Java Message Service API

CSE 487/587
Feb 17, 2005

References:
http://java.sun.com/products/jsp/
JRun Programmer's Guide

Additional Notes

- Project 1 deadline extended to March 6
- Will cover JMS concepts today
- Practical example to be covered in the recitations next week
- Today's lecture should let you finalize your project design

Introduction

- The **Java Message Service API** allows applications to create, send, receive and read messages using reliable, asynchronous, loosely coupled communication
- It is part of the J2EE specification since J2EE ver 1.3. Hence, every implementation of J2EE (JRun, WebSphere) must support JMS.

ftp://ftp.oreilly.com/pub/conference/java2001/Hunter_et_al_jaxp.pdf

Introduction (contd.)

- **Loosely-coupled**: Sender and receiver need not be available at the same time. In fact they can be oblivious of each other. They only need to know the "destination" and the format of the message
- Different from RPC (tightly-coupled) and e-mail (humans)

JMS API Architecture

- **JMS Provider** is a messaging system that implements the JMS interfaces and provides admin and control features.

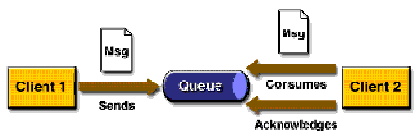
  JRun has its own built-in JMS Provider. It can also support external providers.
- **JMS Clients** are the programs or components written in Java that produce and consume messages. Any J2EE component can act as a JMS Client.

JMS API Architecture (contd.)

- **Messages** are the objects that communicate information between JMS Clients
- **Administered Objects** are preconfigured JMS objects created by an administrator for the use of clients.
  - **Destinations**
  - **Connection Factories**

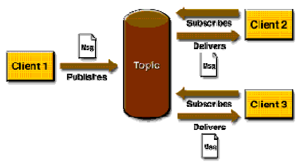
Messaging Domains

- **Point-to-Point Messaging (PTP)**
  - Each message is addressed to a specific queue, and receiving clients extract messages from these queues.
  - Queues retain all messages sent to them until the messages are consumed or until the messages expire.
  - Each message has only one consumer.
  - A sender and a receiver of a message have no timing dependencies. The receiver can fetch the message whether or not it was running when the client sent the message.
  - The receiver acknowledges the successful processing of a message.



Messaging Domains (contd.)

- **Publish-Subscribe (Pub-Sub)**
  - Clients address messages to a topic, which functions somewhat like a bulletin board.
  - The system takes care of distributing the messages from publishers to subscribers.
  - Publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has an active subscription to the topic.
  - Exception: Durable Subscriptions
  - Topics retain messages only as long as it takes to distribute them to current subscribers.
  - Each message can have zero or multiple consumers.



Message Consumption

- Receivers can consume messages in the following two ways (JMS is inherently asynchronous. Here, we use the following terms in a slightly different sense.):
  - **Synchronous**: A subscriber or a receiver explicitly fetches the message from the destination by calling the `receive` method.

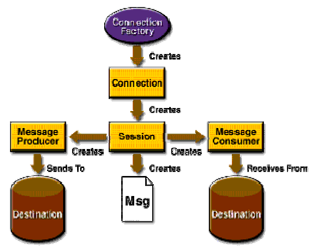
    The `receive` method can block until a message arrives or can time out if a message does not arrive within a specified time limit.
  - **Asynchronous**: A client can register a message listener with a consumer. A message listener is similar to an event listener.

    Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's `onMessage` method, which acts on the contents of the message.

    Message-driven beans use Asynchronous Message Consumption

ftp://ftp.oreilly.com/pub/conference/java2001/Hunter_et_al_jaxp.pdf

JMS API Programming Model



Objects involved in JMS Programming

JMS API Programming Model

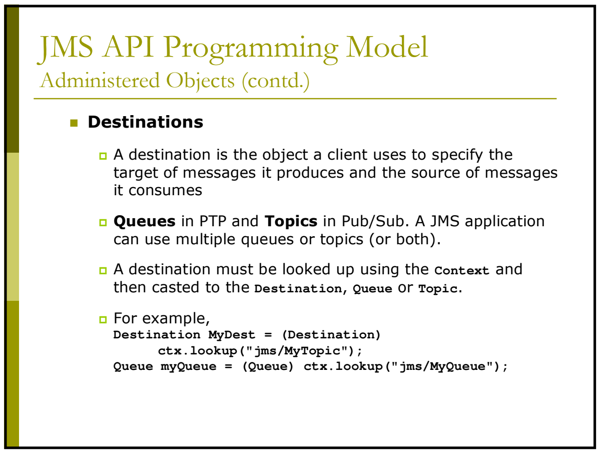
- **Administered Objects:** These objects may be implemented differently in every JMS implementation. Hence, they are set up administratively rather than through code, using admin tools provided by the JMS implementation.
- Of course, these objects implement interfaces that are specified by the JMS (J2EE) specification.
- Two objects: **`ConnectionFactory`** & **`Destination`**

JMS API Programming Model
Administered Objects (contd.)

- **Connection Factory**
  - A connection factory is the object a client uses to create a connection to a provider.
  - Each connection factory is an instance of the **`ConnectionFactory`**, **`QueueConnectionFactory`**, or **`TopicConnectionFactory`** interface.
  - At the beginning of a JMS client program, you usually perform a JNDI lookup of a connection factory, then cast and assign it to a **`ConnectionFactory`** object.
  - For example,

```
Context ctx = new InitialContext();
ConnectionFactory cFactory = (ConnectionFactory)
     ctx.lookup("jms/ConnectionFactory");
```

ftp://ftp.oreilly.com/pub/conference/java2001/Hunter_et_al_jaxp.pdf

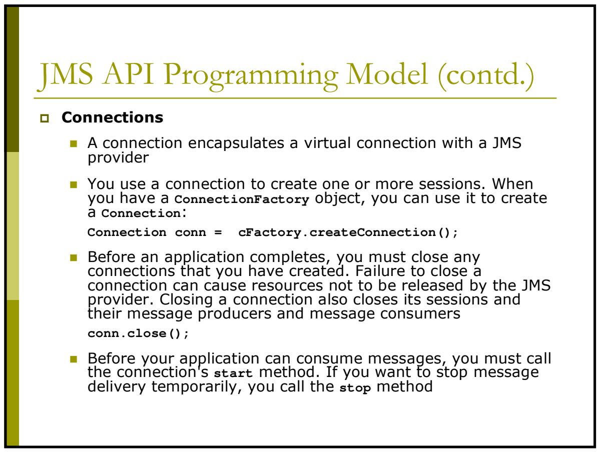


JMS API Programming Model

Administered Objects (contd.)

- **Destinations**
  - A destination is the object a client uses to specify the target of messages it produces and the source of messages it consumes
  - **Queues** in PTP and **Topics** in Pub/Sub. A JMS application can use multiple queues or topics (or both).
  - A destination must be looked up using the `Context` and then casted to the `Destination`, `Queue` or `Topic`.
  - For example,

```
Destination MyDest = (Destination)
      ctx.lookup("jms/MyTopic");
Queue myQueue = (Queue) ctx.lookup("jms/MyQueue");
```



JMS API Programming Model (contd.)

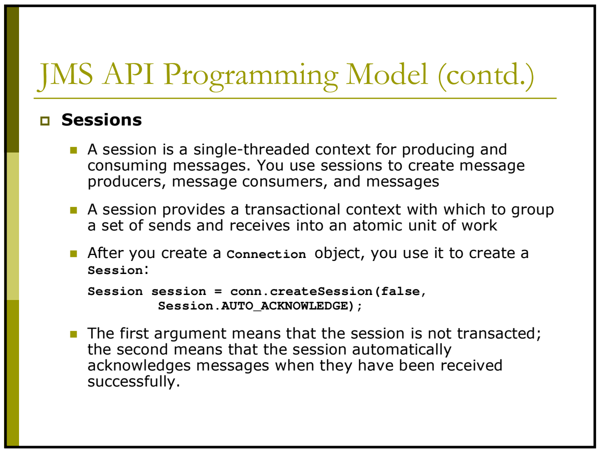
- **Connections**
  - A connection encapsulates a virtual connection with a JMS provider
  - You use a connection to create one or more sessions. When you have a `ConnectionFactory` object, you can use it to create a `Connection`:

```
Connection conn =  cFactory.createConnection();
```

  - Before an application completes, you must close any connections that you have created. Failure to close a connection can cause resources not to be released by the JMS provider. Closing a connection also closes its sessions and their message producers and message consumers

```
conn.close();
```

  - Before your application can consume messages, you must call the connection's `start` method. If you want to stop message delivery temporarily, you call the `stop` method



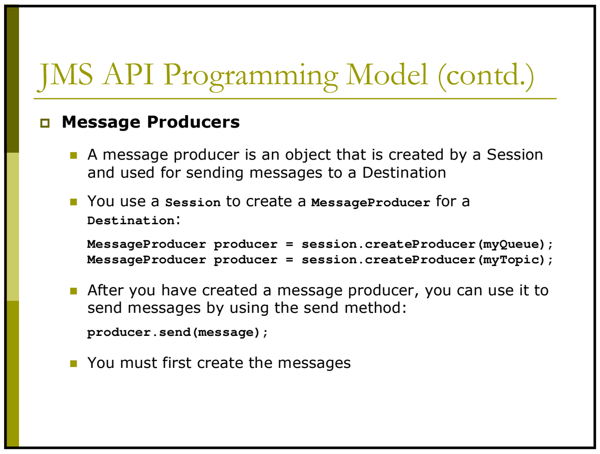
JMS API Programming Model (contd.)

- **Sessions**
  - A session is a single-threaded context for producing and consuming messages. You use sessions to create message producers, message consumers, and messages
  - A session provides a transactional context with which to group a set of sends and receives into an atomic unit of work
  - After you create a `Connection` object, you use it to create a `Session`:

```
Session session = conn.createSession(false,
         Session.AUTO_ACKNOWLEDGE);
```

  - The first argument means that the session is not transacted; the second means that the session automatically acknowledges messages when they have been received successfully.

ftp://ftp.oreilly.com/pub/conference/java2001/Hunter_et_al_jaxp.pdf



JMS API Programming Model (contd.)

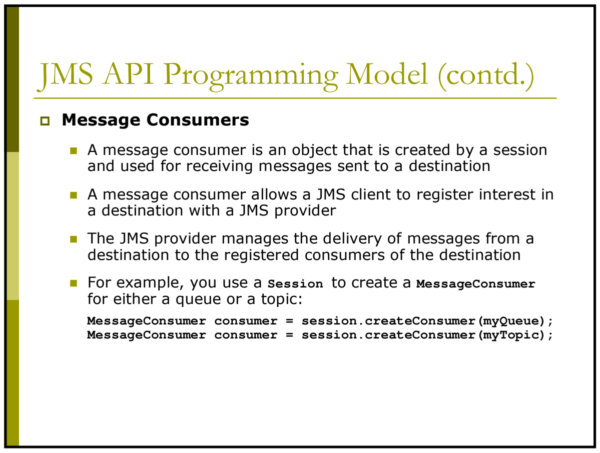
- **Message Producers**
  - A message producer is an object that is created by a Session and used for sending messages to a Destination
  - You use a `Session` to create a `MessageProducer` for a `Destination`:

```
MessageProducer producer = session.createProducer(myQueue);
MessageProducer producer = session.createProducer(myTopic);
```

  - After you have created a message producer, you can use it to send messages by using the send method:

```
producer.send(message);
```

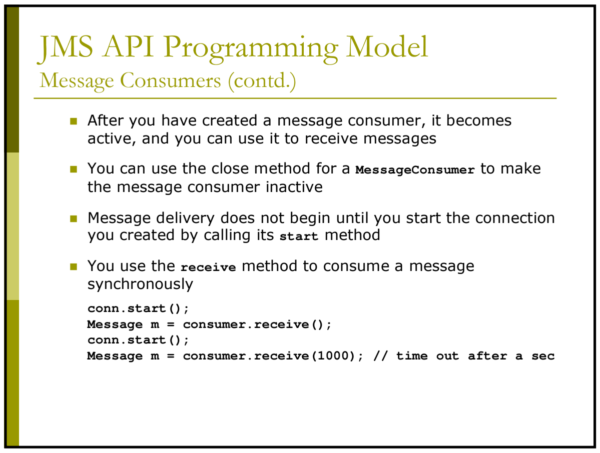
  - You must first create the messages



JMS API Programming Model (contd.)

- **Message Consumers**
  - A message consumer is an object that is created by a session and used for receiving messages sent to a destination
  - A message consumer allows a JMS client to register interest in a destination with a JMS provider
  - The JMS provider manages the delivery of messages from a destination to the registered consumers of the destination
  - For example, you use a `Session` to create a `MessageConsumer` for either a queue or a topic:

```
MessageConsumer consumer = session.createConsumer(myQueue);
MessageConsumer consumer = session.createConsumer(myTopic);
```



JMS API Programming Model
Message Consumers (contd.)

- After you have created a message consumer, it becomes active, and you can use it to receive messages
- You can use the close method for a `MessageConsumer` to make the message consumer inactive
- Message delivery does not begin until you start the connection you created by calling its `start` method
- You use the `receive` method to consume a message synchronously

```
conn.start();
Message m = consumer.receive();
conn.start();
Message m = consumer.receive(1000); // time out after a sec
```

ftp://ftp.oreilly.com/pub/conference/java2001/Hunter_et_al_jaxp.pdf

JMS API Programming Model (contd.)

- **Message Listeners (Asyncronous Message Consumption)**
  - A message listener is an object that acts as an asynchronous event handler for messages
  - This object implements the `MessageListener` interface, which contains one method, `onMessage`
  - In the `onMessage` method, you define the actions to be taken when a message arrives
  - The `onMessage` method takes one argument of type `Message`, which your implementation of the method can cast to any of the other message types
  - You register the message listener with a specific `MessageConsumer` by using the `setMessageListener` method

JMS API Programming Model

Message Listeners (contd.)

- For example,

```
Listener myListener = new Listener();
consumer.setMessageListener(myListener);
```

- After you register the message listener, you call the `start` method on the `Connection` to begin message delivery
- The `onMessage` method is called by the JMS Provider when a message is delivered
- Your `onMessage` method should handle all exceptions. It must not throw checked exceptions
- At any time, only one of the session's message listeners is running

JMS API Programming Model (contd.)

- **Message Selectors**
  - If your messaging application needs to filter the messages it receives, you can use a JMS API message selector, which allows a message consumer to specify the messages it is interested in
  - Message selectors assign the work of filtering messages to the JMS provider rather than to the application
  - A message selector is a `String` that contains an expression
  - The `createConsumer` method allows you to specify a message selector as an argument when you create a message consumer
  - The message consumer then receives only messages whose headers and properties match the selector. A message selector cannot select messages on the basis of the content of the message body.

JMS API Programming Model (contd.)

- **Messages**
  - The ultimate purpose of a JMS application is to produce and to consume messages that can then be used by other software applications
  - JMS messages have a basic format that is simple but highly flexible
  - A JMS message has three parts: a header, properties, and a body

JMS API Programming Model

Messages (contd.)

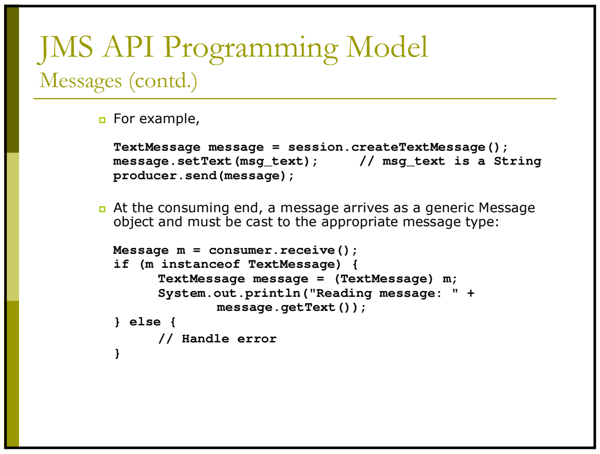
- **Message Headers**
  - A JMS message header contains a number of predefined fields that contain values that both clients and providers use to identify and to route messages
  - Each header field has associated setter and getter methods
  - Some header fields are intended to be set by a client, but many are set automatically by the send or the publish method, which overrides any client-set values.

Header Field	Set By
`JMSDestination`	`send` or `publish` method
`JMSDeliveryMode`	`send` or `publish` method
`JMSExpiration`	`send` or `publish` method
`JMSPriority`	`send` or `publish` method
`JMSMessageID`	`send` or `publish` method
`JMSTimestamp`	`send` or `publish` method
`JMSCorrelationID`	Client
`JMSReplyTo`	Client
`JMSType`	Client
`JMSRedelivered`	JMS provider

JMS API Programming Model

Messages (contd.)

- **Message Properties**
  - You can create and set properties for messages if you need values in addition to those provided by the header fields. You can use properties to provide compatibility with other messaging systems, or you can use them to create message selectors
  - The JMS API provides some predefined property names that a provider can support. The use either of these predefined properties or of user-defined properties is optional
- **Message Bodies**
  - The JMS API defines five message body formats, also called message types
  - The JMS API provides methods for creating messages of each type and for filling in their contents.



JMS API Programming Model

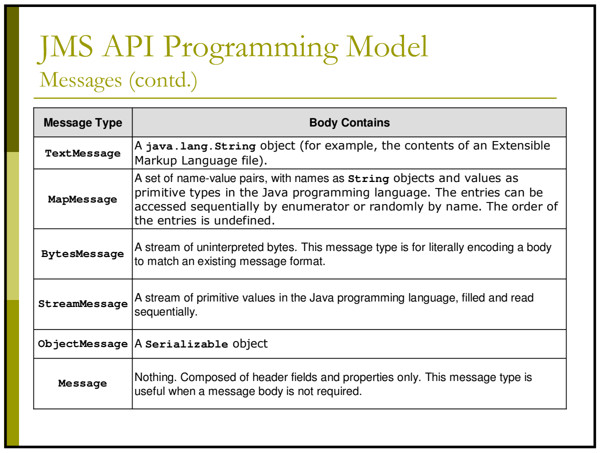
Messages (contd.)

- For example,

```
TextMessage message = session.createTextMessage();
message.setText(msg_text);     // msg_text is a String
producer.send(message);
```

- At the consuming end, a message arrives as a generic Message object and must be cast to the appropriate message type:

```
Message m = consumer.receive();
if (m instanceof TextMessage) {
      TextMessage message = (TextMessage) m;
      System.out.println("Reading message: " +
              message.getText());
} else {
      // Handle error
}
```



JMS API Programming Model

Messages (contd.)

Message Type	Body Contains
`TextMessage`	A `java.lang.String` object (for example, the contents of an Extensible Markup Language file).
`MapMessage`	A set of name-value pairs, with names as `String` objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.
`BytesMessage`	A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.
`StreamMessage`	A stream of primitive values in the Java programming language, filled and read sequentially.
`ObjectMessage`	A `Serializable` object
`Message`	Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required.