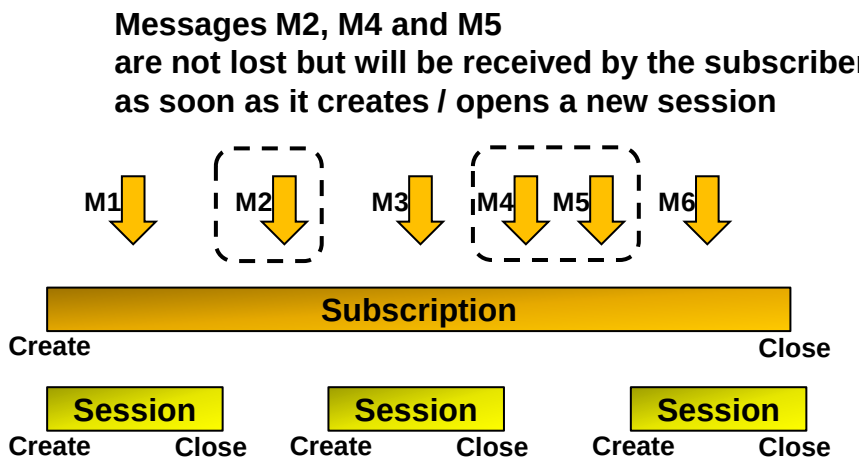
By default messages never expire.
When using message priorities, there is the danger that messages are never delivered and queues / topics grow beyond all bounds, thus consuming storage space.

Thus JMS allows to set a lifetime for messages (TTL, Time To Live).
Default lifetime is 0 (= never expires).

e. Durable subscriptions:

Receivers for messages from topics must subscribe to the topic before receiving messages.
The subscriptions are non-persistent, i.e. after each reboot the receiver must subscribe again.
Durable subscriptions allow a receiver to permanently subscribe (= persistent subscription).



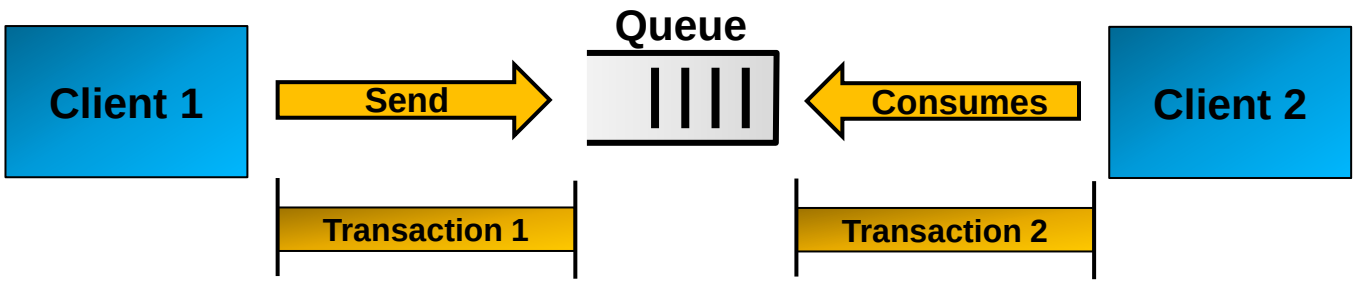


6. JMS robustness features (5/7)

f. Transactions (1/3):

Transactions allow grouping a series of operations together into an atomic unit of work.
If one of the operations in the transaction fails, it can be rolled back (effects of all operations in the transaction are undone).
All produced messages are destroyed and all consumed messages are recovered (unless they have expired).
If all operations are successful, a transaction commit means that all produced messages are sent and all consumed messages are acknowledged.

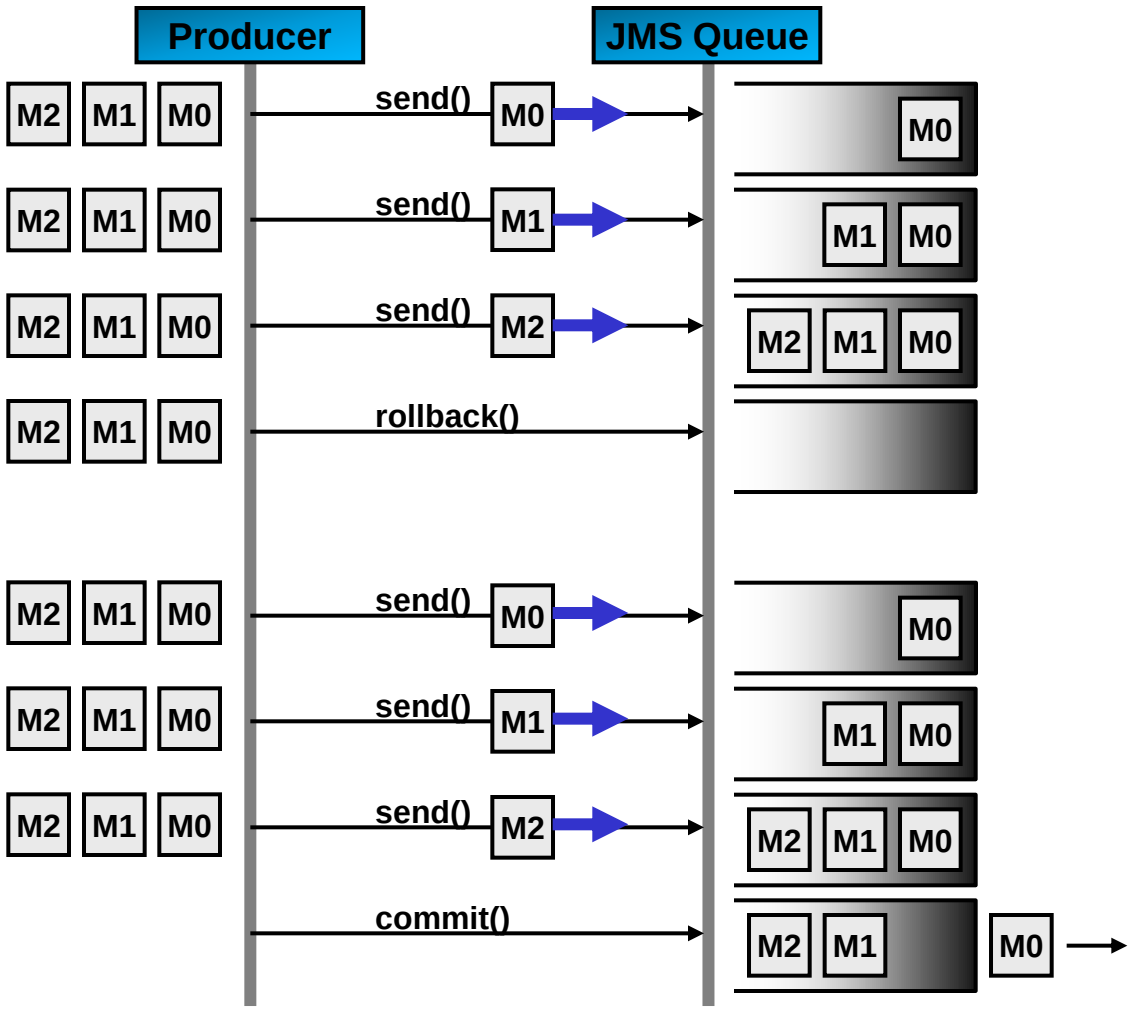
N.B.: Transactions can not be combined with a request-reply mechanism where the production of the next message depends on the successful reception of a reply. A message in a transaction is only sent upon the call of the commit() method, therefore a transaction may only contain message send or receive operations.



6. JMS robustness features (6/7)

f. Transactions (2/3):

A. Transmit commit() and rollback():



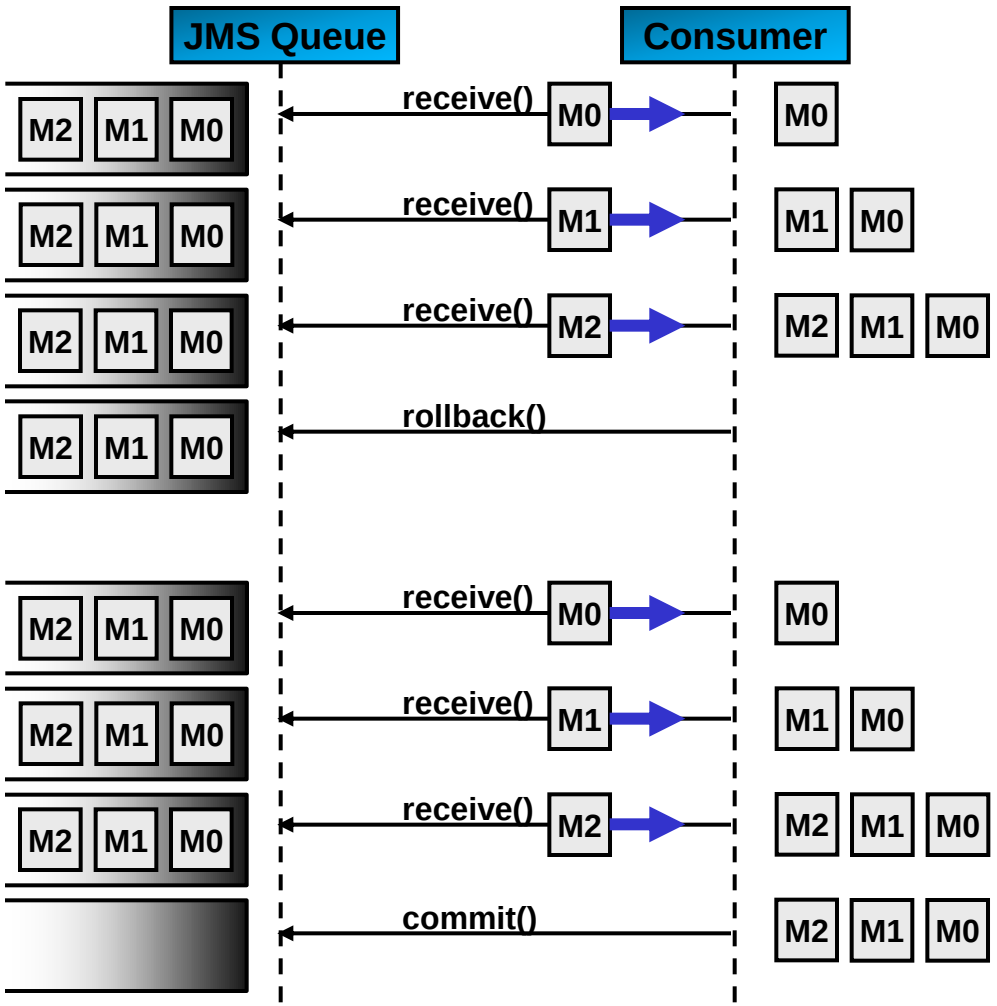
Rollback clears all unsent (uncommitted) messages of the transaction in the JMS transmit queue.

Commit causes that JMS queue sends all unsent messages of the transaction.

6. JMS robustness features (7/7)

f. Transactions (3/3):

A. Receive commit() and rollback():



Rollback instructs the JMS queue to ignore all delivered messages (message delivery restarts at message M0 again).

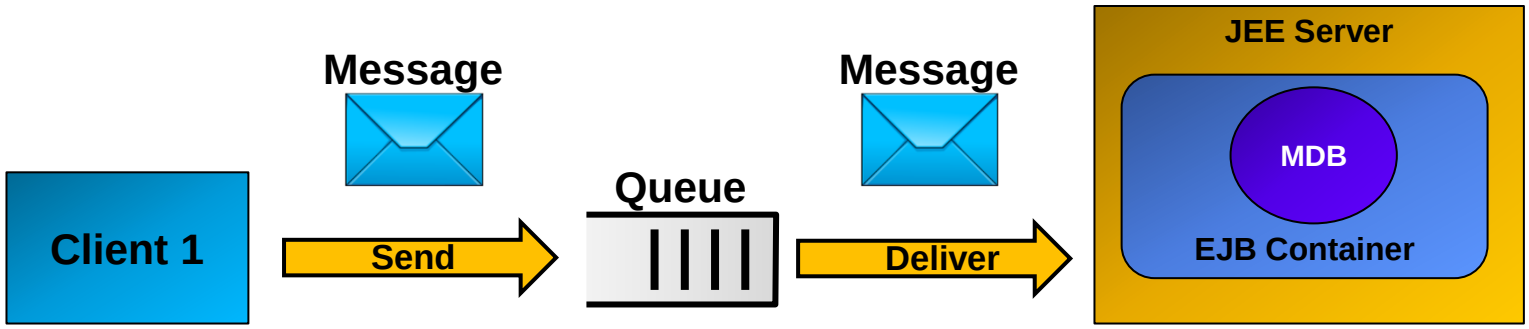
Commit tells the JMS queue that the consumer has successfully received all messages. The JMS queue clears these messages from the receive queue.

7. Integrating JMS with EJB

JMS can be nicely combined with and integrated into EJB for fully asynchronous interaction with EJBs.

EJB provides the bean type Message Driven Bean (MDB) that is able to receive and process messages from a JMS queue.

The MDB may itself process the message or pass it on to other EJBs through simple method calls.



8. JNDI (Java Naming and Directory Interface) (1/3)

Like RMI, JMS makes use of the central Java naming service JNDI.
The naming service provides 2 elementary services:

1. **Associate names with objects (create a binding)**
2. **Find / locate objects based on a name (lookup)**

Some important JNDI terms:

Association:
An association of a name to an object is called a binding.
Example: Filename (=name) to file (=object) binding.

Context:
A context is a set of name to object bindings.
Example: A file system directory defines a set of filename to file bindings.

Naming System:
A naming system is a connected set of contexts of the same type (same naming convention).
Examples: DNS (Domain Name System) or LDAP (Lightweight Directory Access Protocol) systems.

Namespace:
Namespace = set of names in a naming system.
Example: All names in a DNS system.

8. JNDI (Java Naming and Directory Interface) (2/3)

JNDI architecture:

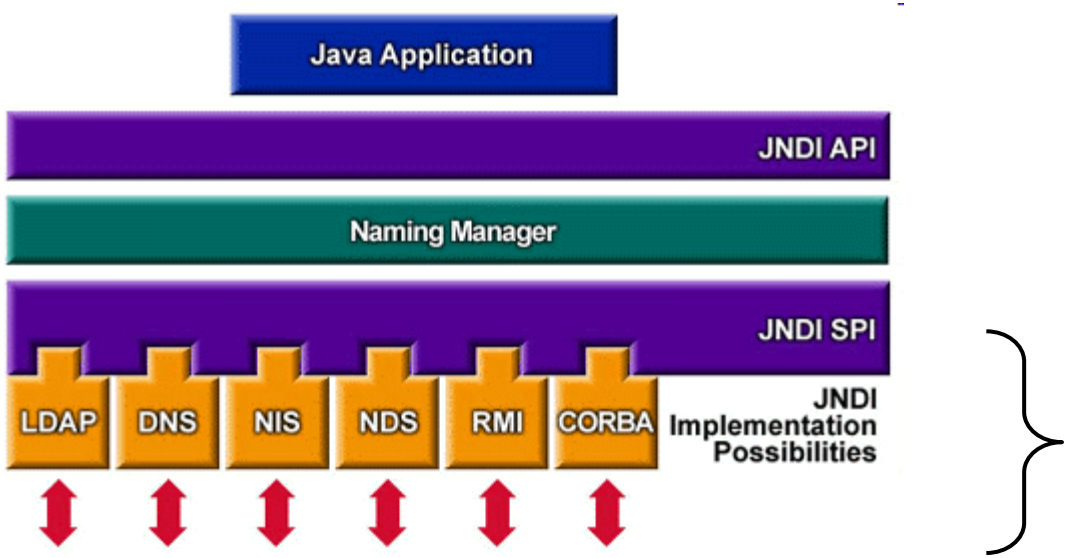
JNDI defines only an interface for client accesses and service provider plugins.

JNDI defines a common API for naming services, irrespective of the naming system used.

An application that implements the JNDI API is a JNDI provider.

The JNDI SPI (Service Provider Interface) allows to plug-in different naming service providers, e.g. for LDAP, DNS or CORBA).

An application could define its own naming service provider by implementing the JNDI SPI.



Source: Oracle

JNDI providers

8. JNDI (Java Naming and Directory Interface) (3/3)

JNDI directory services:

Besides naming services, JNDI also provides access to directory services through a common API.

Directories are structured trees of information.

Directory examples:

- **Microsoft Active Directory**
- **IBM / Lotus Notes**
- **Sun NIS (Network Information Service)**
- **LDAP (Lightweight Directory Access Protocol)**

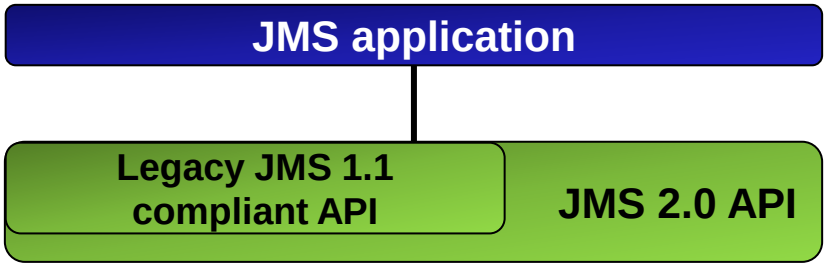
JNDI directory access example:

```
try {
  // Create the initial directory context
  DirContext ctx = new InitialDirContext();
  // Ask for all attributes of the object
  Attributes attrs = ctx.getAttributes("cn=Egli Peter");
  // Find the surname ("sn") and print it
  System.out.println("sn: " + attrs.get("sn").get());
  // Close the context when we're done
  ctx.close();
} catch (NamingException e) {
  System.err.println("Problem getting attribute: " + e);
}
```

9. JMS 2.0

JMS 2.0, released in April 2013, builds on the API defined by JMS 1.1, but brings a couple of simplifications, namely:

- **Simplified API methods in addition to the legacy JMS 1.1 API:**



- **New method to extract message body `getBody()` taking the expected body type as parameter thus obviating the need to type cast the body.**

- **New methods to create a session with fewer arguments:**
  `createSession(int sessionMode)`
  `createSession()`

- **Easier resource configuration, namely the possibility to use the Java EE 7 default connection factory by applying the @Inject annotation.**
  This eliminates the need for proprietary JMS provider connection factory configuration.