ID2212 Network Programming with Java
Lecture 8

# Message-Based Communication, Java Message Service (JMS) API, JavaEmail API. Java Naming and Directory Interface (JNDI).

Vladimir Vlassov and Leif Lindbäck
KTH/ICT/SCS
HT 2015

# Message-Oriented Middleware and The Java Message Service API (JMS)

**`javax.jms`**

home page:
http://www.oracle.com/technetwork/java/jms/index.html

# Message-Oriented Middleware, MOM

- Enables the exchange of general-purpose messages in a distributed application.
- Data is exchanged by message queuing, either synchronously or asynchronously.
- Reliable message delivery is achieved using message queues, and by providing security, transactions and the required administrative services.

# Difference between MOM and RPC/RMI

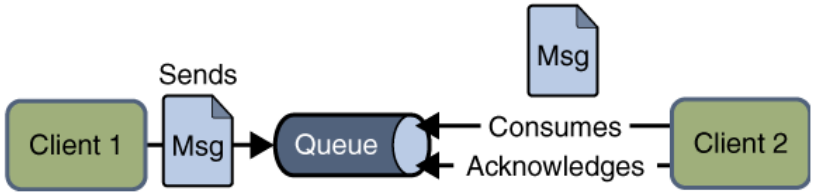
- When using RPC or RMI, the server must be available to accept calls. If the server is down, the call can not be made.
- When using MOM, messages can be sent to servers that are down.
  - Messages under a MOM system are placed into a queue and retrieved whenever the server requests them.
  - Whether the server is available at the time the message is sent is irrelevant.
- Senders call the MOM, instead of calling the server directly.
- That way, applications can be relieved of non-functional requirements, like interoperability, reliability, security, scalability, performance, etc.
  - It is up to the MOM (and its administrator) to handle that.

# The Java Message Service API (JMS)

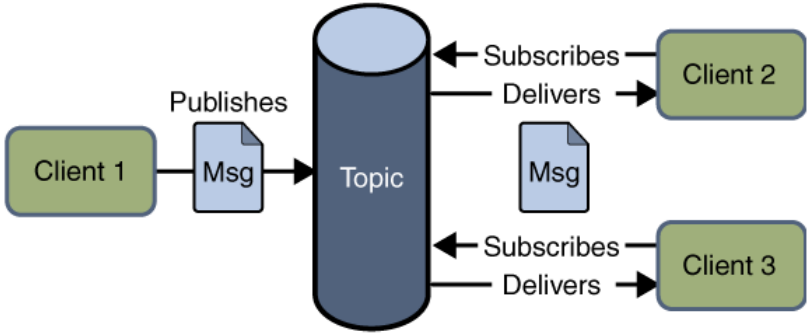
- JMS provides a Java API for an existing message queue. The JMS specification defines how to call the provider, it does not include a provider.
- ***Synchronous and Asynchronous message production*** (send)
- ***Synchronous message consumption*** (receive)
- ***Asynchronous message consumption*** by a message listener registered as consumer.
  - Message-driven EJBs asynchronously consume messages.
- ***Reliable messaging***: Can ensure that a message is delivered once and only once.
- ***JMS provider*** is a messaging agent performing messaging

# Two Messaging Domains

- **Queues**: Point-to-Point (PTP) Messaging Domain



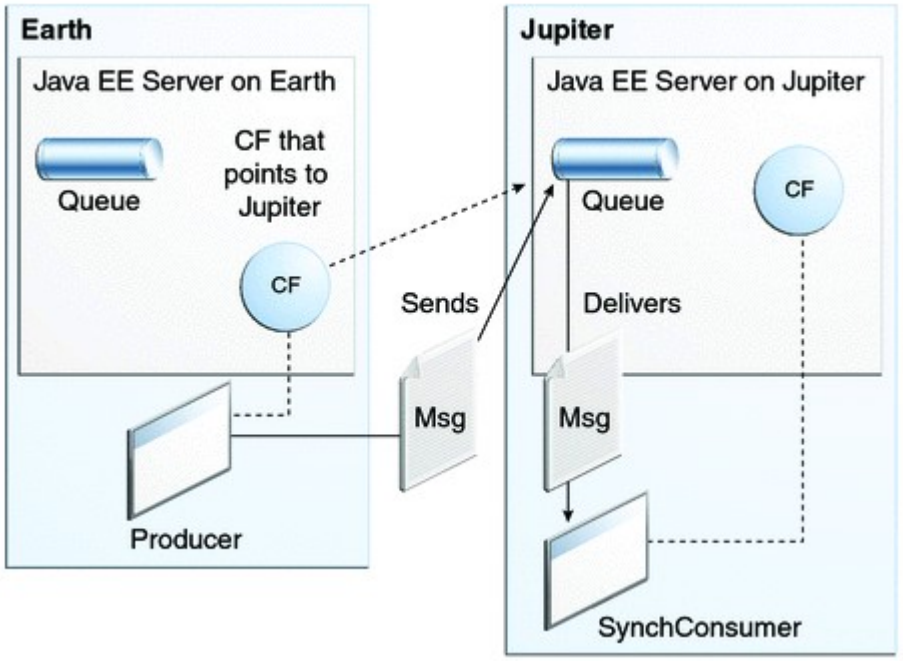

- **Topics**: Publish/Subscribe (pub/sub) Messaging Domain




- A stand-alone JMS provider can implement one or both domains.
- A Java EE provider must implement both domains.

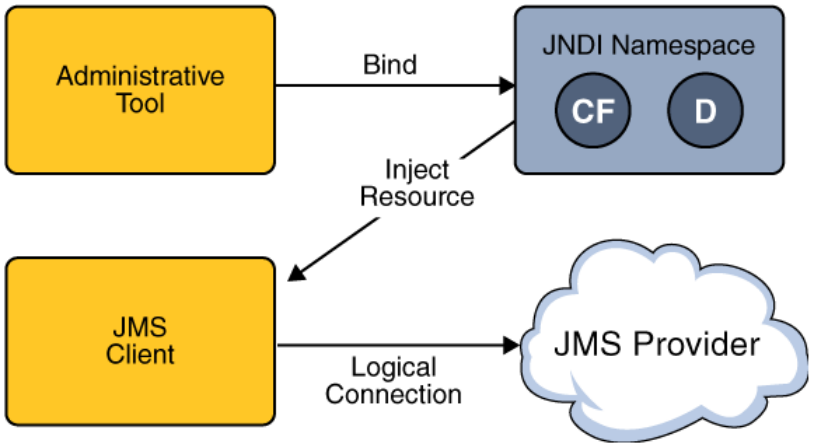
# Clients on Different Systems

- Clients can communicate with each other, when running on different systems in a network.
  - The systems must be visible to each other by name (IP address) and must have compatible message queues.
  - **Configuration issue**



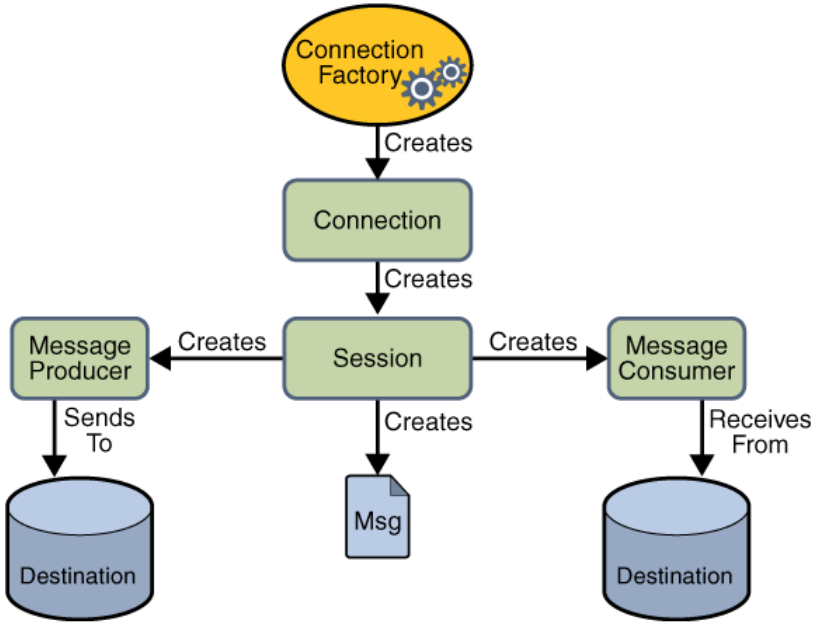
# JMS Architecture

- A JMS application is composed of:
- ***A JMS provider***
  - There are many message queues that can be used as JMS provider, e.g., Apache ActiveMQ, RabbitMQ and IBM WebSphere MQ. The GlassFish server also includes a JMS provider.
- ***JMS clients***
  - producing and/or consuming ***messages***.
- ***Messages***
  - objects that communicate information between JMS clients.
- ***Administered objects***
  - ***Destinations (D)***;
  - ***Connection Factories (CF)*** described in Administered Objects
  - created by an administrator for the use of clients



# JMS Programming Concepts

- Administered Objects
  - Connection Factory
  - Destinations (queues, topics, both)
- Connection
- Session
- Message Producers
- Message Consumers
  - Message consumers
  - Message listeners
  - Message selectors
- Messages
  - Headers, properties, bodies
- Queue Browsers




- Steps:
  - Creating a connection and a session
  - Creating message producers and consumers
  - Sending and receiving messages

# ConnectionFactory

- An administered object, deployed to the server by the message queue administrator.
- Encapsulates a set of connection configuration parameters, defined by the administrator.
- Used by a JMS client to create a connection with a JMS provider.
- When used in a Java EE server, the connection factory object is created and injected by the server:

```
@Resource(mappedName="jms/MyConnectionFactory")
private static ConnectionFactory connectionFactory;
```

# Destination

- An administered object, deployed to the server by the message queue administrator.
- Encapsulates a provider-specific address.
- Used by a client to specify the target of messages it produces and the source of messages it consumes.
- When used in a Java EE server, the connection factory object is created and injected by the server:

```
@Resource(mappedName="jms/MyQueue")
private static Queue queue;

@Resource(mappedName="jms/MyTopic")
private static Topic topic;
```

# Connection

- Encapsulates an open connection with a JMS provider.
- Typically represents an open TCP/IP socket between a client and the service provider.
- Created by a `ConnectionFactory`:

```
Connection connection =
  connectionFactory.createConnection();
...
connection.close();
```

# Session

- A single-threaded context for producing and consuming messages.
- Used to create message producers and consumers, messages, queue browsers, temporary queues and topics.
- Retains messages it consumes until they have been acknowledged.
- A not transacted session with automatic acknowledgement of messages:

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

- A transacted session, messages are acknowledged on commit:

```
Session session = connection.createSession( true, 0);
```

# MessageProducer

A message producer is created by a session, and used for sending messages to a destination.

- Create a producer for a `Destination` object (`Queue` or `Topic`):

```
MessageProducer producer =
session.createProducer(destination);
```

- Send messages by using the send method:

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

- Create an unidentified producer and specify a destination when sending a message:

```
MessageProducer producer =
session.createProducer(null);
producer.send(destination, message);
```

# MessageConsumer

- A message consumer is created by a session and used for receiving messages sent to a destination.
- Create a consumer for a `Destination` object (`Queue` or `Topic`):

```
MessageConsumer consumer =
                session.createConsumer(dest);
```

- Start the connection and use the receive method to consume a message synchronously.

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

# MessageListener

- A message listener acts as an asynchronous event handler for messages.
  - Implements the `MessageListener` interface, wich has one method, `onMessage`.

```
public void onMessage(Message message);
```

- Register the message listener with a specific MessageConsumer

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

# Messages

- A JMS message has three parts:
  1. (required) a header,
  2. (optional) properties,
  3. (optional) a body.
- A ***header*** contains predefined fields with values that both clients and providers use to identify and to route messages.

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

# Message Body Types

- Five message body formats (a.k.a. message types)

```
TextMessage message =
  session.createTextMessage();
message.setText(msg_text);
producer.send(message);
...
Message m = consumer.receive();
if (m instanceof TextMessage) {
   TextMessage message =
    (TextMessage) m;
   System.out.println("Message:"
     + message.getText());
} else {
   // Handle error
}
```

| Message Type | Contents |
|---|---|
| `TextMessage` | A `String` object (for example, the contents of an XML file). |
| `MapMessage` | A set of name-value pairs, names as `String` and values as primitive types. Entries can be accessed sequentially by enumerator or randomly by name. |
| `BytesMessage` | A stream of bytes. |
| `StreamMessage` | A stream of primitive values. |
| `ObjectMessage` | A `Serializable` object. |
| `Message` | Nothing, but header fields and properties only. |

# JavaMail API

`javax.mail`

home page:

http://www.oracle.com/technetwork/java/javamail/index.html

# JavaMail Programming Concepts

- `Session`, a basic email session
- `Message`, an email
- `Address`, an email address of a recipient or a sender
- `Transport`, a facility used to connect to the mail server and to send a message
- `Store`, an email store
- `Folder`, an email folder
- `Authenticator`, knows how to obtain authentication for a network connection. Usually, by prompting the user.

# Session

- A basic mail session
- An object of the `Session` class
- For example:

```
Properties props = new Properties();
// Fill props with any information, e.g. mail server,
// protocol, username
Session session = Session.getDefaultInstance(props,
                                             Authenticator);
// The authenticator object will be called to retrieve the
// user's credentials, for example password.
```

# `Authenticator`

- An object to access to the mail server using a username and password.
- Develop a subclass of `Authenticator` that is used to create a `PasswordAuthentication` object when authentication is necessary.
- Instantiate the `Authenticator` subclass and pass it to the `Session` object.

# Message

- An email message to be sent
- An object of a `Message` subclass
  - such as `javax.mail.internet.MimeMessage` – a email message that understands MIME types and headers
- For example:

```
MimeMessage message = new MimeMessage(session);
message.setText("Hello");
message.setSubject("First");
```

# Address

- An email address of a recipient or a sender – an object of the `javax.mail.internet.InternetAddress` class
- For example:

```
Address fromAddress =
    new InternetAddress( “vlad@kth.se”, “Vladimir Vlassov”);
Address toAddress =
    new InternetAddress(“leifl@kth.se”);
Address ccAddress =
    new InternetAddress(“id2212_teachers@ict.kth.se”);
message.setFrom(fromAddress);
message.addRecipient(Message.RecipientType.TO, toAddress);
message.addRecipient(Message.RecipientType.CC, ccAddress);
message.addRecipient(Message.RecipientType.BCC, fromAddress);
```

# Transport

- An object of the `Transport` class used to connect to the mail server and to send a message;
  - Uses a specific protocol for sending messages (usually SMTP).
- For example:

```
message.saveChanges();
Transport transport = session.getTransport("smtp");
transport.connect();
transport.sendMessage(message, message.getAllRecipients());
transport.close();
```

# `Store` and `Folder`

- Represent an email store and an email folder, respectively
- An object of the `Folder` class is used for fetching messages from an associated mail folder
- For example:

```
Store store = session.getStore("pop3");
store.connect();
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
Message message[] = folder.getMessages();
System.out.println(((MimeMessage)message).
                                    getContent());
folder.close(expunge);
store.close();
```

# Sending Email Messages

1. Get the system Properties.
2. Setup a mail server:
   - Add the name of an SMTP server to the properties for the `mail.smtp.host` property key.
3. Get a Session object based on the Properties.
4. Create a MimeMessage from the session.
5. Set the from field of the message.
6. Add recepient(s) to the message (to, cc, bcc).
7. Set the subject of the message.
8. Set the content of the message.
9. Use a Transport to send the message.

# Fetching and Reading Email Messages

- Typical steps:
  1. Get the system Properties.
  2. Get a Session object based on the Properties.
  3. Get a Store for your email protocol, either pop3 or imap.
  4. Connect to the mail host's store with the appropriate username and password.
  5. Get the folder to read, e.g. the INBOX.
  6. Open the folder read-only.
  7. Get a directory of the messages in the folder (a list of messages).
  8. Display the messages one by one (e.g. the “from” field, the “subject” field, a message content).
  9. Close the connection to the folder and store.

# Email or Message Queue?

- Both message queues (MOM) and email servers can be used to create a loosely coupled, asynchronous messaging system
  - Both provide guaranteed delivery.
  - Both support point-to-point and publish/subscribe messaging models.
- Message queues are cleaner and more powerful if the message exchange needs to be machine-driven rather than human-driven.
- Message queues provide more programmatic control, are transactional, and can give better throughput.
- Using emails has the advantages of being human-readable and of using an infrastructure already available more or less everywhere.

# JNDI: Java Naming and Directory Interface

`javax.naming`

home page:

http://www.oracle.com/technetwork/java/index-jsp-137536.html

# JNDI Programming Concepts

- ***Name***
  - A generic name associated with an object or an object reference.
  - A naming system determines the syntax that the name must follow.
- ***Binding***
  - the association of a name with an object or object reference.
- ***Context***
  - a set of bindings. The object of a binding might be another context. If so, the contexts together form a tree.
- ***Initial context***
  - the starting (root) context.
- ***Naming service***
  - A server that enables binding names to objects and looking up objects by names.
- ***Directory service***
  - A server that provides a collection of named objects with attributes

# JNDI Programming Concepts (cont)

- ***Naming service***
  - the means by which names are bound to objects and objects are found by their names.
  - A client of the service can bind an object to a name, and look up an object by its name.
  - Provides a lookup (resolution) operation that returns the object with a given name
  - May provide operations for binding names, unbinding names, and listing bound names.
  - The operations are performed within the context.
- ***Context (Naming Service)***
  - A set of name-to-object bindings.
  - Has an associated naming convention.
  - Provides naming service operations performed in the context

# Storing Object in Naming Services

- Two general ways:
  - Store a serialized version of the Java object.
  - Store a reference with information how to construct or locate an instance of the object, for example
    - The class name
    - A vector of `RefAddr` objects representing address(es) of objects

# Directories and Directory Service

- ***Directory***
  - A set of directory objects.
  - A directory object is a named object with attributes, e.g. id/value pairs.
  - A directory can be searched for an object not only by its name but also by its attributes.
- ***Directory service***
  - Many naming services are extended with a directory service. A directory service associates names with objects and also allows such objects to have attributes.
  - A client of the service can bind an object to a name, set /change object's attributes, create subdirectories, search a directory for objects by names and/or by attributes.

# JNDI Architecture

- JNDI provides a Java API for an existing naming or directory service.
- The JDK includes for example the RMI and CORBA name services, that can be used as JNDI implementations.