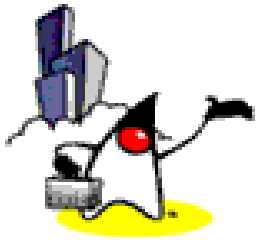
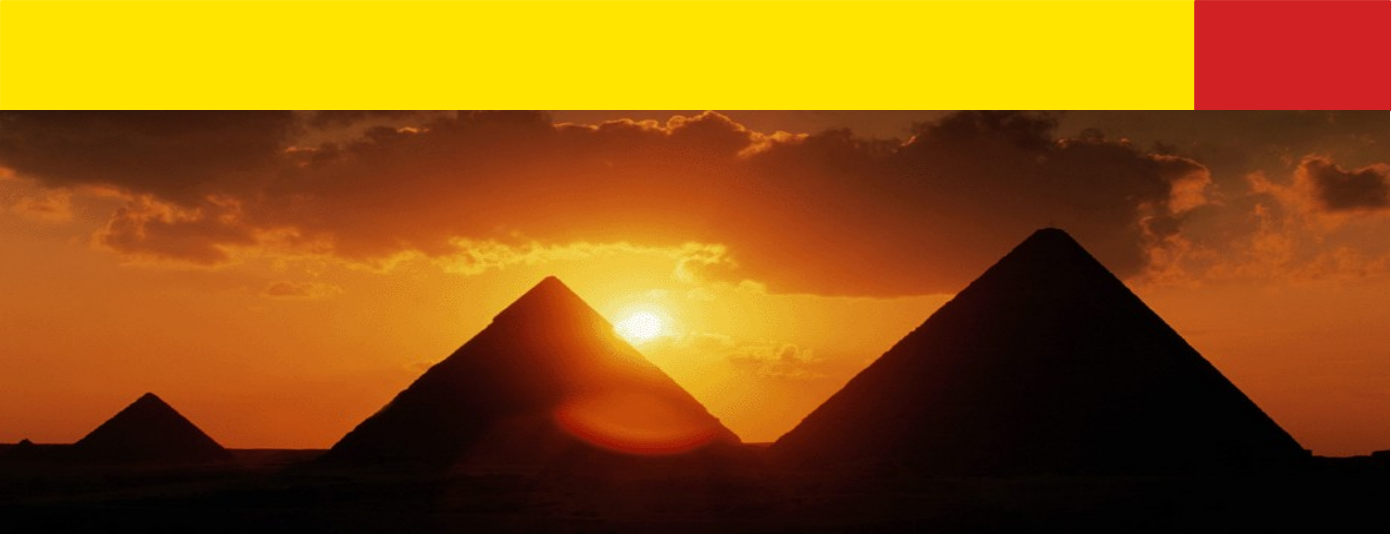
Java Messaging Service (JMS)

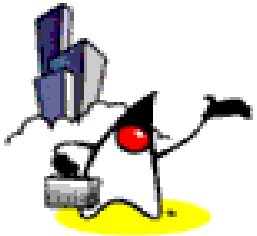


Agenda

- What is Messaging?
  - Messaging models, Reliability, Transaction, Distributed messaging, Security
- Why Messaging?
- What is JMS?
- Architecture of JMS
- JMS Programming APIs
- Steps for writing JMS clients (sender and receiver)
- JMS and EJB (MDB)



What is Messaging?



Messaging System Concepts

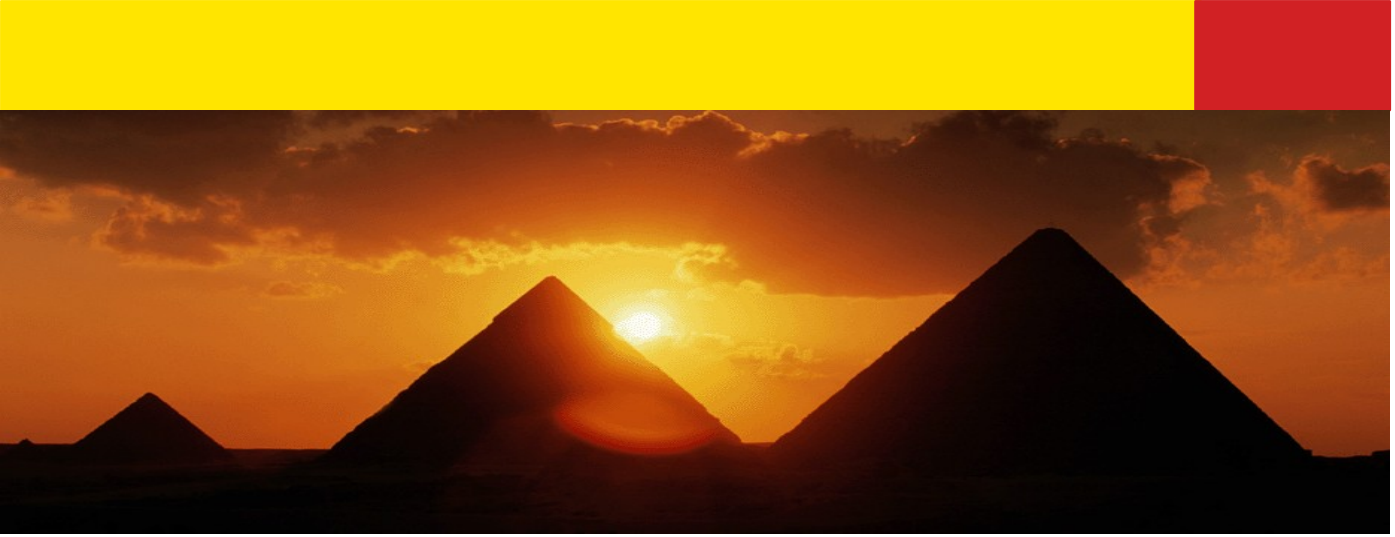
- De-coupled (Loosely-coupled) communication
- Asynchronous communication
- Messages are the means of communication between applications.
- Underlying messaging software provides necessary support
  - MOM (Message Oriented Middleware), Messaging system, Messaging server, Messaging provider, JMS provider: they all mean this underlying messaging software

Messaging System Features

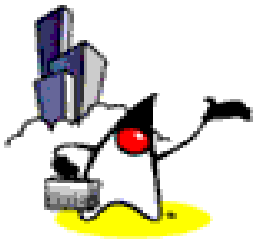
- Support of two messaging models
  - Point-to-point
  - Publish/Subscribe
- Reliability
- Transactional operations
- Distributed messaging
- Security

Additional Features

- Some Messaging System vendors support
  - Guaranteed real-time delivery
  - Secure transactions
  - Auditing
  - Metering
  - Load balancing



Messaging Models



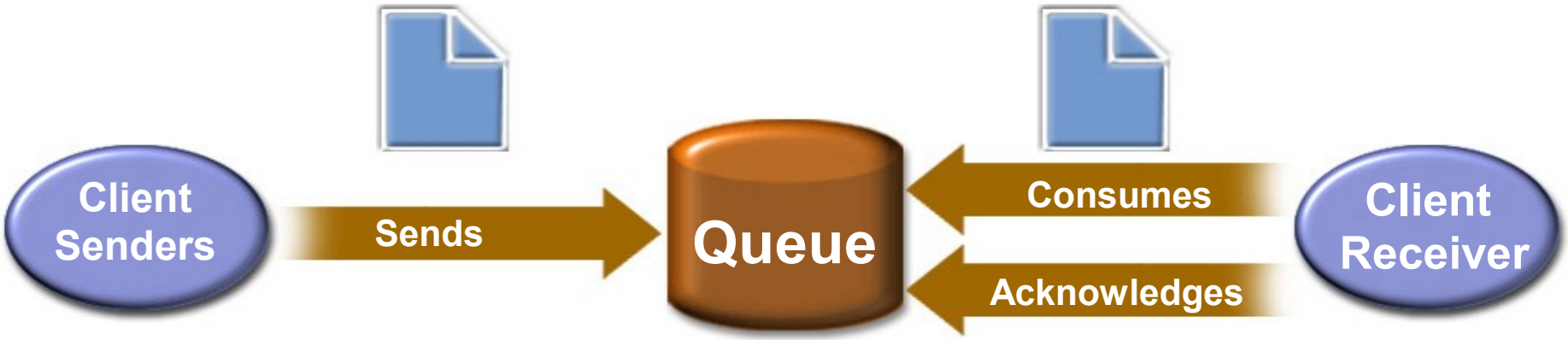
Messaging Models

- Point to Point
  - A message is consumed by a single consumer
- Publish/Subscribe
  - A message is consumed by multiple consumers

Point-to-Point

- A message is consumed by a single consumer
- There could be multiple senders
- "Destination" of a message is a named queue
- First in, first out (at the same priority level)
- Senders (producers) sends a message to a named queue (with a priority level)
- Receiver (consumer) extracts a message from the queue

Point-to-Point



Posts messages to the queue

Receives messages on the queue

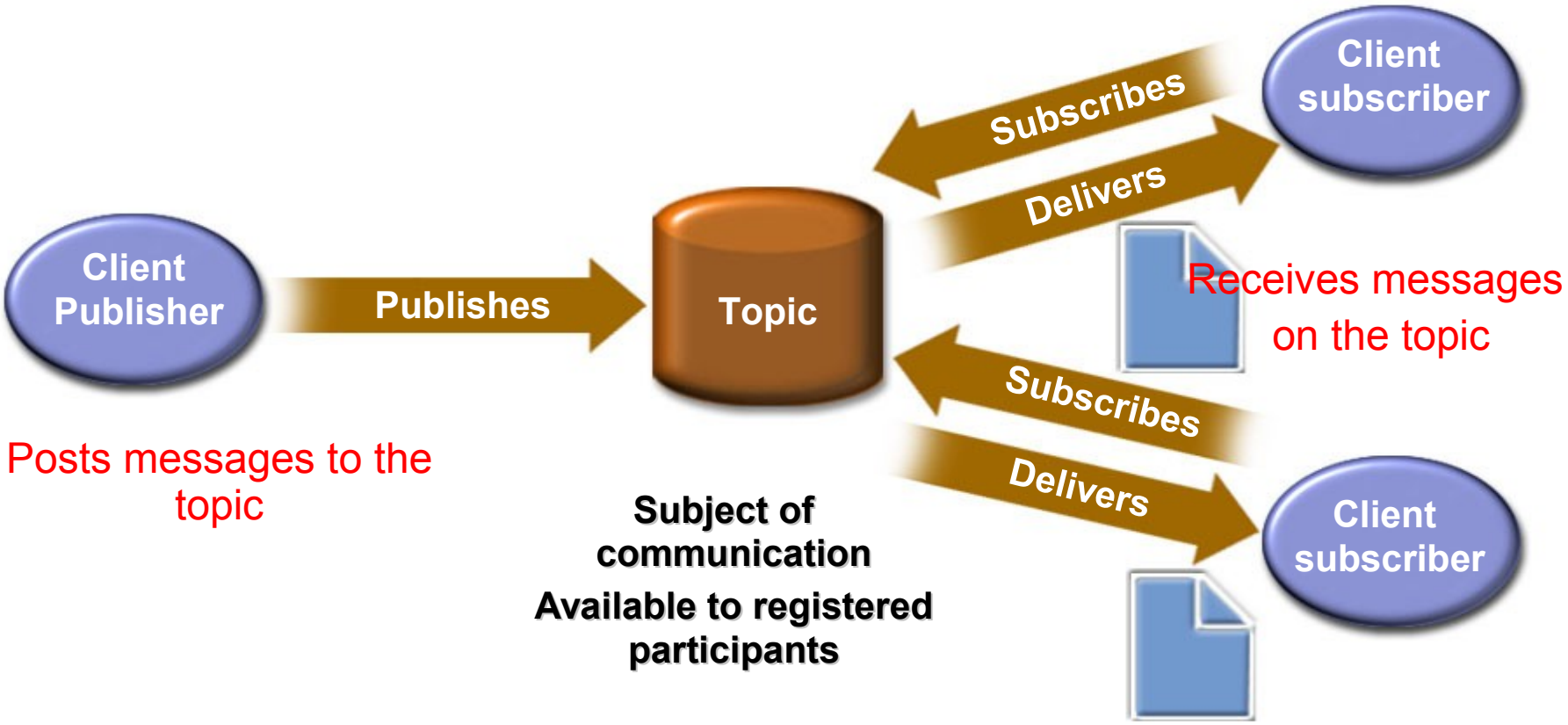
When to use Point-to-Point?

- Use it when every message you send must be processed successfully by one consumer

Publish/Subscribe (Pub/Sub)

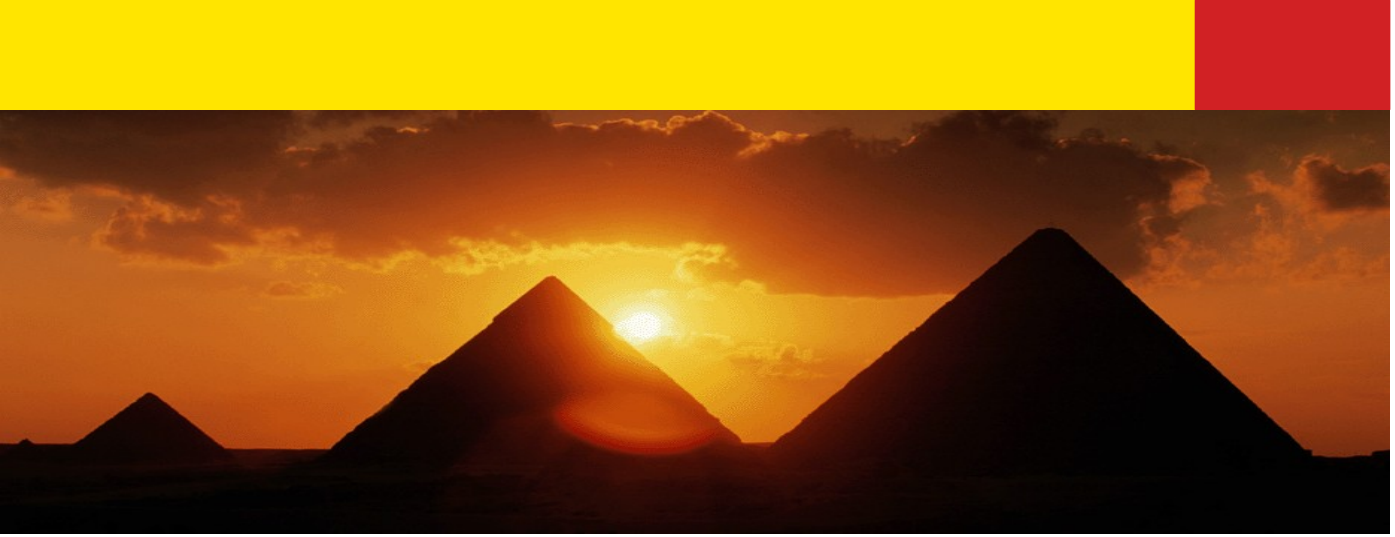
- A message is consumed by multiple consumers
- "Destination" of a message is a named topic
  - not a queue
- Producers “publish” to topic
- Consumers “subscribe” to topic

Publish-and-Subscribe

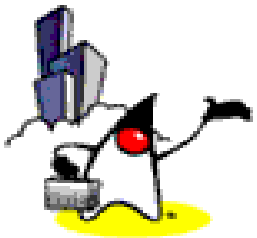


When to use Pub/Sub?

- Use it when a message you send need to be processed by multiple consumers
- Example: HR application
  - Create “new hire” topic
  - Many applications (“facilities”, “payroll”, etc.) subscribe “new hire” topic

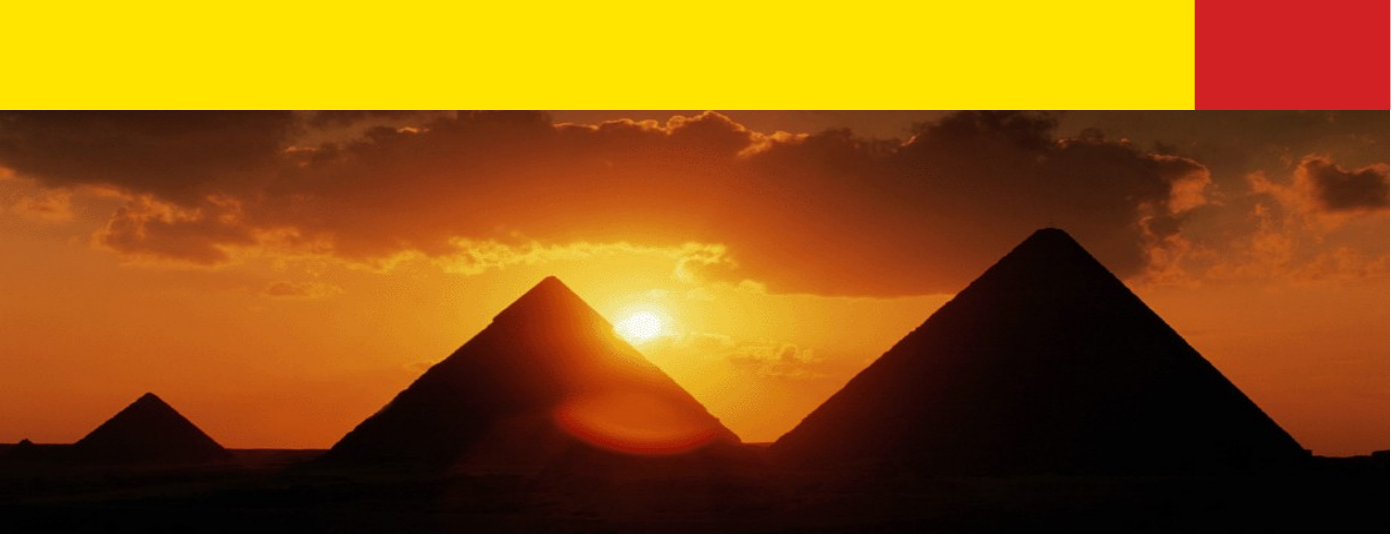


Reliability

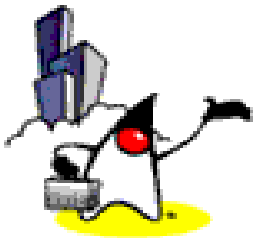


Reliability

- Some guarantee of delivery of a message
  - Different degree of reliability is possible
  - Sender can specify different level of reliability
  - Higher reliability typically means less throughput
- Typically uses persistent storage for preserving messages



Transactional Operations



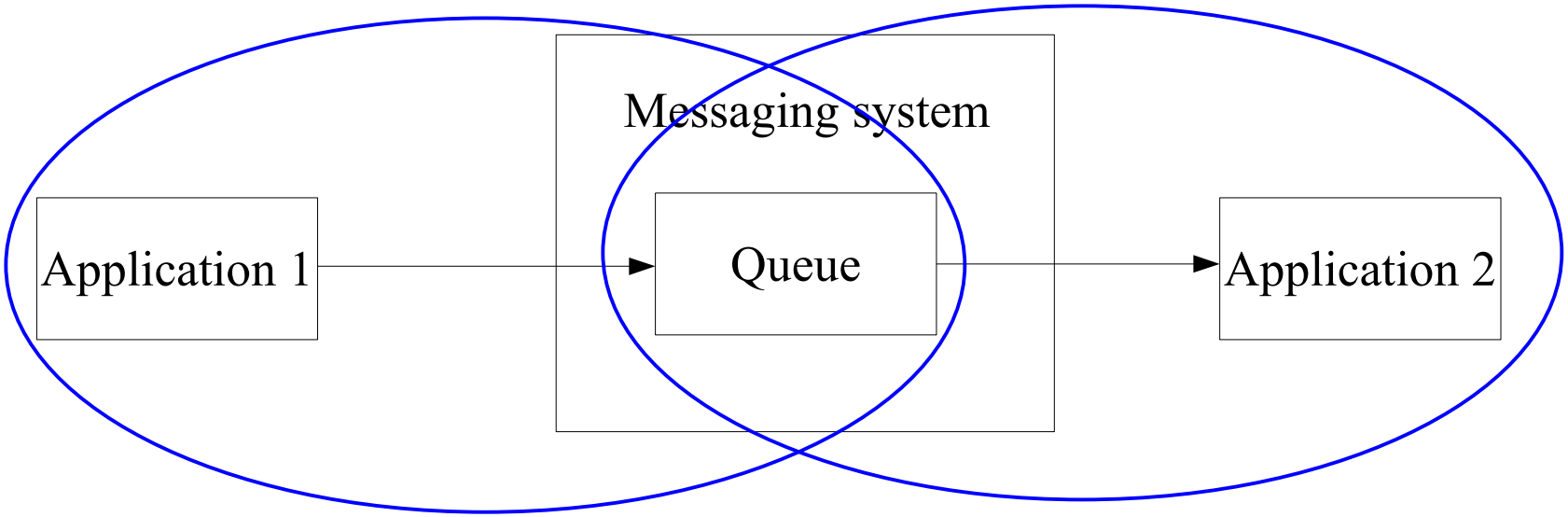
Transactional Operations

- Transactional production
  - Sender groups a series of messages into a transaction
  - Either all messages are enqueued successfully or none are
- Transactional consumption
  - Consumer retrieves a group of messages as a transaction
  - Unless all messages are retrieved successfully, the messages remain in a queue or topic

Transactional Scope

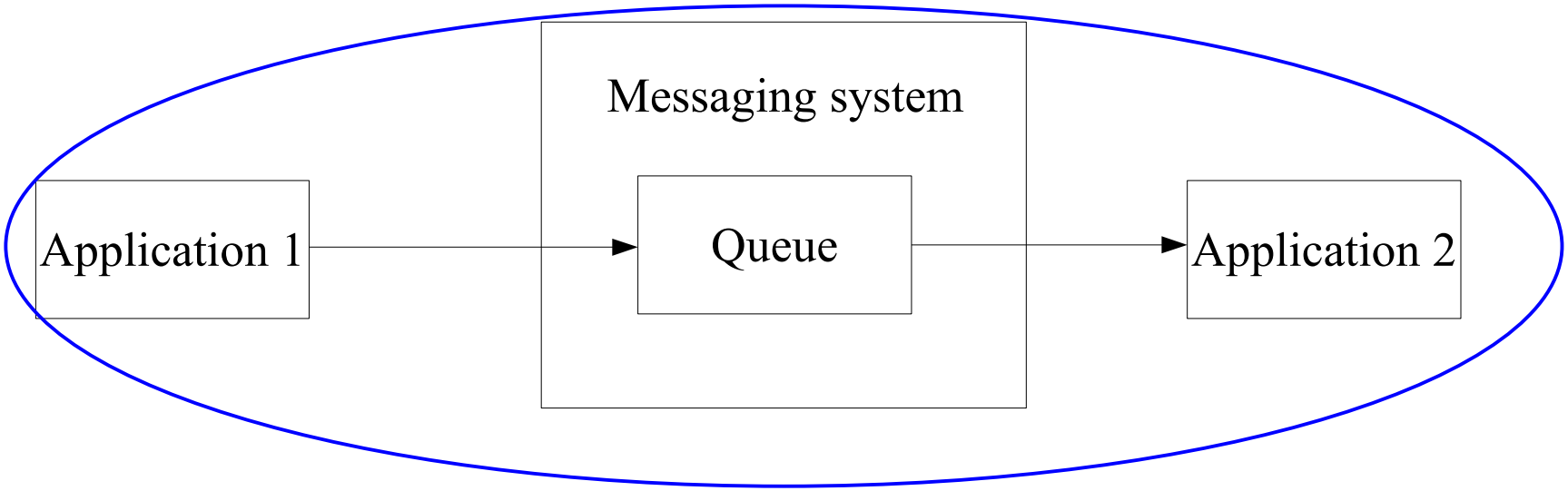
- Client-to-Messaging system scope
  - Transaction encompasses the interaction between each messaging client (applications) and the messaging system
  - JMS supports this
- Client-to-Client scope
  - Transaction encompasses both clients
  - JMS does not support this

Client-to-Messaging System Transactional Scope

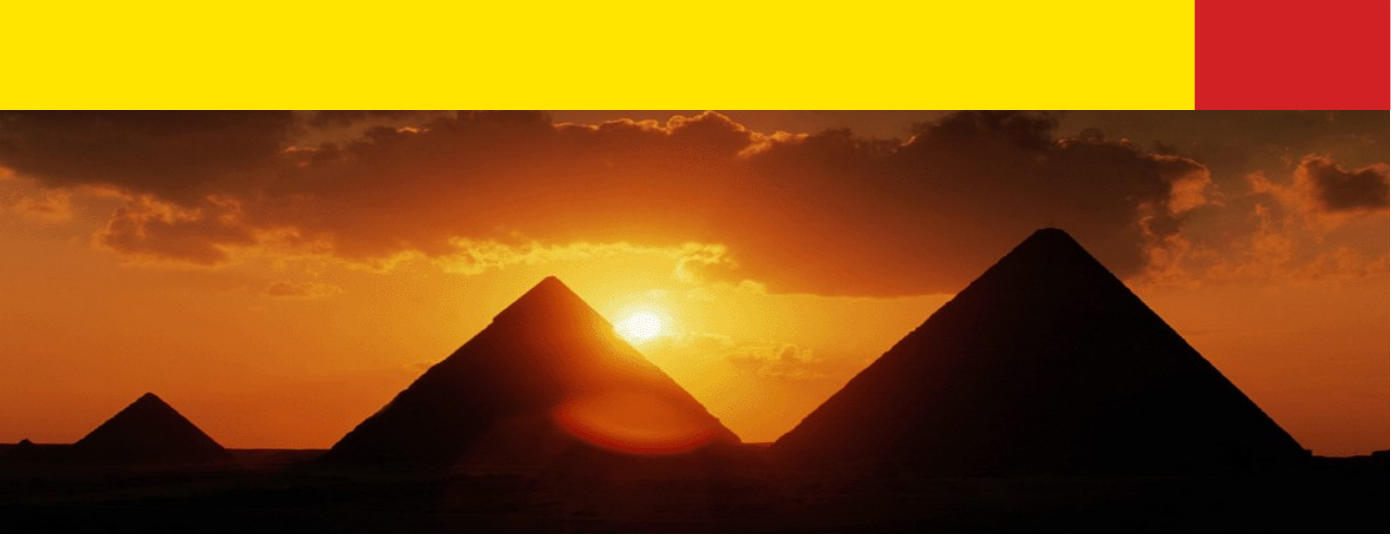


source: Applied Enterprise JavaBeans[1]

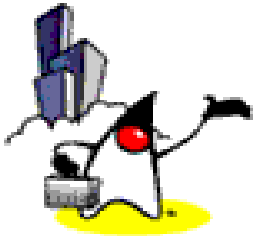
Client-to-Client Transactional Scope



source: Applied Enterprise JavaBeans[1]



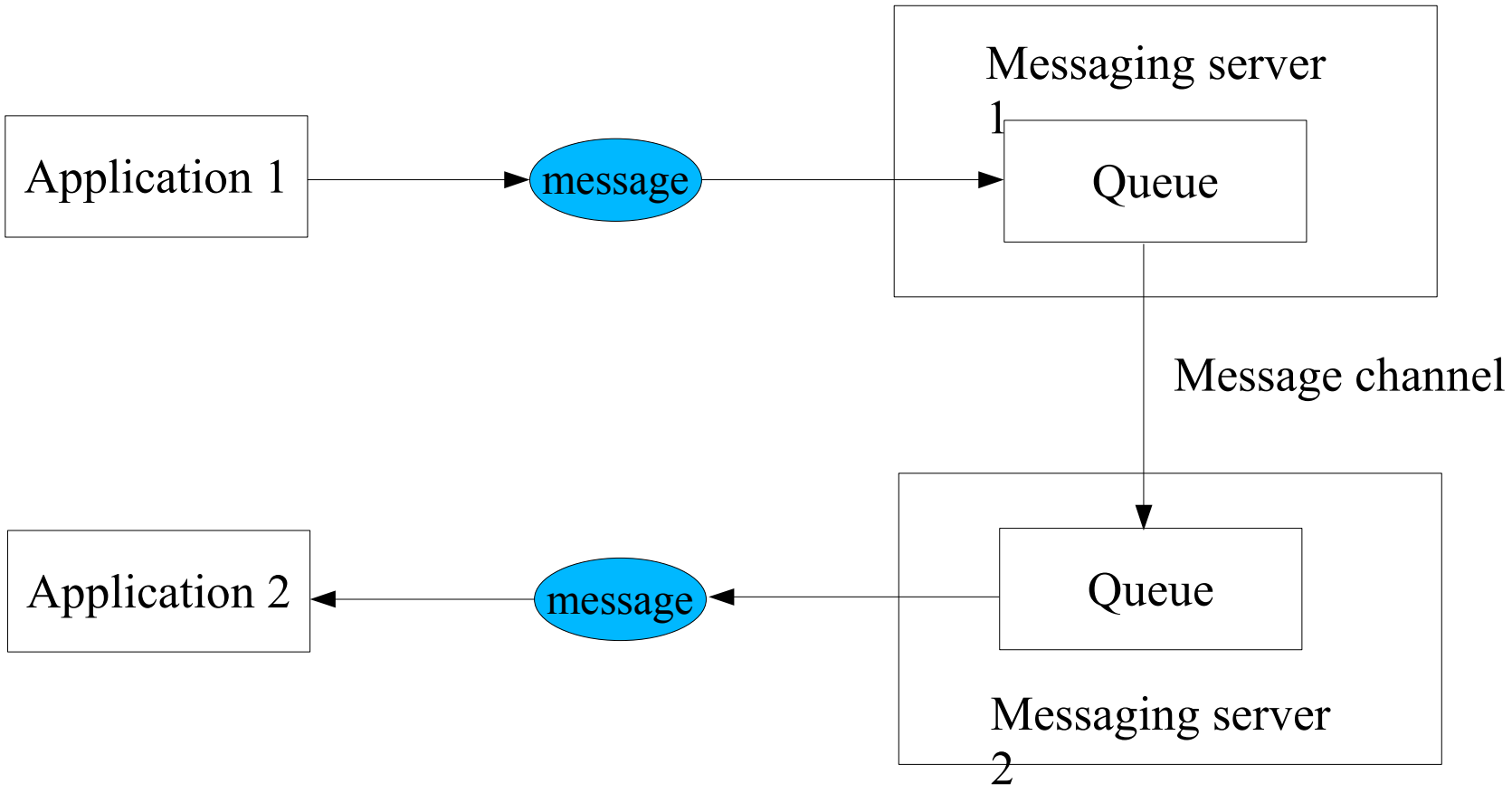
Distributed Messaging



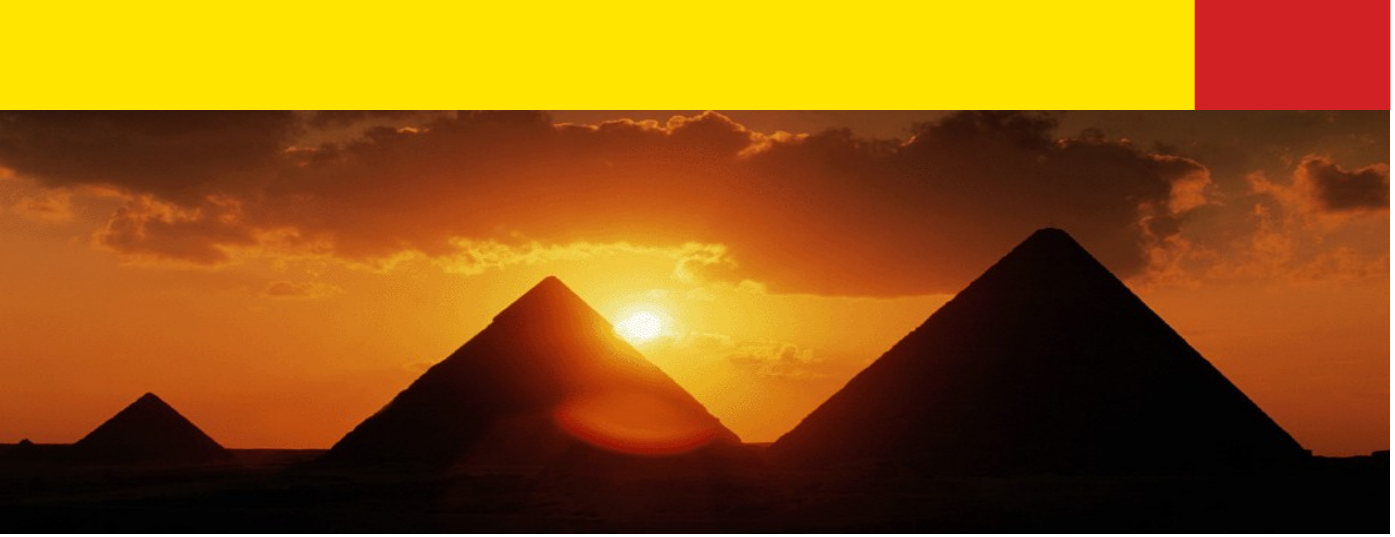
Distributed Messaging

- Enterprise messaging systems might provide an infrastructure in which messages are being forwarded between servers
  - called “message channel”

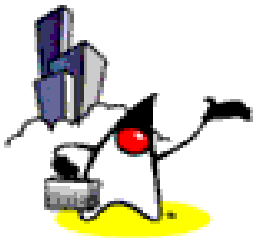
Distributed Messaging



source: Applied Enterprise JavaBeans[1]

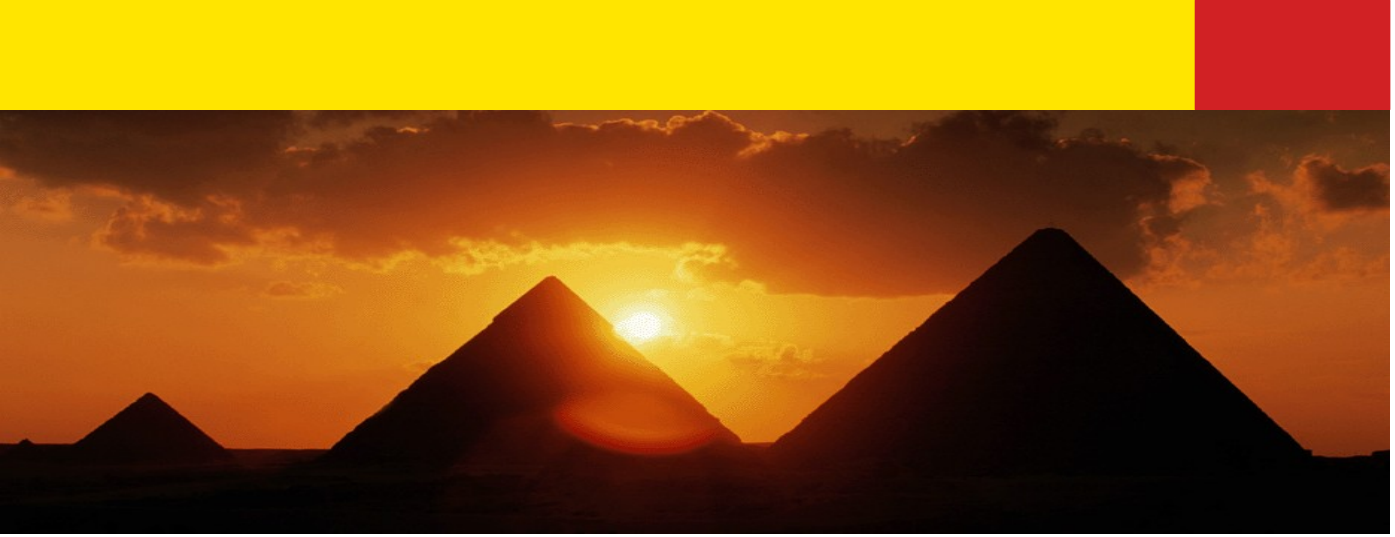


Security

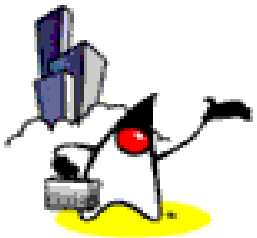


Security Issues

- Authentication
  - Messaging systems typically require clients to present signed certificates
- Confidentiality of messages
  - Messaging system typically provide encryption
- Data integrity of messages
  - Messaging system typically provide data integrity through message digest
- Security is currently handled in vendor-specific way



Why Messaging?



Why Messaging?

- Platform independence
- Network location independence
- Works well in heterogeneous environments

Why Messaging?

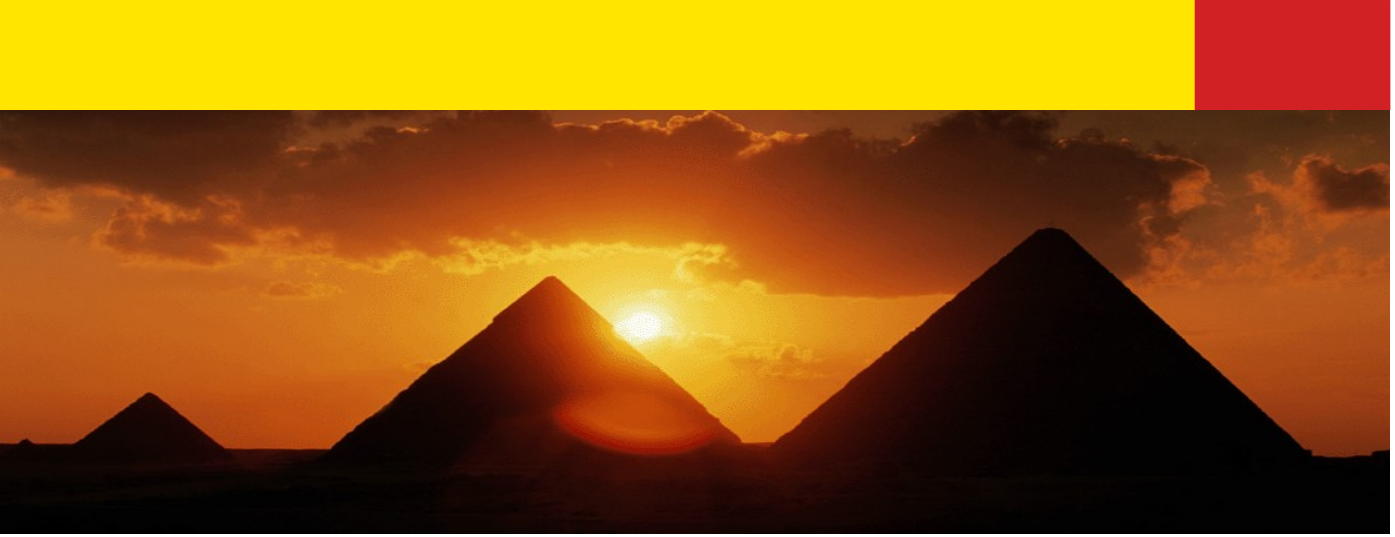
- Anonymity
  - Who doesn't matter
  - Where doesn't matter
  - When doesn't matter
- Contrast with RPC-based systems
  - CORBA
  - RMI

Why Messaging?

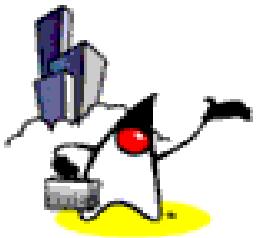
- Scalability
  - Handle more clients with
    - No change in the application
    - No change in the architecture
    - No degradation in system throughput
  - Increase hardware capacity of messaging system if higher scalability is desired

Why Messaging?

- Robustness
  - Receivers can fail.
  - Senders can fail.
  - Network can fail.
  - Messaging System continues to function.

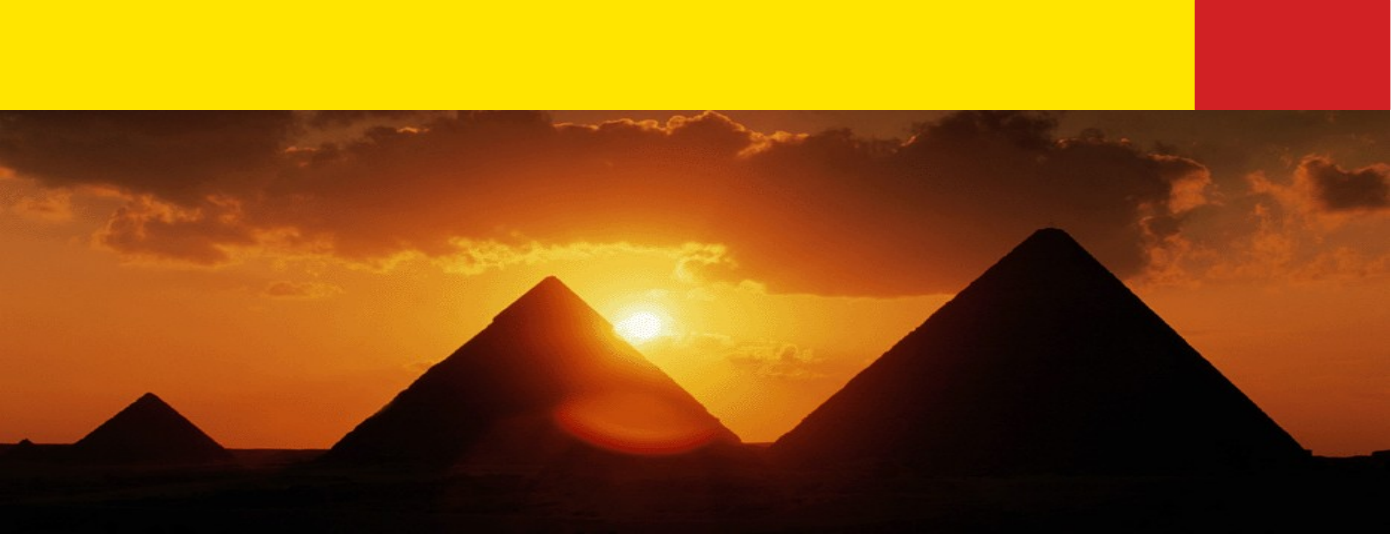


Example Messaging Applications

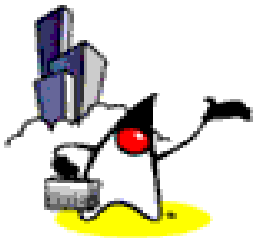


Messaging Applications

- Credit card transactions
- Weather reporting
- Workflow
- Network management
- Supply chain management
- Customer care
- Communications (Voice Over IP, Paging Systems, etc.)
- Many more



What is JMS?



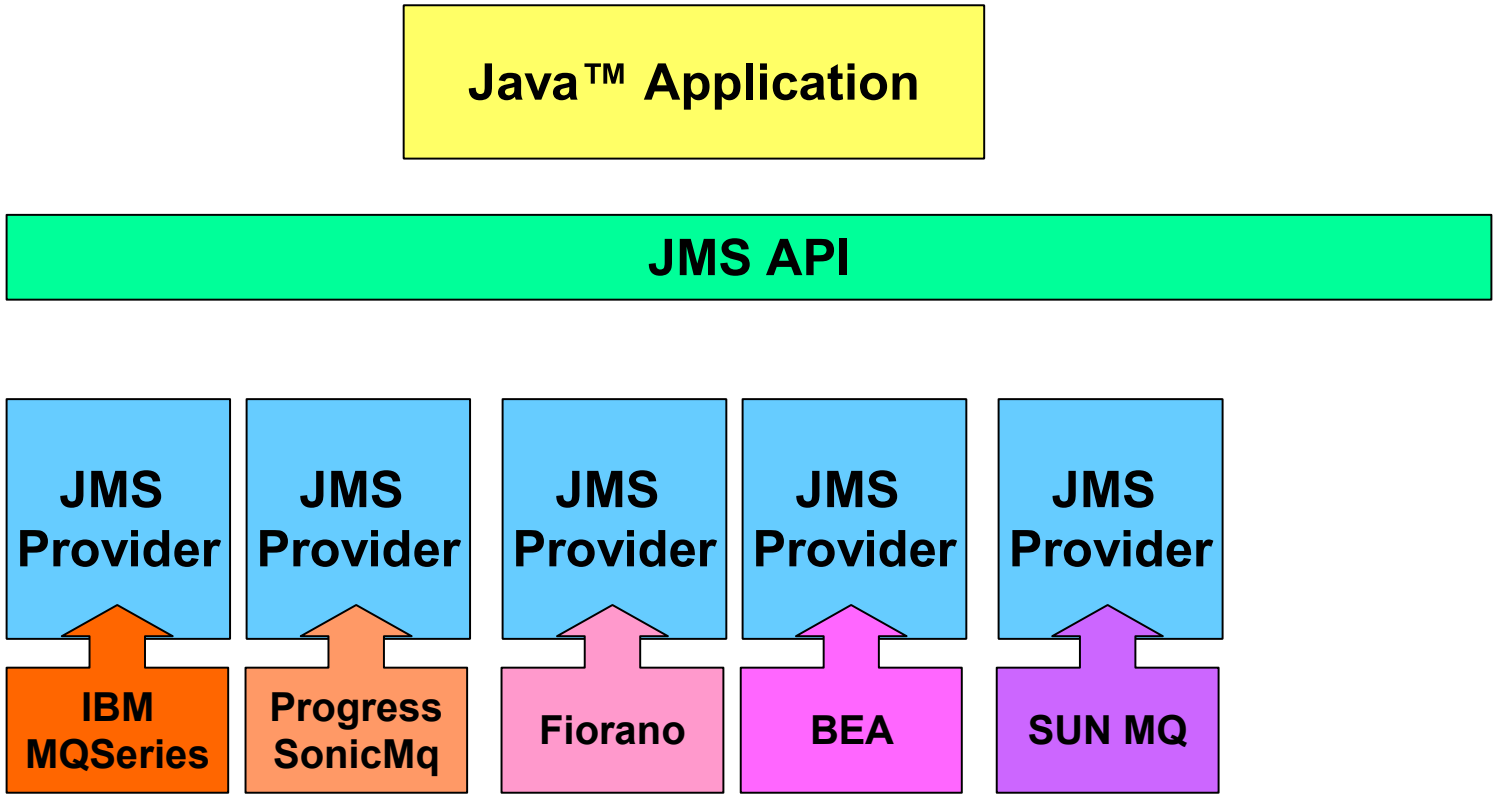
What is JMS?

- JMS is a set of Java interfaces and associated semantics (APIs) that define how a JMS client accesses the facilities of a messaging system
- Supports message production, distribution, delivery
- Supported message delivery semantics
  - Synchronous or Asynchronous
  - transacted
  - Guaranteed
  - Durable

What is JMS? (Continued)

- Supports existing messaging models
  - Point-to-Point (reliable queue)
  - Publish/Subscribe
- Message selectors (on the receiver side)
- 5 Message types

JMS is an API

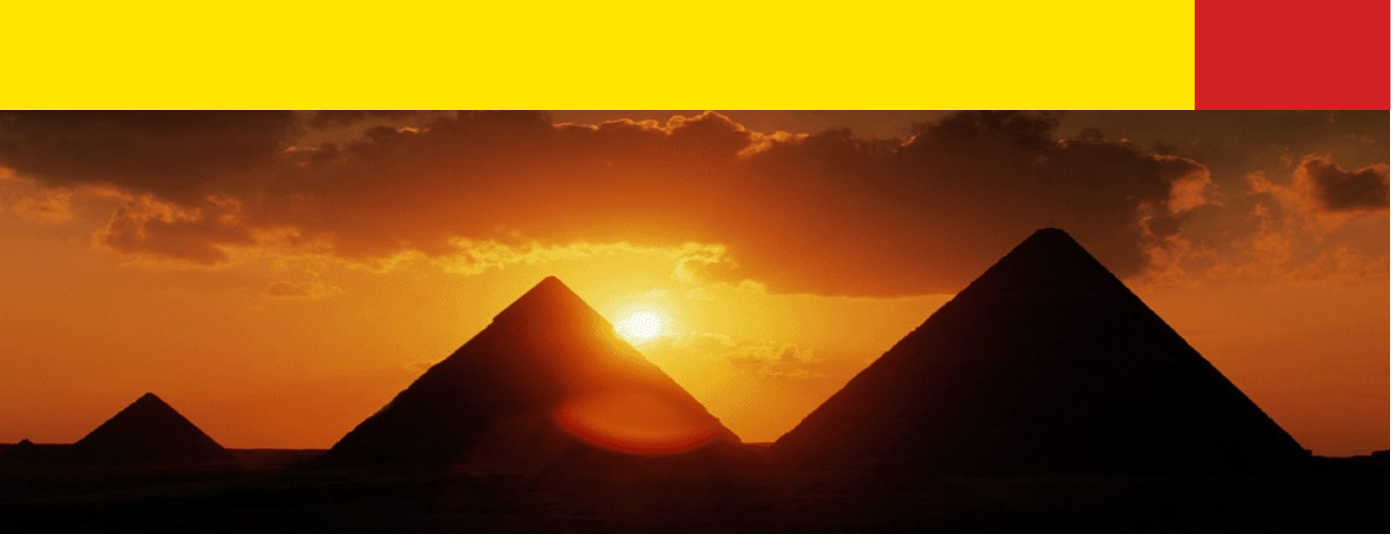

Java™ Application
JMS API
JMS Provider
JMS Provider
JMS Provider
JMS Provider
JMS Provider
IBM MQSeries
Progress SonicMq
Fiorano
BEA
SUN MQ

JMS and J2EE

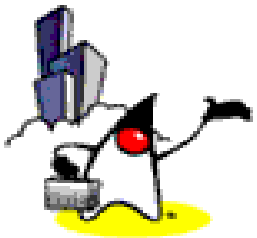
- Allows Java Developers to access the power of messaging systems
- Part of the J2EE Enterprise Suite

JMS Design Goals

- Consistency with existing APIs
- Independence from Messaging system provider
- Minimal effort on part of Messaging system provider
- Provide most of the functionality of common messaging systems
- Leverage Java technology



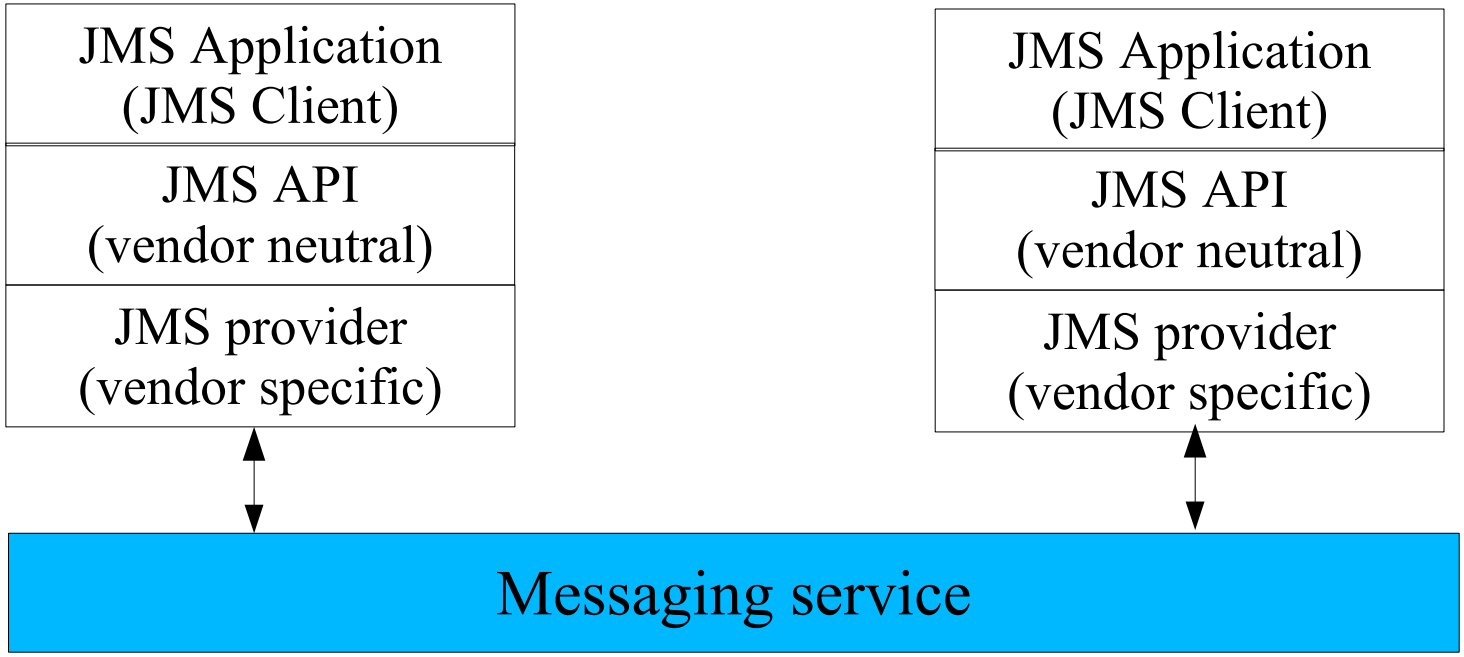
Architecture of a JMS Application



JMS Architectural Components

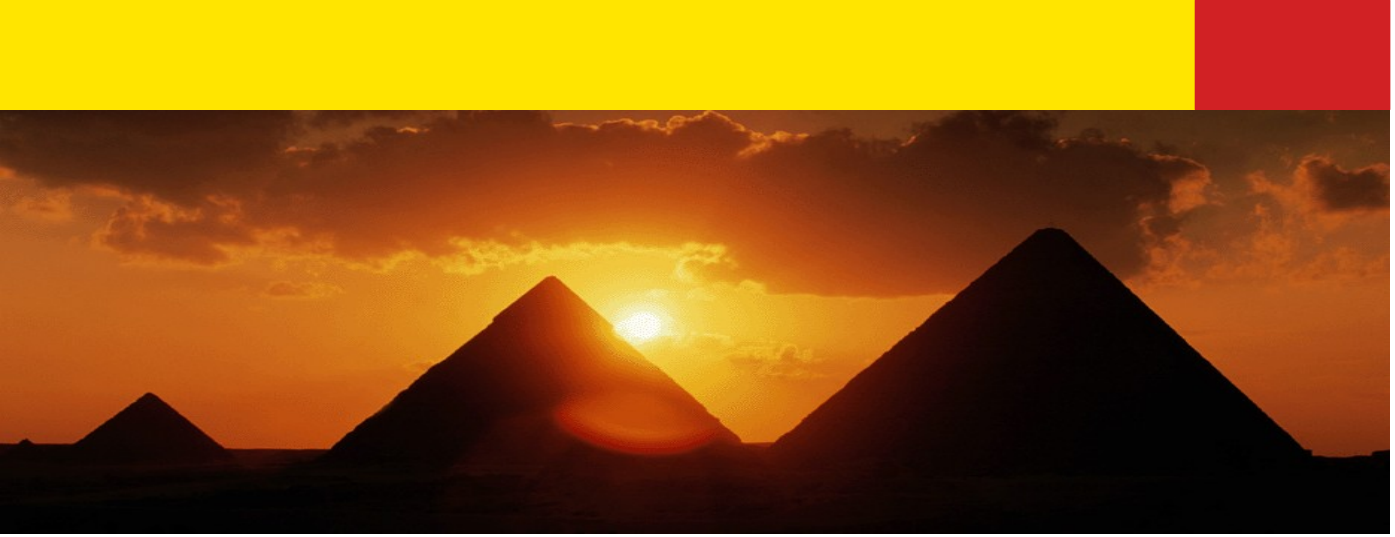
- JMS clients
- Non-JMS clients
- Messages
- JMS provider (Messaging systems)
- JNDI administered objects
  - Destination
  - ConnectionFactory

Architecture of JMS Application

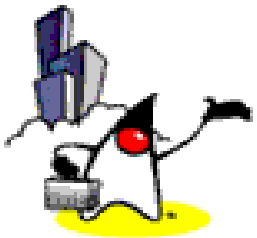


JMS Terminology

- Domain (Messaging modes)
  - point-to-point, publish/subscribe
- Session
- Connection
- Destination
- Produce, send, publish
- Consume, receive, subscribe



JMS Domains (Messaging Models)

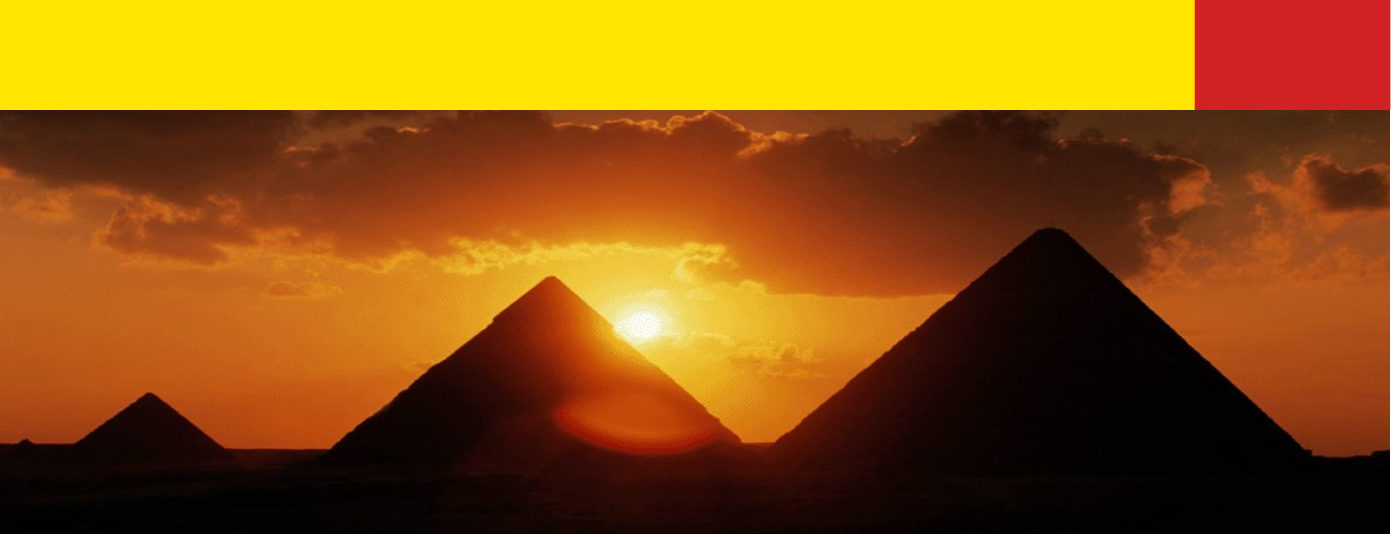


JMS Domains (Messaging Styles)

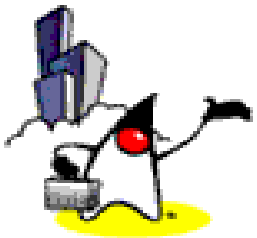
- JMS Point-to-Point
  - Messages on a queue can be persistent or non-persistent
- JMS Pub/Sub
  - Non-durable
  - Durable

JMS Pub/Sub Non-durable vs. JMS Pub/Sub Durable

- Non-durable
  - Messages are available only during the time for which the subscriber is active
  - If subscriber is not active (not connected), it will miss any messages supplied in its absence
- Durable
  - Messages are retained on behalf of subscribers that are not available at the time the message was produced



JMS Messages



Messages

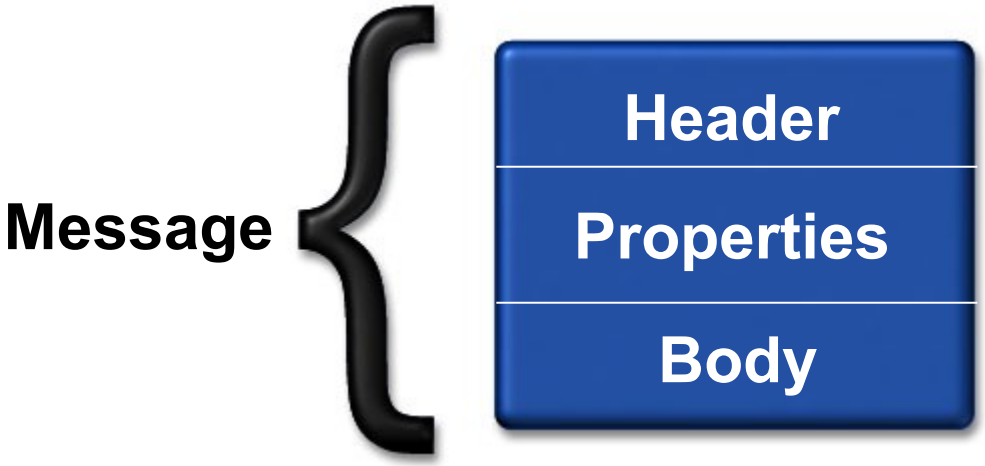
- Messages are the means of communication between messaging applications
- Actual on-the-wire Message formats vary widely among messaging systems
  - A messaging system can interoperate with only with the same messaging system

Message Java Interface

- JMS provides a unified and abstract message model via this interface
- Actual object implementation of this interface is provider specific

Message Components

- Header
- Properties
- Body



Message Header

- Used for message identification and routing
- Includes Destination
- Also includes other data:
  - delivery mode (persistent, nonpersistent)
  - message ID
  - timestamp
  - priority
  - ReplyTo

Message Header Fields

- JMSDestination
- JMSDeliveryMode
  - persistent or nonpersistent
- JMSMessageID
- JMSTimeStamp
- JMSRedelivered
- JMSExpiration

Message Header Fields

- JMSPriority
- JMSCorrelationID
- JMSReplyTo
  - Destination supplied by a client; where to send reply
- JMSType
  - Type of message body

Message Properties

- Application-specific fields
- Messaging system provider-specific fields
- Optional fields
- Properties are Name/value pairs
- Values can be byte, int, String, etc.

Message Body

- Holds content of message
- Several types supported
- Each type defined by a message interface:
  - StreamMessage
  - MapMessage
  - TextMessage
  - ObjectMessage
  - BytesMessage

Message Body Interfaces

- StreamMessage:
  - Contains Java primitive values
  - Read sequentially
- MapMessage:
  - Holds name/value pairs
  - Read sequentially or by name
- BytesMessage
  - Uninterpreted bytes
  - Used to match an existing message format

Example: Creating Text Message

- To create a simple **`TextMessage`**:

```
TextMessage message =
      session.createTextMessage();
message.setText("greetings");
```

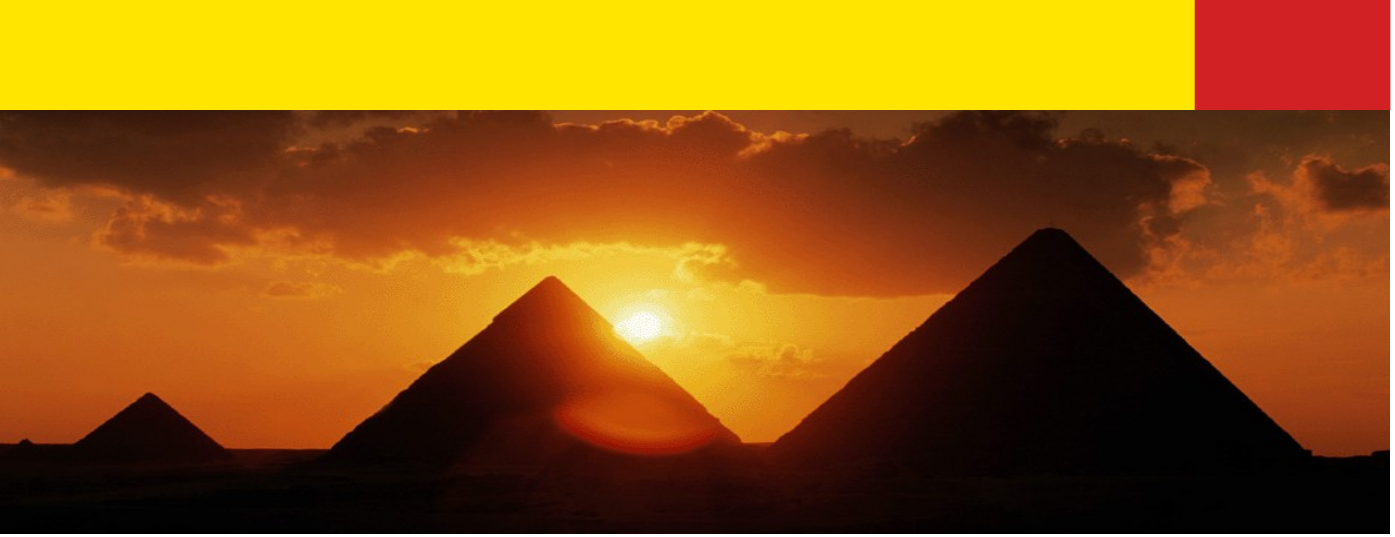
Example: Creating Object Message

- To create a simple **`ObjectMessage`**:

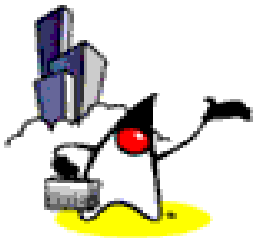
```
ObjectMessage message =
    session.createObjectMessage();
message.setObject(myObject);
```

NOTE*: myObject* must implement

`java.io.Serializable`



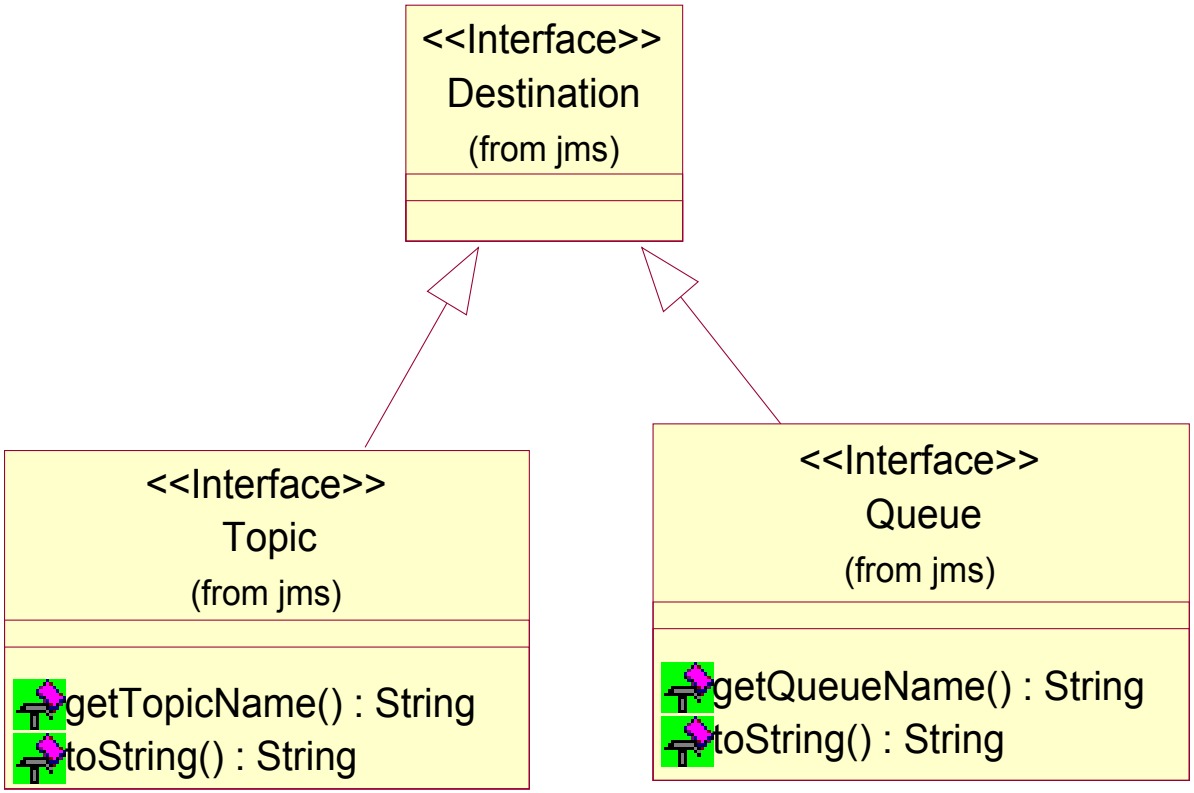
JMS Programming APIs



Destination Java Interface

- Represents an abstraction of topic or queue (not a receiver)
- Parent interface of Queue and Topic interfaces

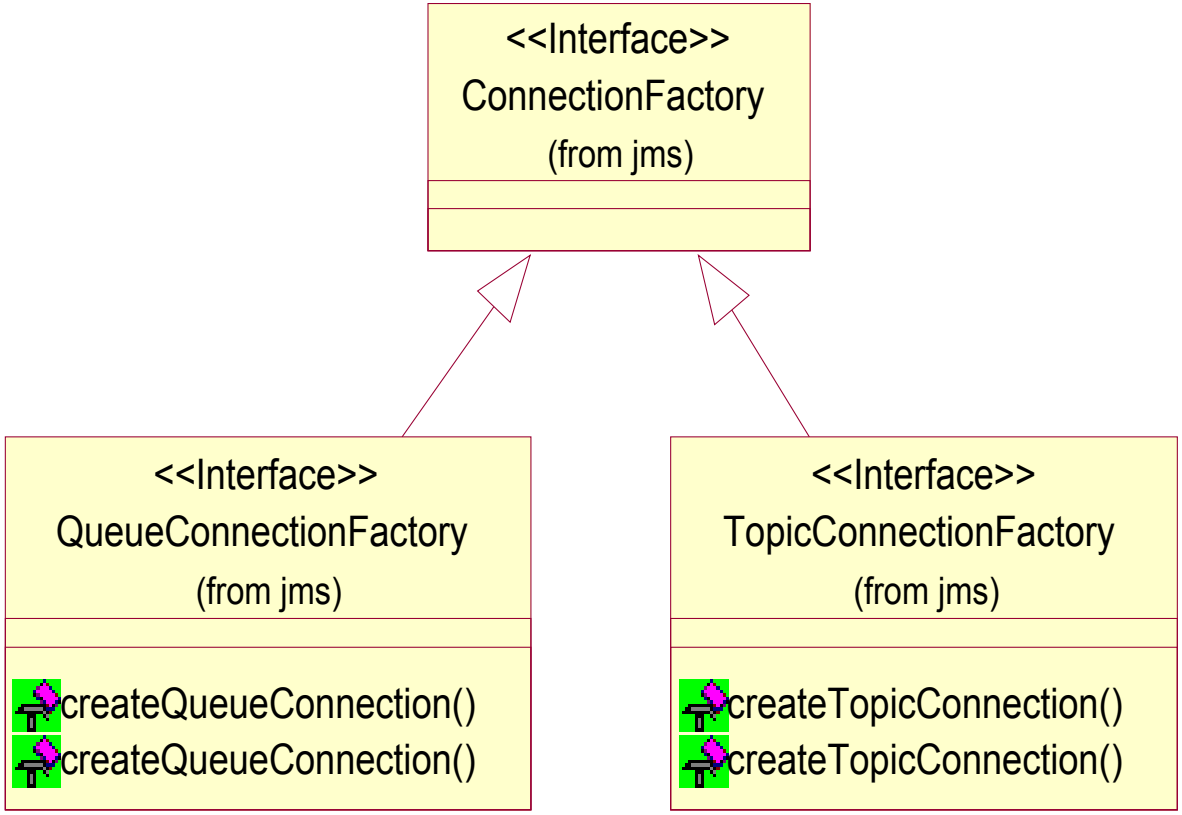
Destination Java Interface



ConnectionFactory Java Interface

- Factory class for creating a provider specific connection to the JMS server
- Analogous to the driver manager (java.sql.DriverManager) in JDBC
- Parent interface of QueueConnectionFactory and TopicConnectionFactory interfaces

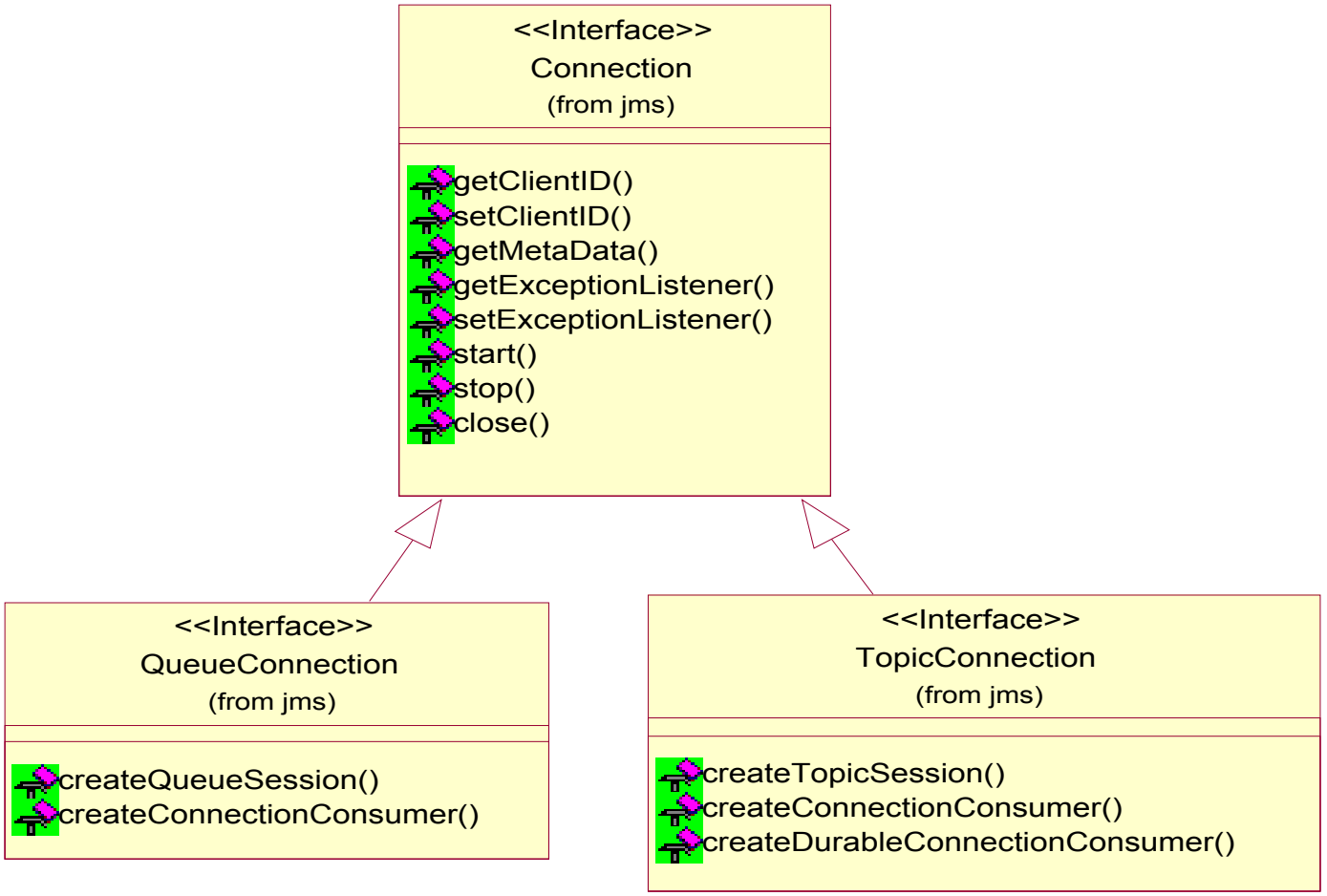
ConnectionFactory Java Interface



Connection Java Interface

- An abstraction that represents a single communication channel to JMS provider
- Created from a ConnectionFactory object
- A connection should be closed when the program is done using it.

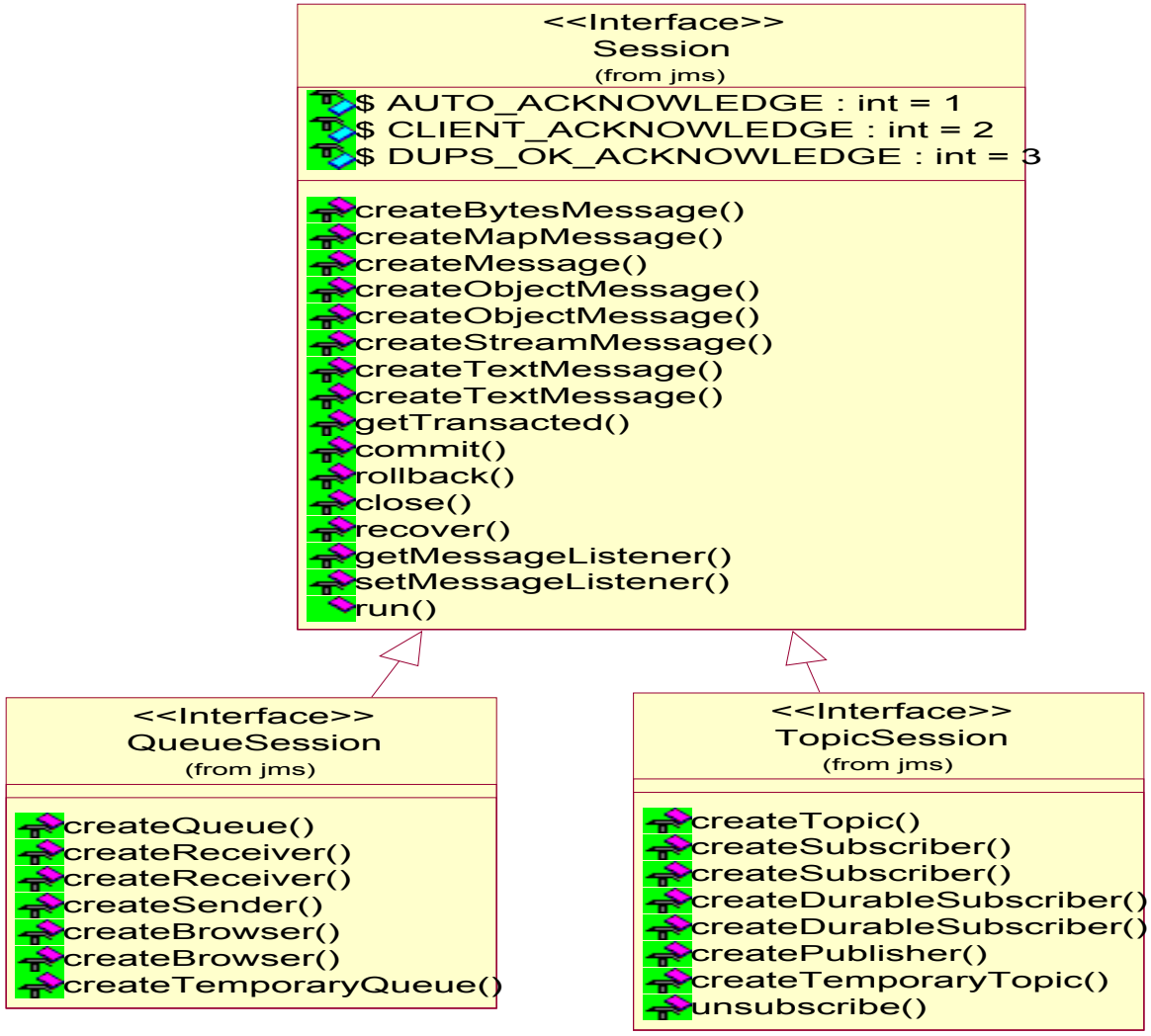
Connection Java Interface



Session Java Interface

- Created from a Connection object
- Once connected to the provider via a Connection, all work occurs in the context of a Session
- A session is single threaded, which means that any message sending and receiving happens in a serial order, one after the other
- Sessions also provide a transactional context

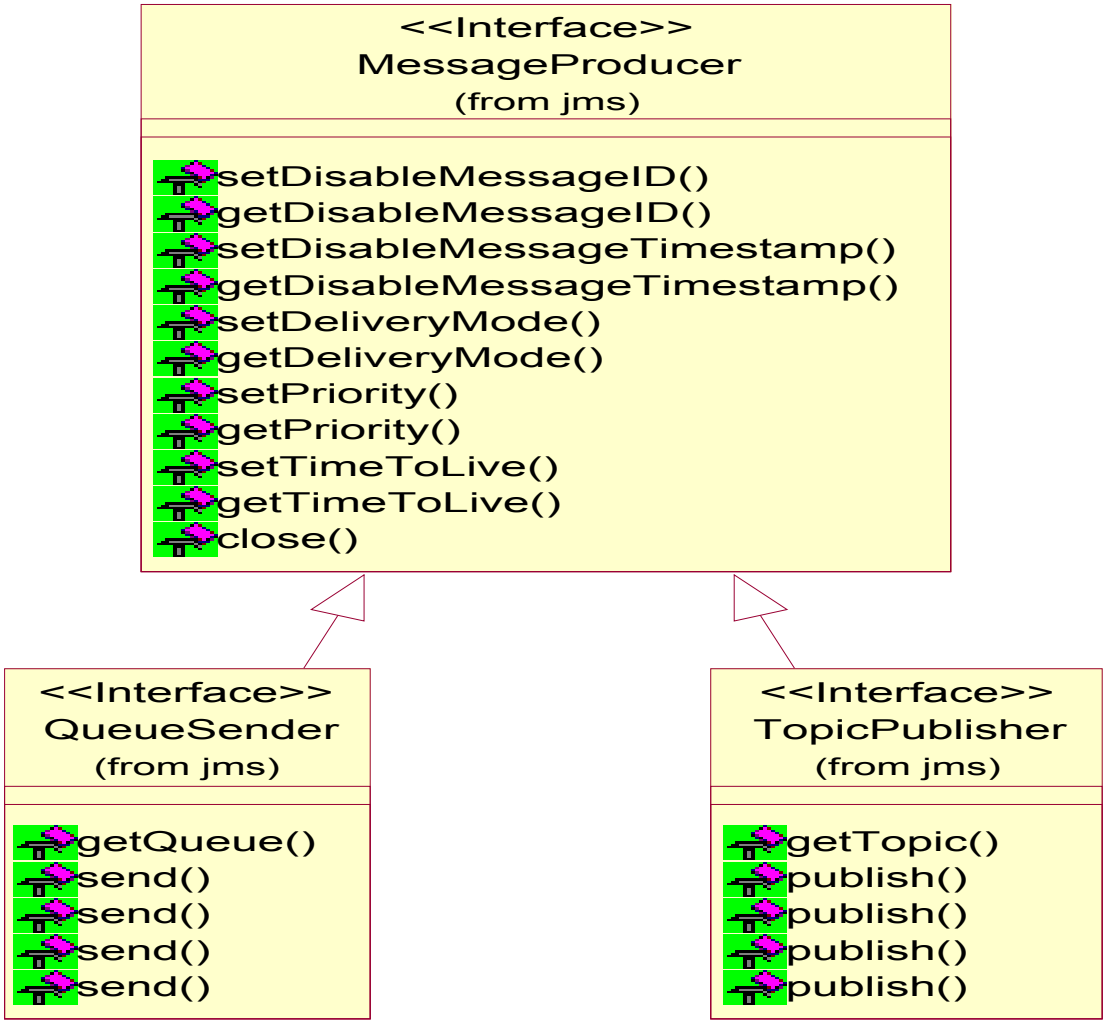
Session Java Interface



MessageProducer Java Interface

- To send a message to a Destination, a client must ask the Session object to create a MessageProducer object

MessageProducer Java Interface



MessageConsumer Java Interface

- Clients which want to receive messages create MessageConsumer object via Session object
- MessageConsumer object is attached to a Destination object
- Client can receive messages in two different modes
  - Blocking
  - Non-blocking

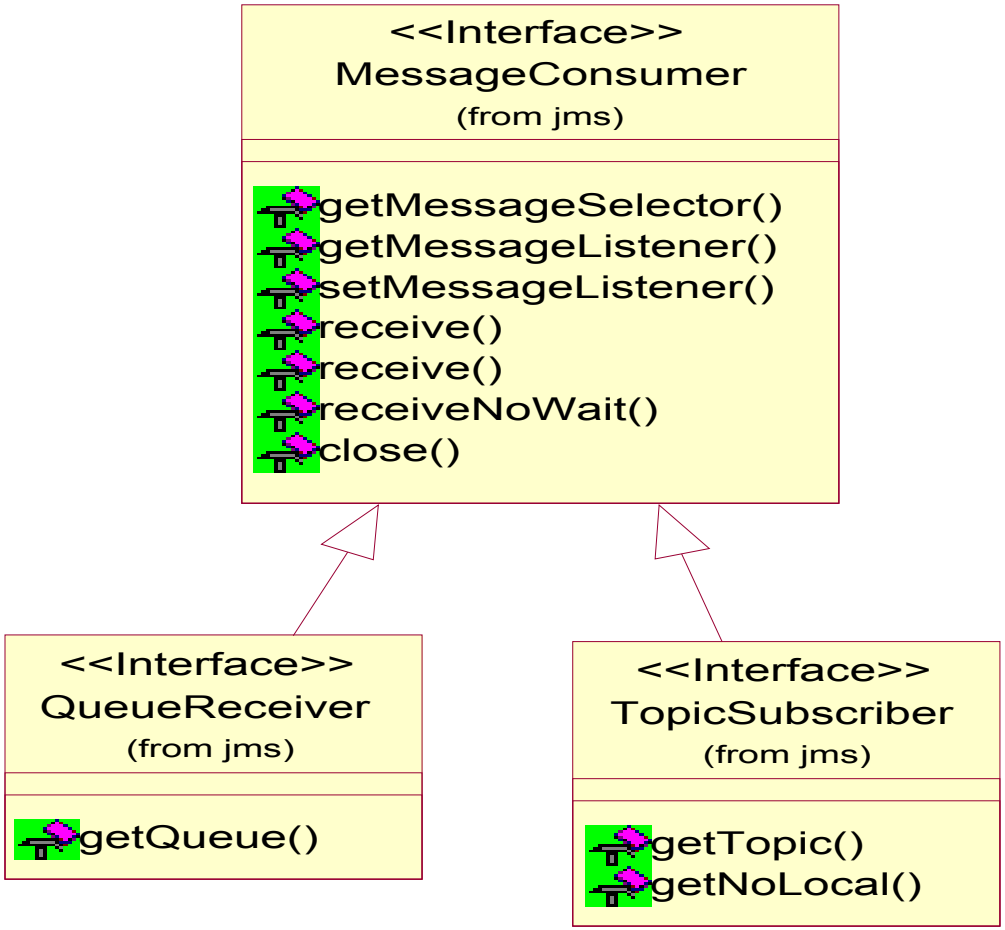
Receiving Messages in Blocking mode

- Client calls receive() method of MessageConsumer object
- Client blocks until a message is available

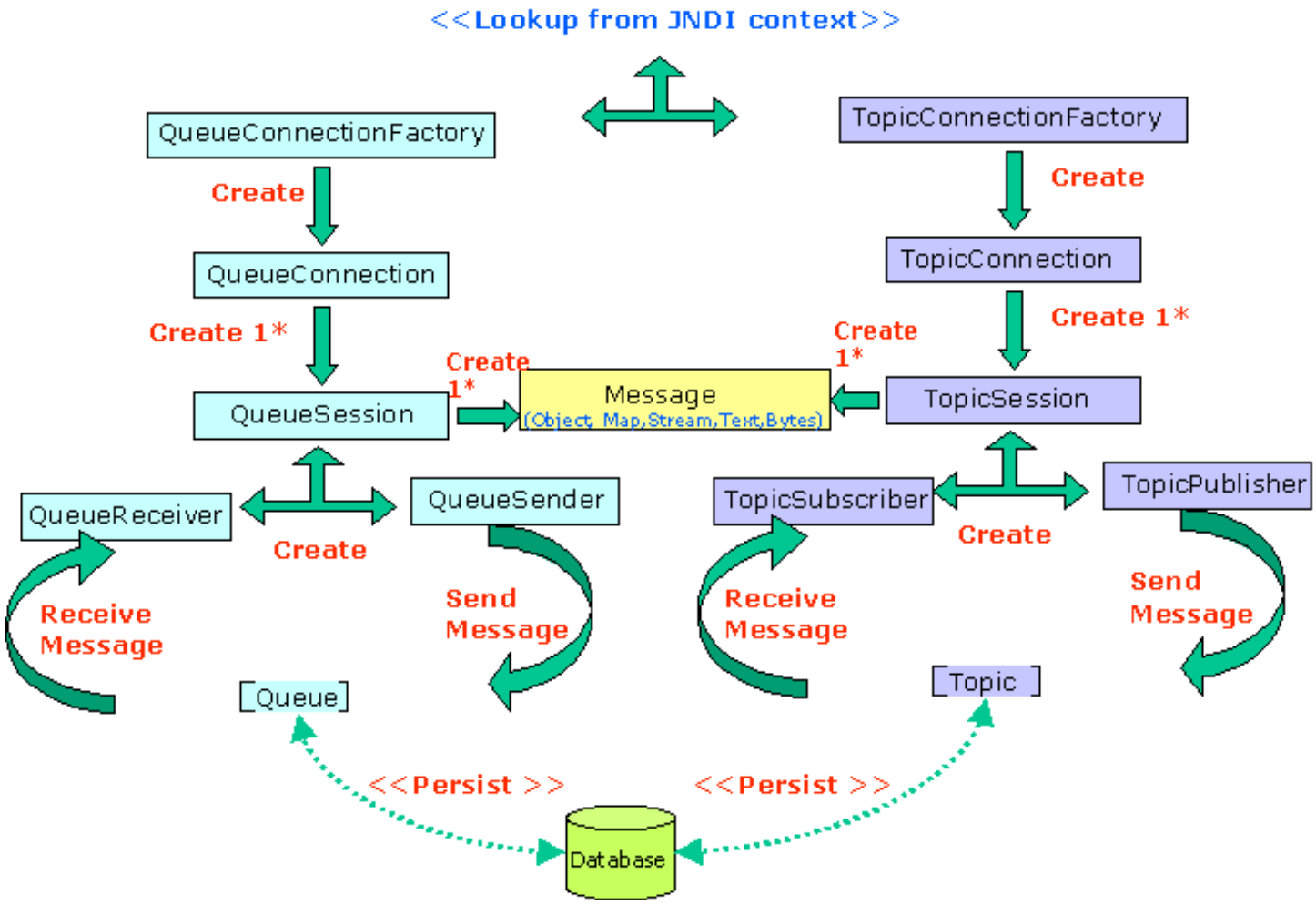
Receiving Messages in Non-blocking mode

- Client registers a MessageListener object
- Client does not block
- When a message is available, JMS provider then calls onMessage() method of the MessageListener object

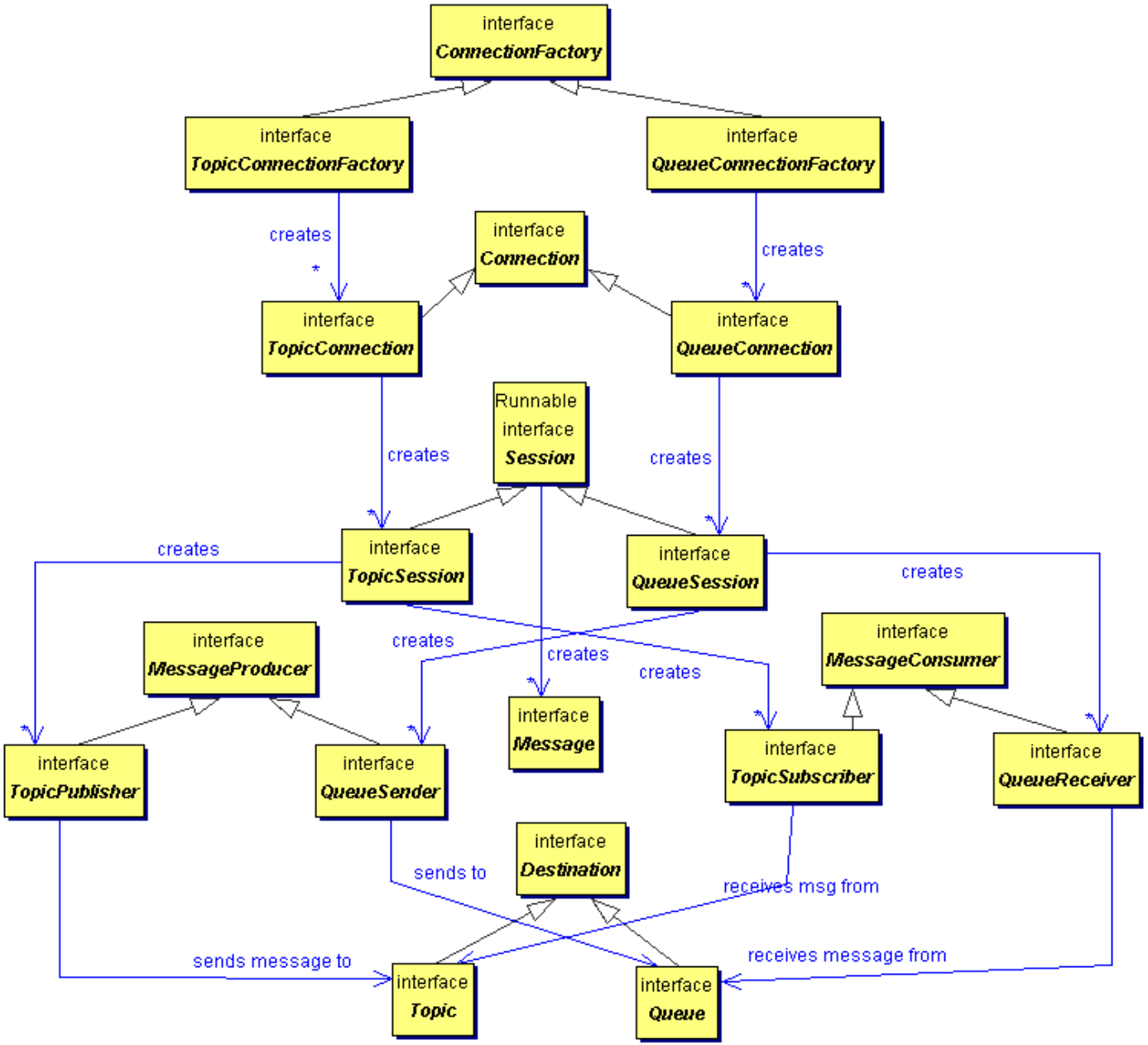
MessageConsumer Java Interface

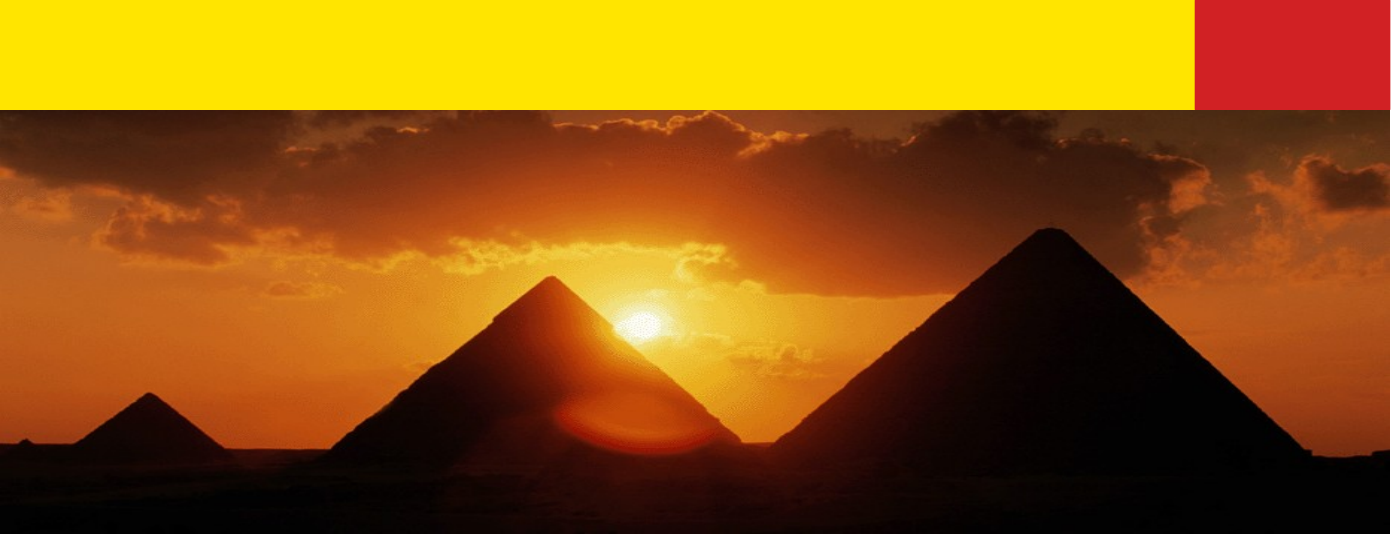


JMS APIs

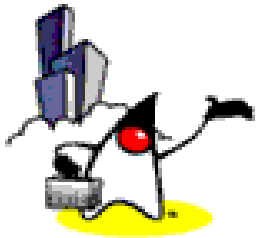

<<Lookup from JNDI context>>
QueueConnectionFactory
Create
QueueConnection
Create 1*
QueueSession
Create 1*
Message
(Object, Map,Stream,Text,Bytes)
QueueReceiver
Create
QueueSender
Receive Message
Send Message
[Queue]
TopicConnectionFactory
Create
TopicConnection
Create 1*
Create 1*
TopicSession
TopicSubscriber
Create
TopicPublisher
Receive Message
Send Message
[Topic]
<<Persist >>
<<Persist >>
Database

JMS API


interface
ConnectionFactory
interface
TopicConnectionFactory
interface
QueueConnectionFactory
creates
*
creates
*
interface
Connection
interface
TopicConnection
interface
QueueConnection
Runnable
interface
Session
creates
creates
creates
interface
TopicSession
interface
QueueSession
creates
creates
creates
creates
interface
MessageProducer
interface
MessageConsumer
interface
Message
interface
TopicPublisher
interface
QueueSender
interface
TopicSubscriber
interface
QueueReceiver
interface
Destination
sends to
receives msg from
sends message to
receives message from
interface
Topic
interface
Queue



Steps for Writing JMS Sender Application



Steps for Building a JMS Sender Application

1. Get ConnectionFactory and Destination object (Topic or Queue) through JNDI
2. Create a Connection
3. Create a Session to send/receive messages
4. Create a MessageProducer (TopicPublisher or QueueSender)
5. Start Connection
6. Send (publish) messages
7. Close Session and Connection

(1) Locate ConnectionFactory and Destination objects via JNDI

```
// Get JNDI InitialContext object
Context jndiContext =  new InitialContext();

// Locate ConnectionFactory object via JNDI
TopicConnectionFactory factory =
    (TopicConnectionFactory) jndiContext.lookup(
    "MyTopicConnectionFactory");

// Locate Destination object (Topic or Queue)
// through JNDI
Topic weatherTopic =
   (Topic) jndiContext.lookup("WeatherData");
```

(2) Create Connection Object

```
// Create a Connection object from
// ConnectionFactory object
TopicConnection topicConnection =
    factory.createTopicConnection();
```

3) Create a Session

```
// Create a Session from Connection object.
//      1st parameter controls transaction
//      2nd parameter specifies acknowledgment type
TopicSession session =
    topicConnection.createTopicSession (false,
                    Session.CLIENT_ACKNOWLEDGE);
```

4) Create Message Producer

```
// Create MessageProducer from Session object
//   TopicPublisher for Pub/Sub
//   QueueSender for Point-to-Point
TopicPublisher publisher =
            session.createPublisher(weatherTopic);
```

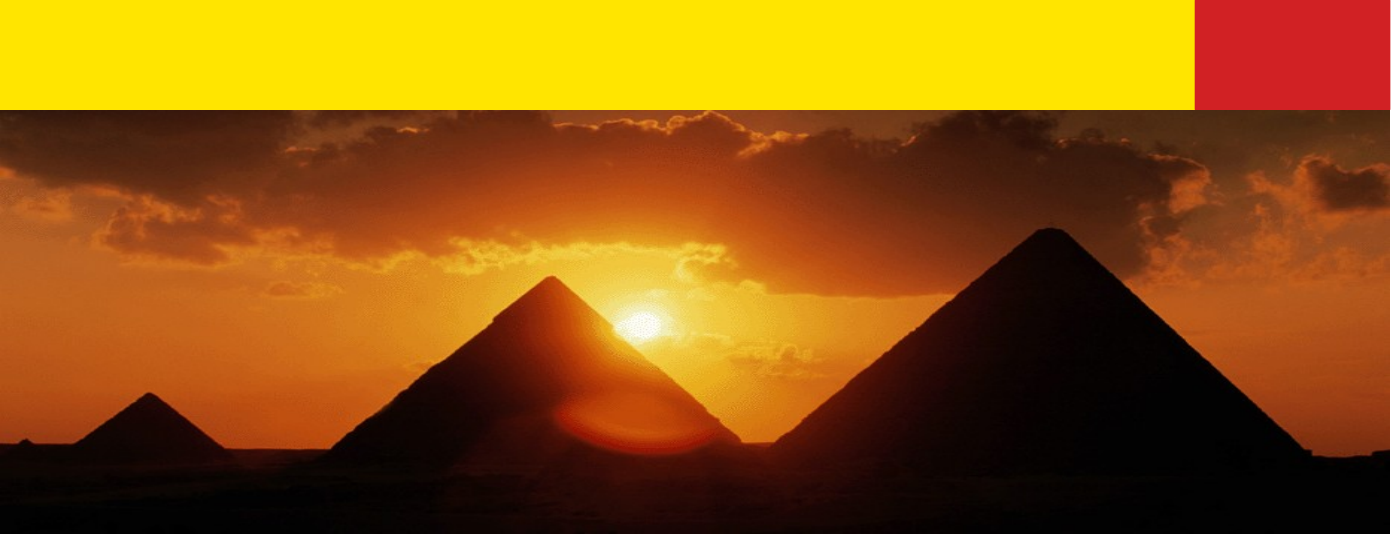
(6) Start Connection

```
// Until Connection gets started, message flow
// is inhibited: Connection must be started before
// messages will be transmitted.
topicConnection.start();
```

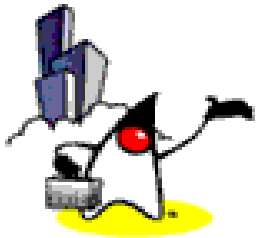
(6) Publish a Message

```
// Create a Message
TextMessage message =
        session.createMessage();
message.setText("text:35 degrees");

// Publish the message
publisher.publish(message);
```



Steps for Writing Non-blocking mode JMS Receiver Application



Steps for Building a JMS Receiver Application (non-blocking mode)

1. Get ConnectionFactory and Destination object (Topic or Queue) through JNDI
2. Create a Connection
3. Create a Session to send/receive messages
4. Create a MessageConsumer (TopicSubscriber or QueueReceiver)
5. Register MessageListener for non-blocking mode
6. Start Connection
7. Close Session and Connection

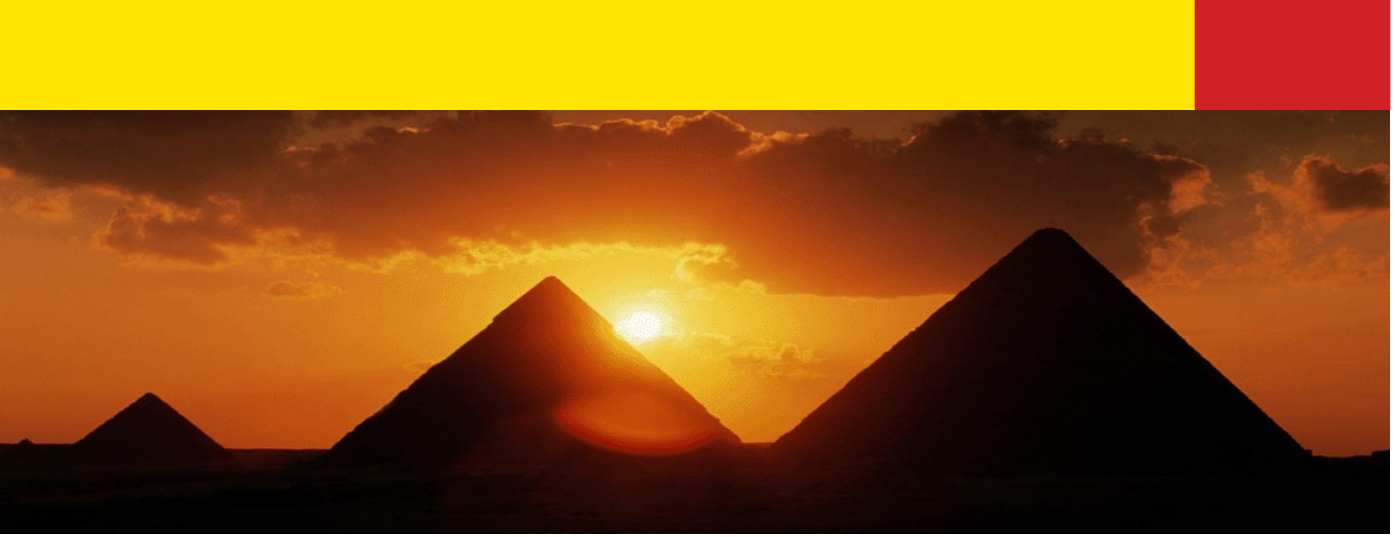
4) Create Message Subscriber

```
// Create Subscriber from Session object
TopicSubscriber subscriber =
            session.createSubscriber(weatherTopic);
```

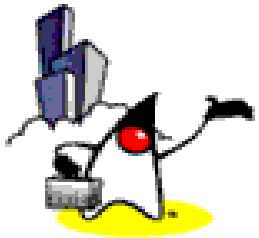
5) Register MessageListener object for non-blocking mode

```
// Create MessageListener object
WeatherListener myListener
            = new WeatherListener();

// Register MessageListener with
// TopicSubscriber object
subscriber.setMessageListener(myListener);
```

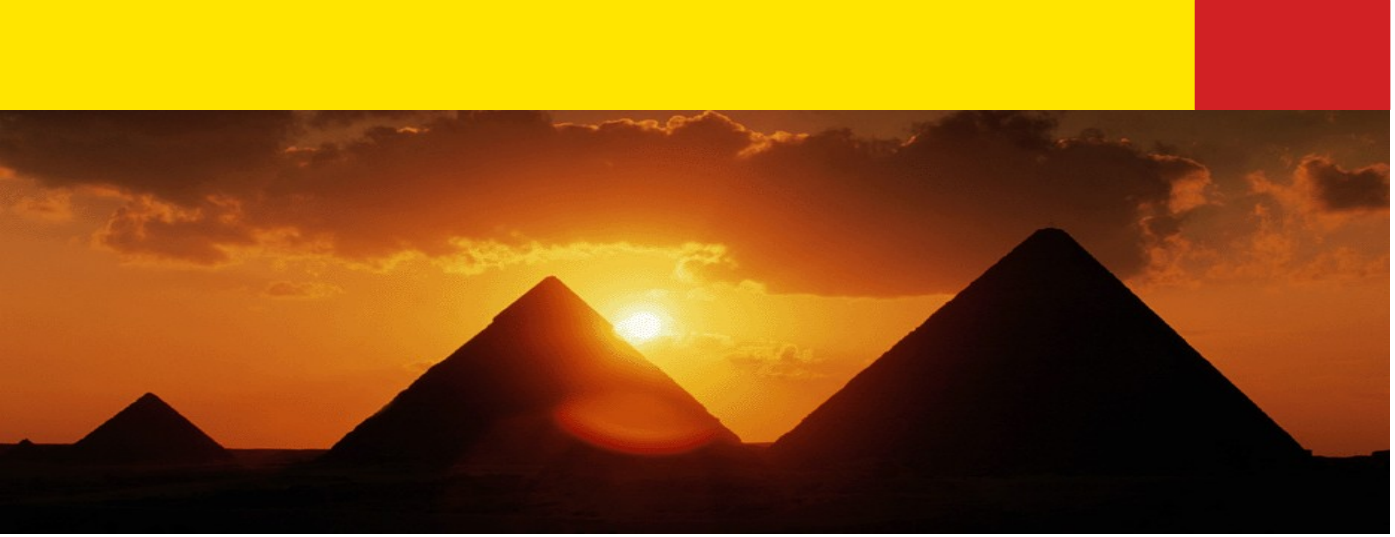


Steps for Writing blocking mode JMS Receiver Application

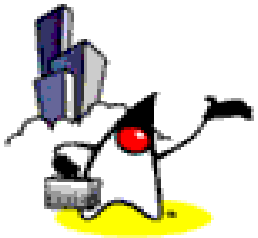


Steps for Building a JMS Receiver Application (non-blocking mode)

1. Get ConnectionFactory and Destination object (Topic or Queue) through JNDI
2. Create a Connection
3. Create a Session to send/receive messages
4. Create a MessageConsumer
5. Start Connection
6. Receive message
7. Close Session and Connection



How to Build Robust JMS Applications



Most Reliable Way

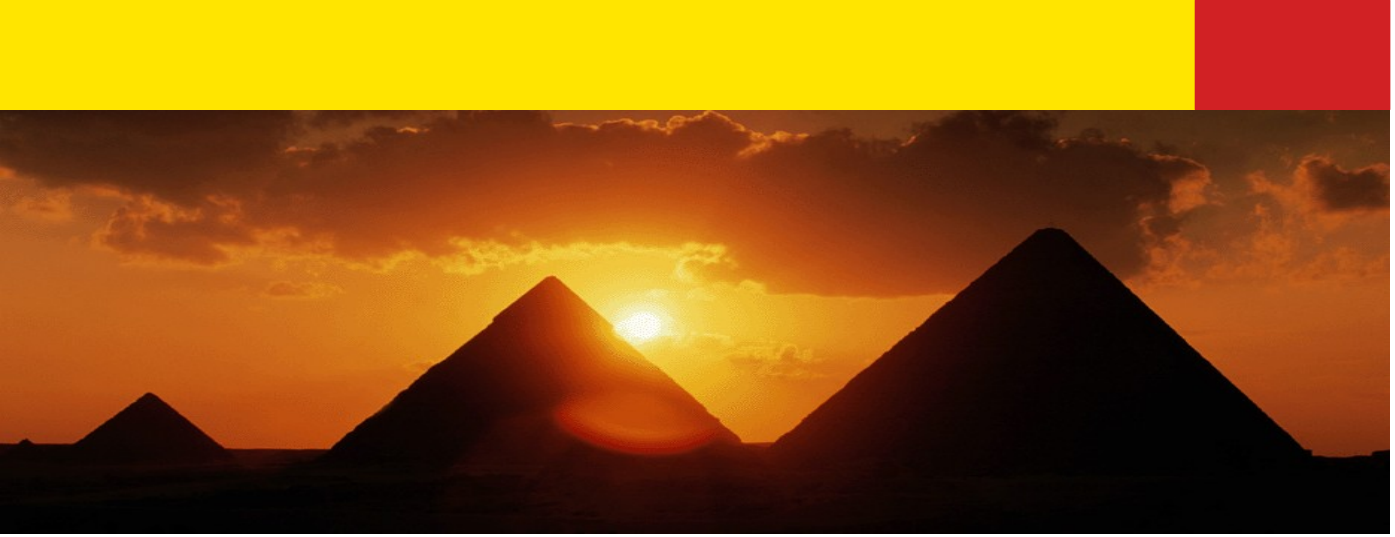
- The most reliable way to produce a message is to send a PERSISTENT message within a transaction.
- The most reliable way to consume a message is to do so within a transaction, either from a queue or from a durable subscription to a topic.

Basic Reliability Mechanisms

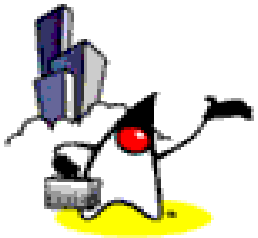
- Controlling message acknowledgment
- Specifying message persistence
- Setting message priority levels
- Allowing messages to expire
- Creating temporary destinations

Advanced JMS Reliability Mechanisms

- Creating durable subscriptions
- Using local transactions



Controlling Message Acknowledgment



Phases of Message Consumption

- The client receives the message
- The client processes the message
- The message is acknowledged
  - Acknowledgment is initiated either by the JMS provider or by the client, depending on the session acknowledgment mode

Transaction And Acknowledgment

- In transacted sessions
  - Acknowledgment happens automatically when a transaction is committed
  - If a transaction is rolled back, all consumed messages are redelivered
- In nontransacted sessions
  - When and how a message is acknowledged depend on the value specified (see next slide) as the second argument of the createSession method

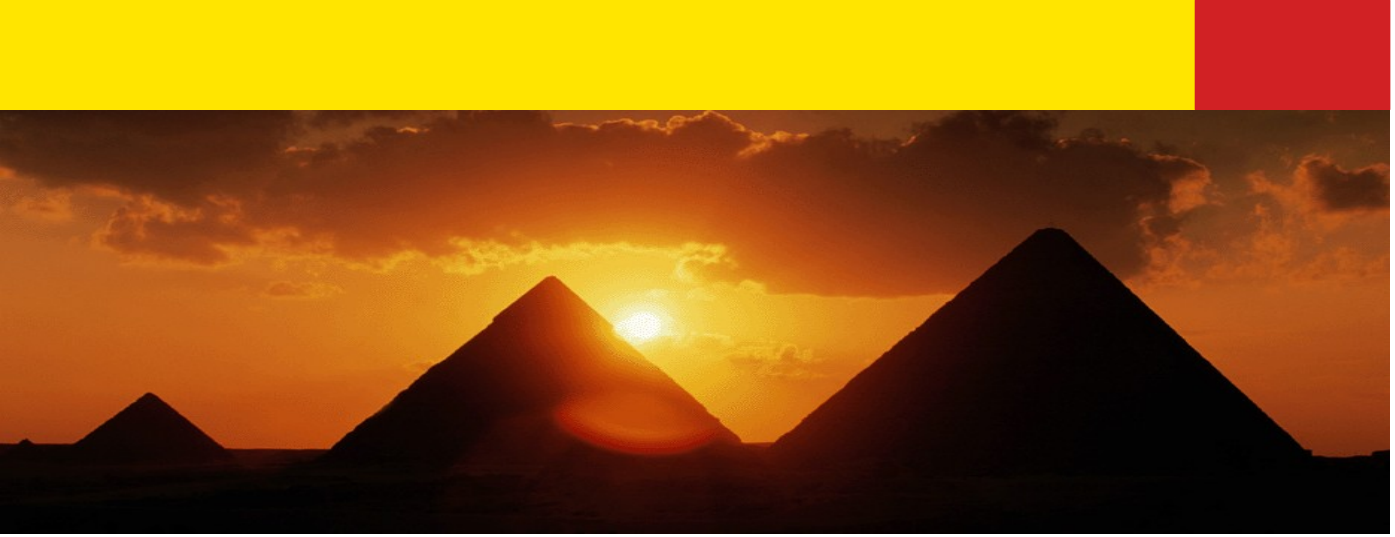
Acknowledgment Types

- Auto acknowledgment (AUTO_ACKNOWLEDGE)
  - Message is considered acknowledged when successful return on MessageConsumer.receive() or MessageListener.onMessage()
- Client acknowledgment (CLIENT_ACKKNOWLEDGE)
  - Client must call acknowledge() method of Message object
- Lazy acknowledgment (DUPS_OK_ACKNOWLEDGE)
  - Messaging system acknowledges messages as soon as they are available for consumers

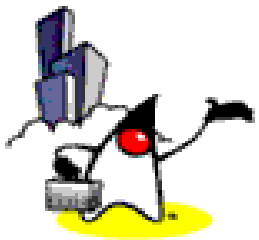
How Acknowledgment Type is set in JMS

- An acknowledge type is set when Session is created by setting appropriate flag
  - QueueConnection.createQueueSession(..,<flag >)
  - TopicConnection.createTopicSession(.., <flag>)
- Example

```
TopicSession session =
        topicConnection.createTopicSession (false,
                Session.CLIENT_ACKNOWLEDGE);
```



Specifying Message Persistence (Delivery Modes)

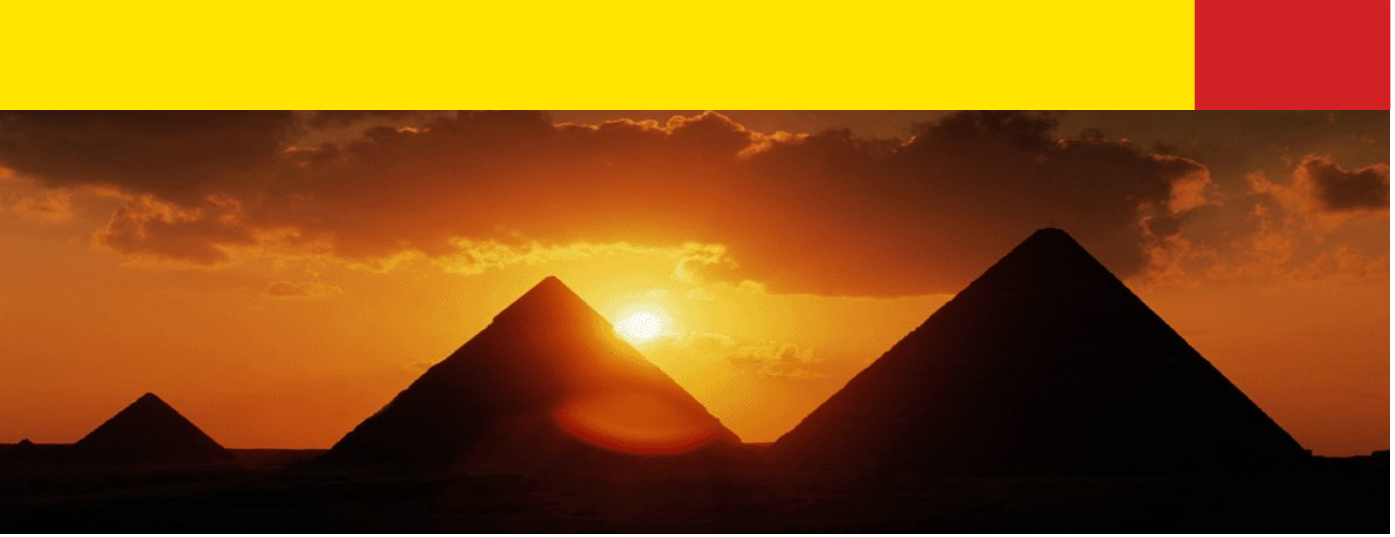


Two Delivery Modes

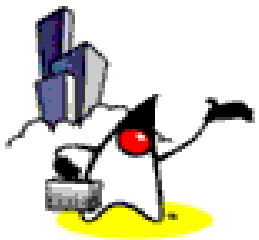
- PERSISTENT delivery mode
  - Default
  - Instructs the JMS provider to take extra care to ensure that a message is not lost in transit in case of a JMS provider failure
- NON_PERSISTENT delivery model
  - Does not require the JMS provider to store the message
  - Better performance

How to Specify Delivery Mode

- SetDeliveryMode method of the MessageProducer interface
  - producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
- Use the long form of the send or the publish method
  - producer.send(message, DeliveryMode.NON_PERSISTENT, 3,10000);

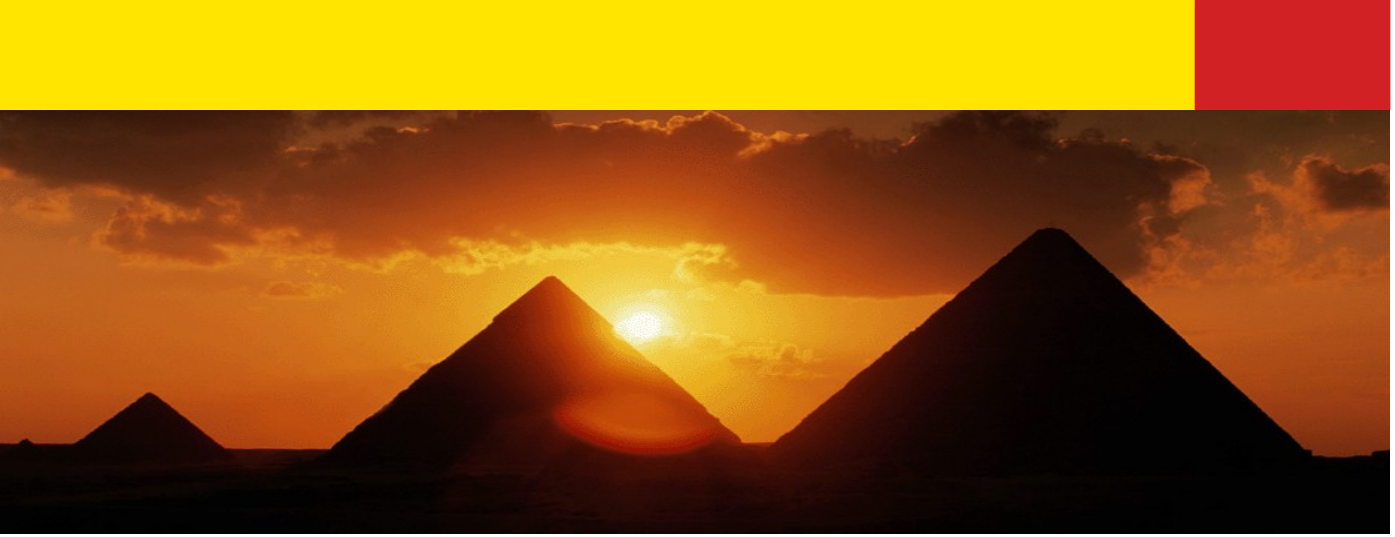


Setting Message Priority Levels

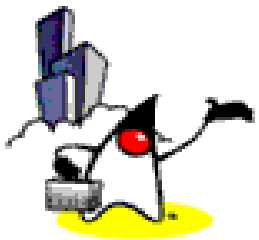


How to Set Delivery Priority

- Ten levels of priority range
  - from 0 (lowest) to 9 (highest)
  - Default is 4
- Use the setPriority method of the MessageProducer interface
  - producer.setPriority(7);
- Use the long form of the send or the publish method
  - producer.send(message, DeliveryMode.NON_PERSISTENT, 7, 10000);

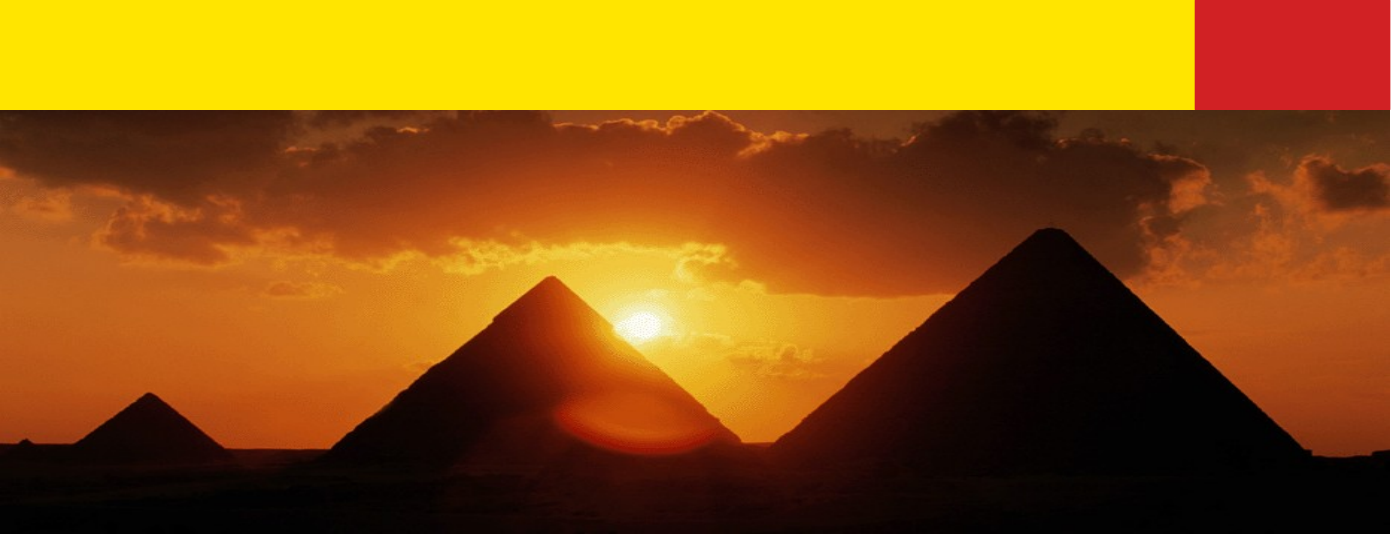


Allowing Messages to Expire

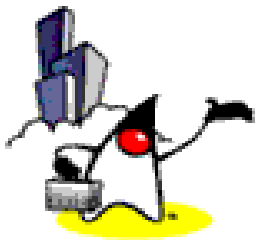


How to set Message Expiration

- Use the setTimeToLive method of the MessageProducer interface
  - producer.setTimeToLive(60000);
- Use the long form of the send or the publish method
  - producer.send(message, DeliveryMode.NON_PERSISTENT, 3, 60000);



Creating Durable Subscriptions

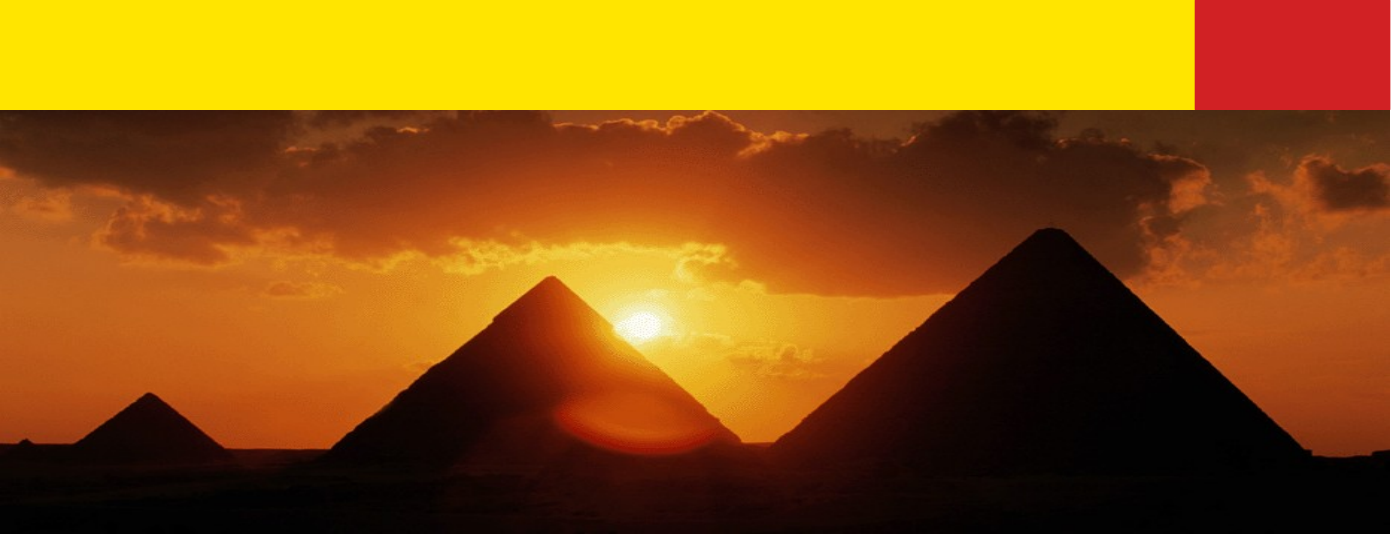


Maximum Reliability

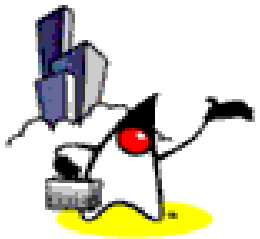
- To ensure that a pub/sub application receives all published messages
  - Use PERSISTENT delivery mode for the publishers
  - In addition, use durable subscriptions for the subscribers
    - Use the Session.createDurableSubscriber method to create a durable subscriber

How Durable Subscription Works

- A durable subscription can have only one active subscriber at a time
- A durable subscriber registers a durable subscription by specifying a unique identity that is retained by the JMS provider
- Subsequent subscriber objects that have the same identity resume the subscription in the state in which it was left by the preceding subscriber
- If a durable subscription has no active subscriber, the JMS provider retains the subscription's messages until they are received by the subscription or until they expire



Transactions in JMS



Transactions in JMS

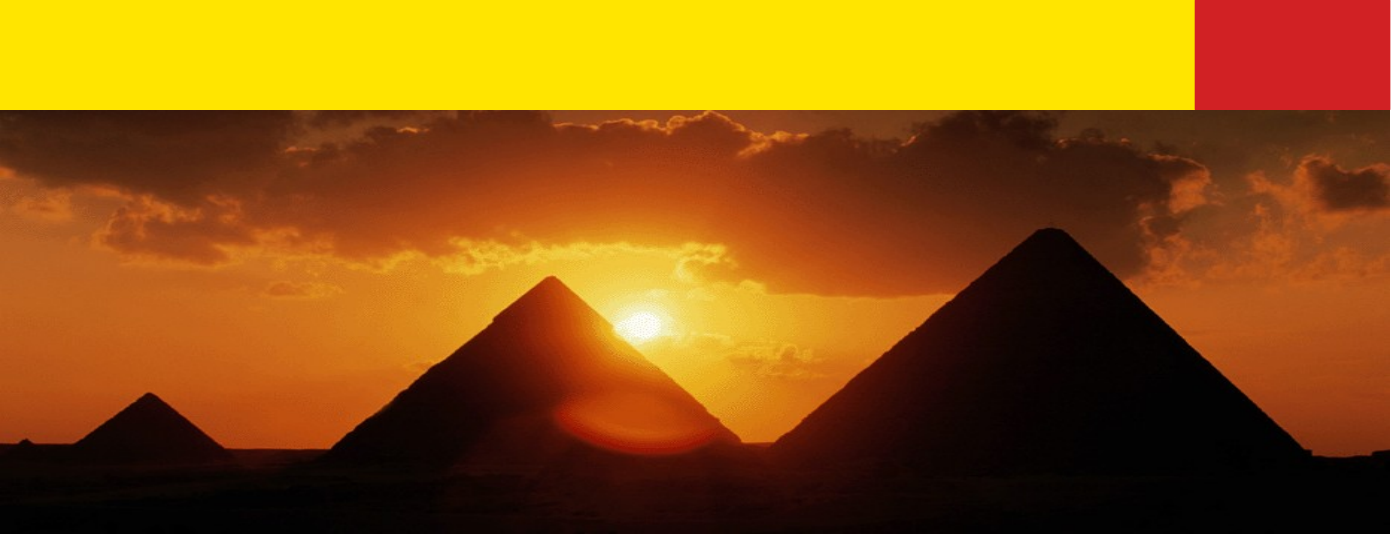
- Transaction scope is only between client and Messaging system not between clients
  - a group of messages are dispatched as a unit (on the sender side)
  - a group of messages are retrieved as a unit (on the receiver side)
- “Local” and “Distributed” transactions

Local Transactions in JMS

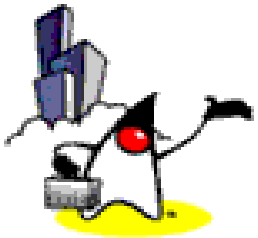
- Local transactions are controlled by Session object
- Transaction begins when a session is created
  - There is no explicit “begin transaction” method
- Transaction ends when Session.commit() or Session.abort() is called
- Transactional session is created by specifying appropriate flag when a session is created
  - QueueConnection.createQueueSession(true, ..)
  - TopicConnection.createTopicSession(true, ..)

Distributed Transactions in JMS

- Coordinated by a transactional manager
- Applications will control the transaction via JTA methods
  - Use of Session.commit() and Session.rollback() is forbidden
- Messaging operations can be combined with database transactions in a single transaction



Message Selector

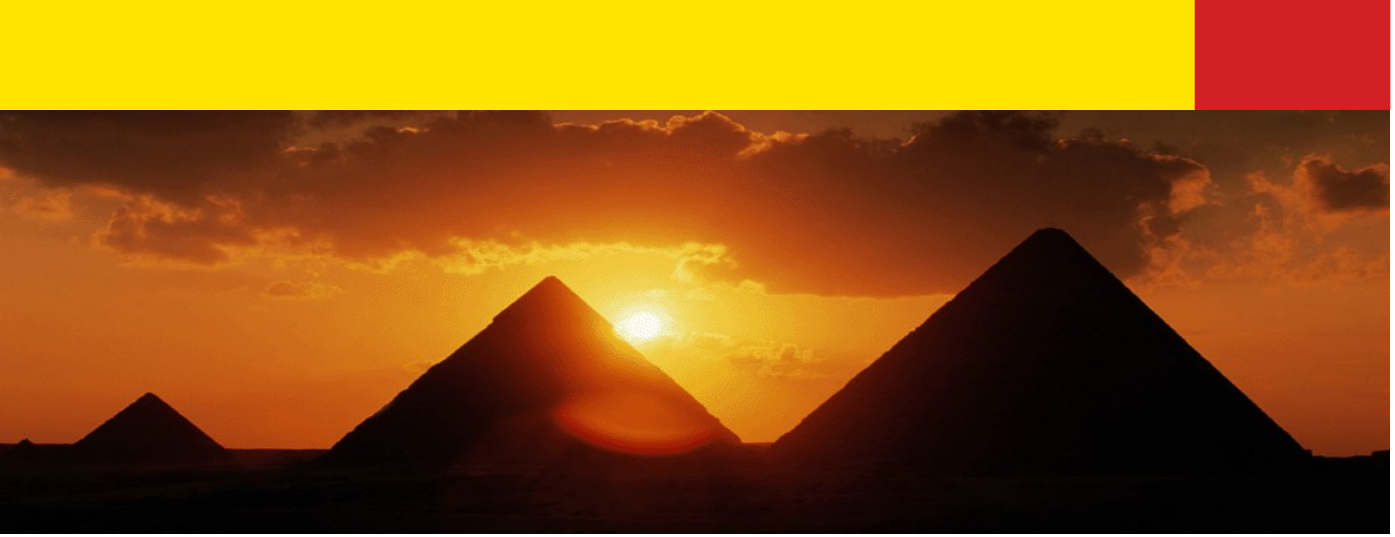


JMS Message Selector

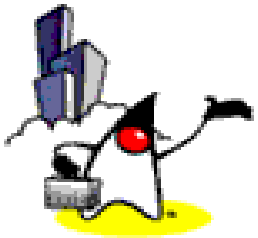
- (Receiver) JMS application uses a selector to select only the messages that are of interest
- A selector is essentially a SQL92 string that specifies “selection” (or filtering) rule
  - A Teller object listening for Account objects as messages may be required to do something only when the account balance drops below a $1000

Example: JMS Message Selectors

- The selector cannot reference the contents of a message
- It can access the properties and header
- Examples
  - JMSType=='wasp'
  - phone LIKE '223'
  - price BETWEEN 100 AND 200
  - name IN('sameer','tyagi')
  - JMSType IS NOT NULL

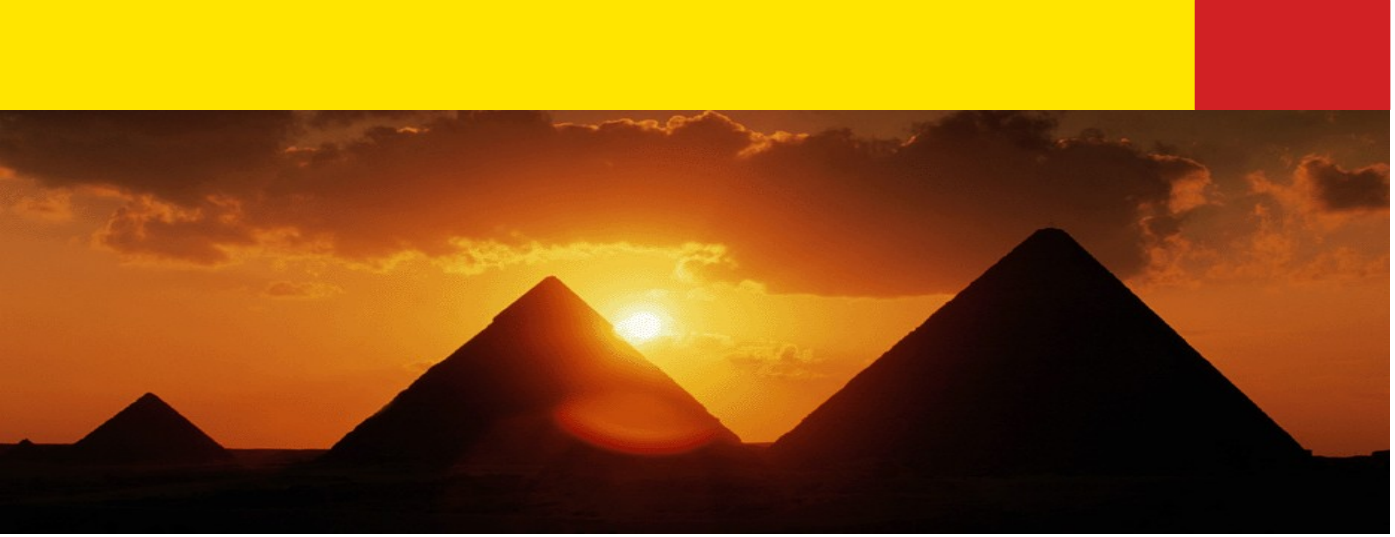


Messaging Features not Defined in JMS

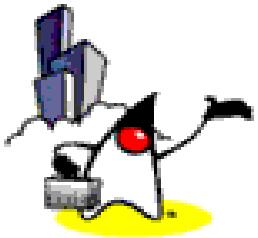


Messaging Features Not Defined in JMS (No APIs)

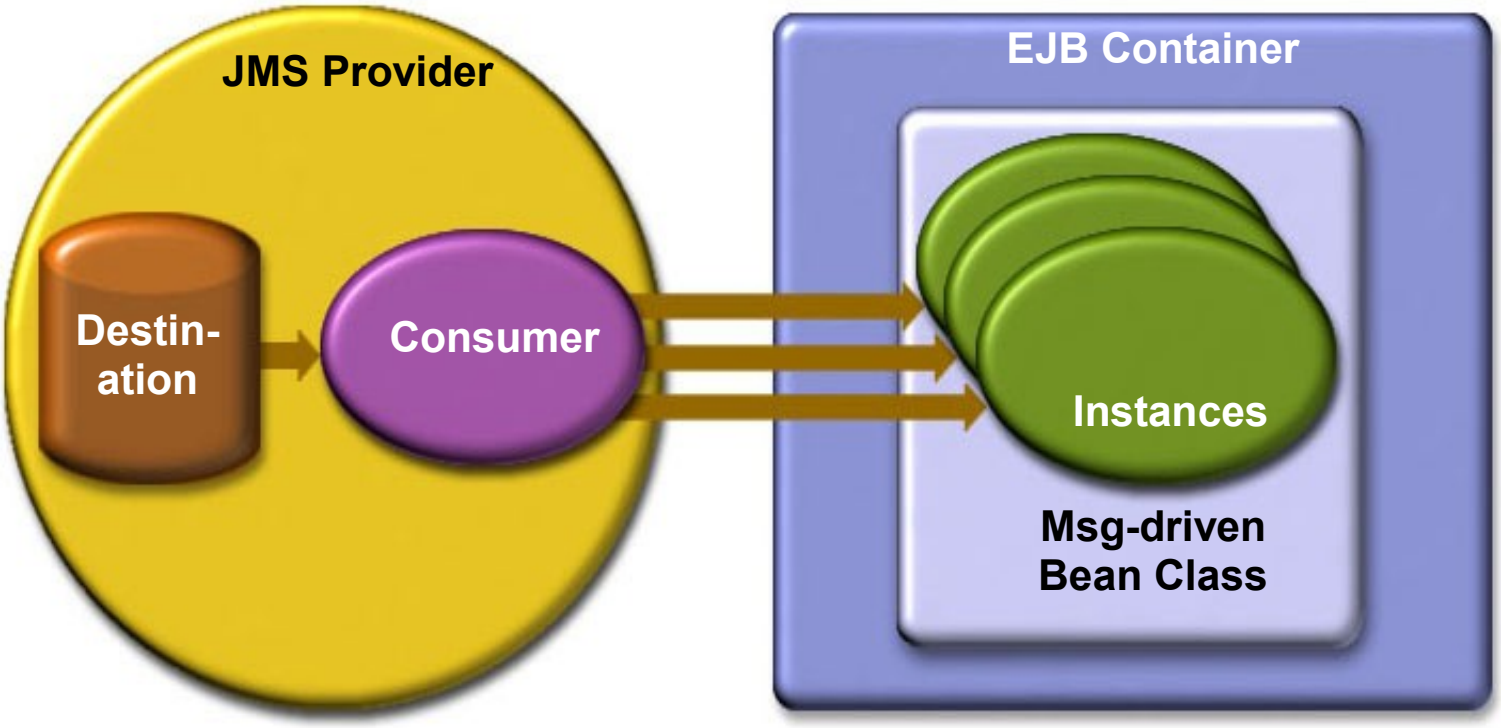
- Encryption
  - JMS spec assumes messaging system handles it
- Access control
  - JMS spec assumes messaging system handles it
- Load balancing
- Administration of queues and topics



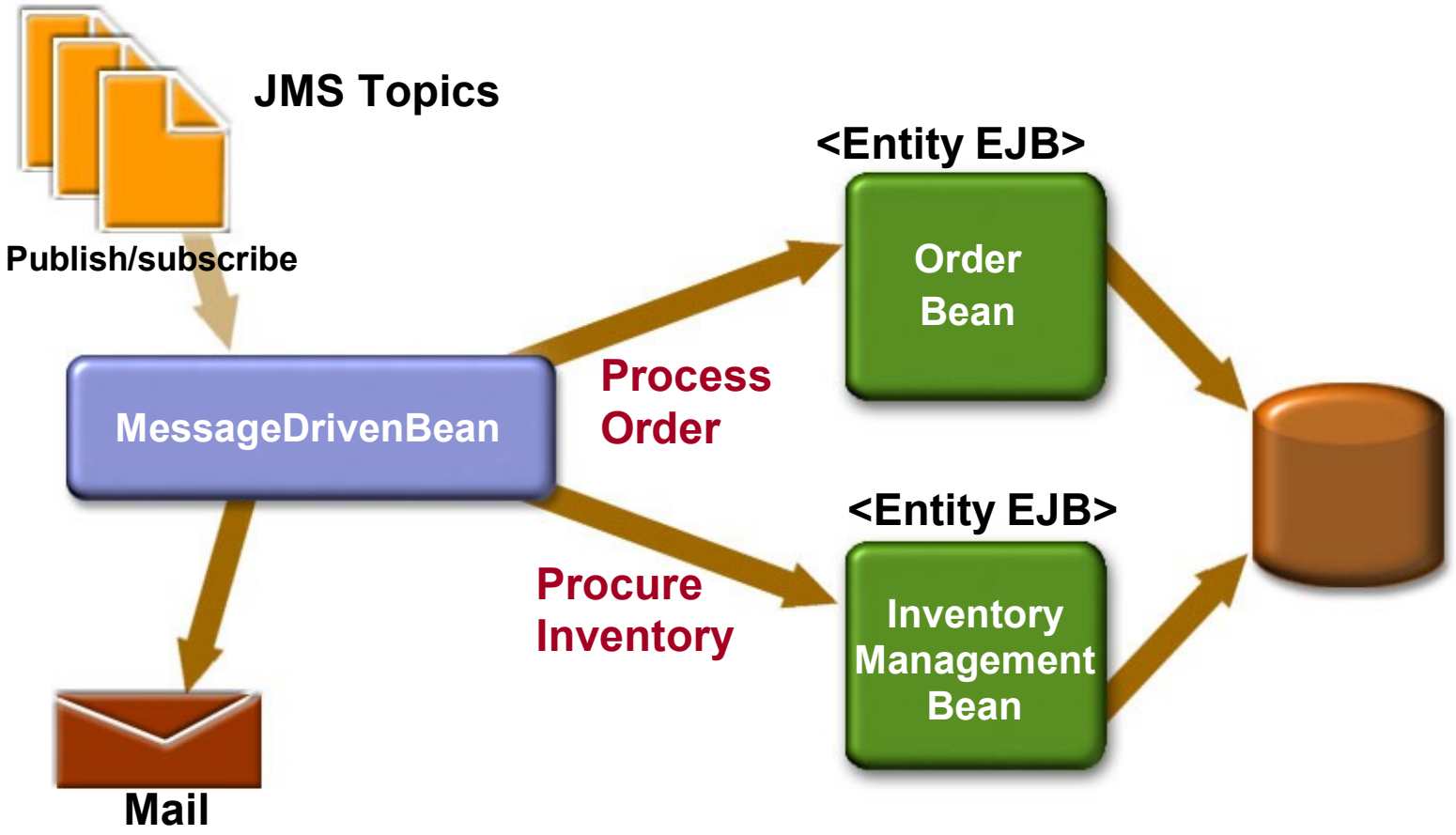
JMS and Message-Driven Bean (MDB)



JMS and MDB


JMS Provider
Destin-
ation
Consumer
EJB Container
Instances
Msg-driven
Bean Class

MDB Example


JMS Topics
Publish/subscribe
MessageDrivenBean
Process Order
Procure Inventory
Mail
<Entity EJB>
Order Bean
<Entity EJB>
Inventory Management Bean

Passion!