JMS and JavaMail API

Java Message Service

Messaging systems allow information to be shared among many applications on a network.

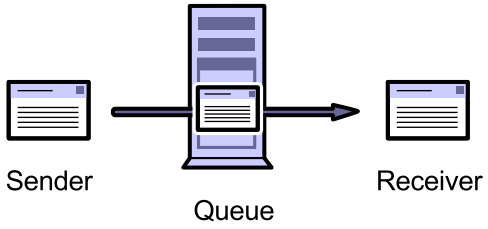
1. A sender creates a message with all the necessary information.
2. The sender gives the message to a message service.
3. The message service then takes the message and gives it to one or more systems set to receive the message.
4. The message recipient uses the information from the message to perform some task.
5. When the task is completed, the recipient might or might not send a response to the sender.

Common Messaging Models

- Point-to-point – Each message is processed by only one recipient.
- Publish/subscribe – Each message can be processed by zero or more subscribers.

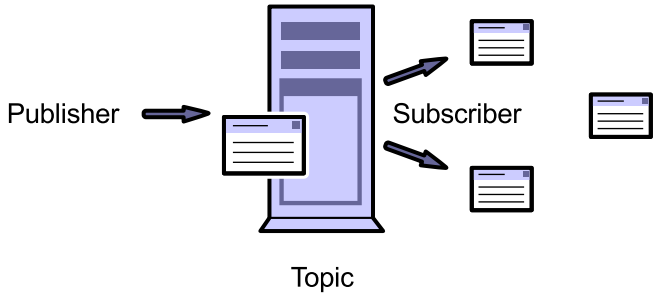
Point-to-Point Messaging in the JMS API

- Point-to-point messaging uses queue-based communication.
- Each message is processed by one receiving application.
- The receiver of a message does not have to be available when the message is sent.
- The receiver can send an acknowledgement of the successful retrieval of a message.



Publish/Subscribe Messaging in the JMS API

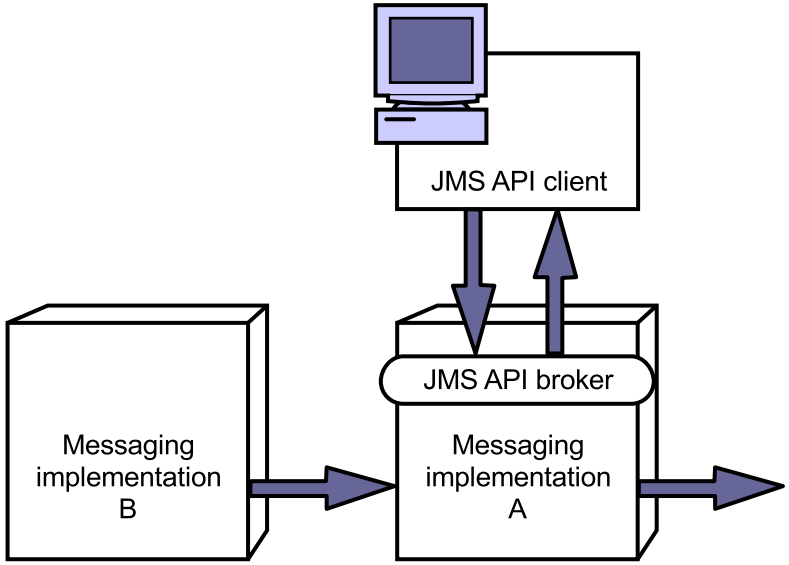
- Publisher applications create and send messages.
- Subscriber applications receive and process messages.
- Each message is received by zero or more subscribers.
- Publish/subscribe message is used for many-to-many communication.
- There is no timing dependency between the message sender and recipient.



JMS API

- Is a Java technology API for messaging
- Enables Java applications to access messaging software
- Integrates with the J.N.D.I. API to enable applications to locate messaging services
- Maintains vendor independence in a business application
- Supports the point-to-point and publish/subscribe models of messaging

Application Independence From a Messaging Implementation



Working With Messages in the JMS API

Each JMS API message has a standard format consisting of the following parts:

- Message header – Contains standard information for identifying and routing the message
- Message properties – Enable customization of the message properties and interoperability with some message service providers
- Message body – Contains the actual message in one of several standard formats

JMS API Message Types

A JMS API message can contain data in one of several formats:

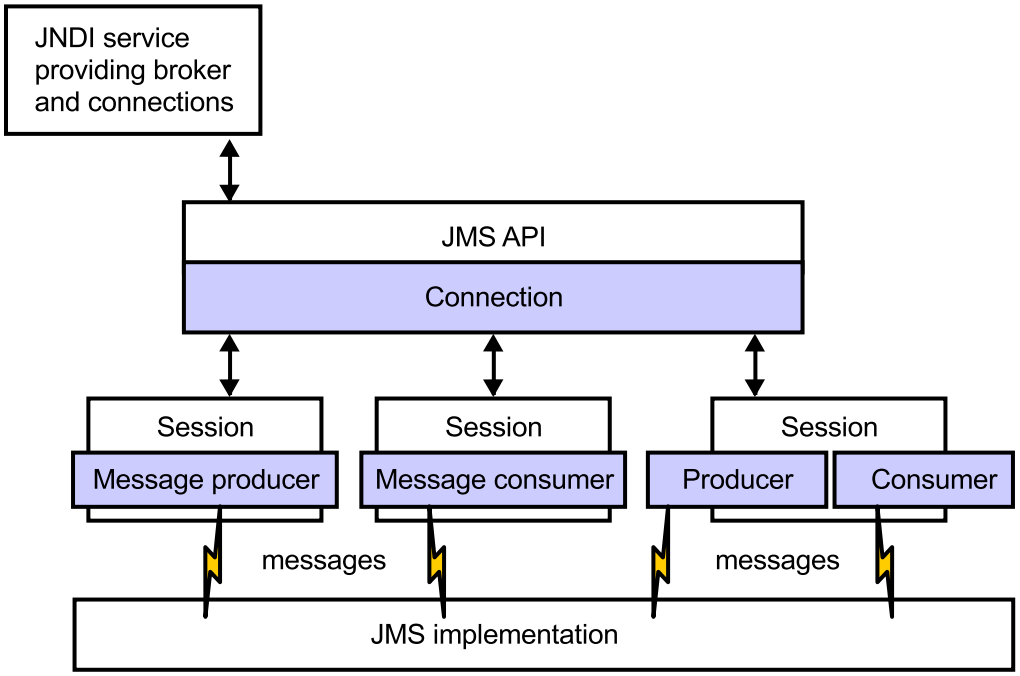
Message Type	**Contents of the Message Body**
`TextMessage`	A `java.lang.String` object
`MapMessage`	A set of name/value pairs, for example, a hash table
`BytesMessage`	A stream of uninterpreted bytes or binary data
`StreamMessage`	A stream of Java technology primitive values filled and read sequentially
`ObjectMessage`	A serializable Java technology object

Describing Key Interfaces in the JMS API

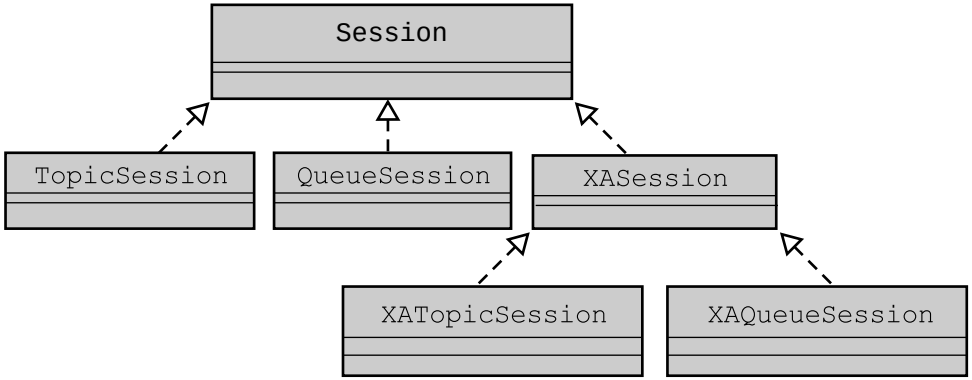
The `javax.jms` package contains the following interfaces used in a JMS API application:

- `Connection` – Encapsulates a virtual connection with a JMS API provider
- `Session` – Single-threaded context for producing and consuming messages
- `QueueSender` – An object created by a session used for sending messages to a queue
- `QueueReceiver` – An object created by a session used for receiving messages from a queue

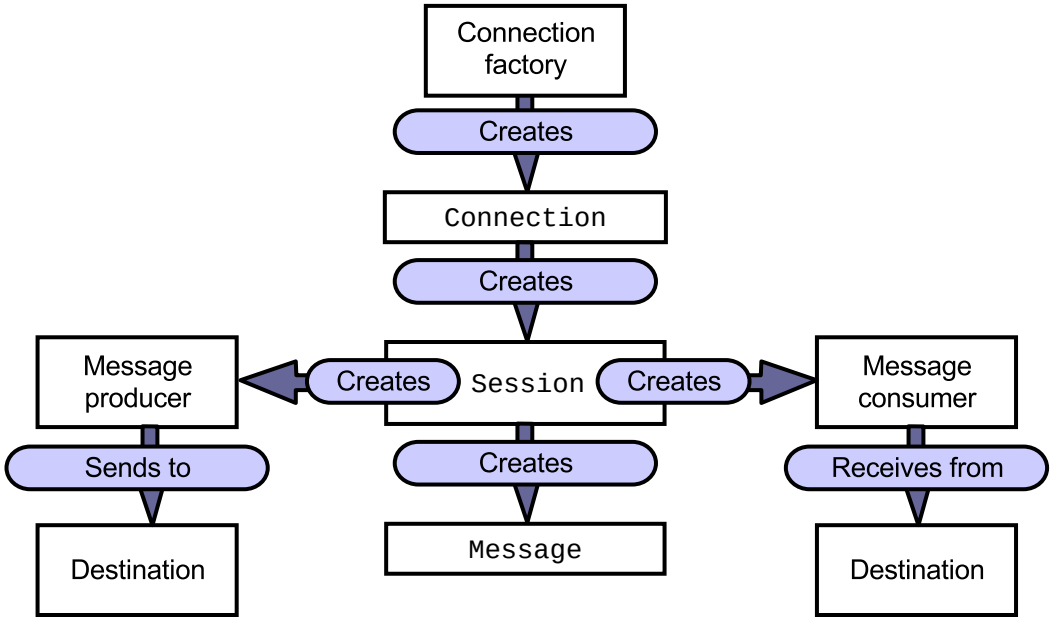
Overview of the JMS API Messaging Process



Session Interface Inheritance Hierarchy



Messaging Process



Creating a Point-to-Point JMS API Application

1. Look up a `Connection` factory using the J.N.D.I. API.
2. Look up the message queue using the J.N.D.I. API.
3. Create a `Connection` using the factory.
4. Create a `Session` object.
5. Create a `MessageSender` object.
6. Create one or more `Message` objects.
7. Send one or more `Message` objects using the `MessageSender` object.
8. Send a control message to the `Queue` object that all messages have been sent.

Creating a Point-to-Point JMS API Application

Example of sending messages to a queue:

```
18  try {
20    queueConnectionFactory = (QueueConnectionFactory)
21      jndiContext.lookup("QueueConnectionFactory");
22    queue = (Queue) jndiContext.lookup(queueName);
23    queueConnection =
24      queueConnectionFactory.createQueueConnection();
25    queueSession = queueConnection.createQueueSession(false,
26      Session.AUTO_ACKNOWLEDGE);
27    queueSender = queueSession.createSender(queue);
28    message = queueSession.createTextMessage();
29    message.setText("This is a simple message”);
30    queueSender.send(message);
31    queueConnection.close();
32  } catch (JMSException e) {
33    System.out.println("Exception occurred: " +
34  }
```

Receiving Messages From a Queue

```
1   try {
2     InitialContext jndiContext = new InitialContext();
3     factory = (QueueConnectionFactory)
4       jndiContext.lookup("QueueConnectionFactory");
5     queue = (Queue) jndiContext.lookup(queueName);
6     QueueConnection connection = factory.createQueueConnection ();
7     QueueSession session = connection.createQueueSession(false,
8       QueueSession.CLIENT_ACKNOWLEDGE );
9     receiver = session.createReceiver(queue);
10    receiver.setMessageListener (new MessageListener(){
11      public void onMessage(Message newMessage){
12        try    {
13          TextMessage message = (TextMessage) newMessage;
14          System.out.println("Message received ");
15          System.out.println(message.getText());
16          message.acknowledge ();
17        } catch (Exception e) {}
18      }
19    });
20    connection.start();
21  } catch (JMSException e){ }
22    catch (NamingException e) { }
```

Receiving Messages Synchronously

```
1   try {
2     InitialContext jndiContext = new InitialContext();
3     factory = (QueueConnectionFactory)
4       jndiContext.lookup("QueueConnectionFactory");
5     queue = (Queue) jndiContext.lookup(queueName);
6     QueueConnection connection = factory.createQueueConnection ();
7     QueueSession session = connection.createQueueSession(false,
8       QueueSession.CLIENT_ACKNOWLEDGE );
9     receiver = session.createReceiver(queue);
10    connection.start();
11    Message message = receiver.receive() // blocks here
12    TextMessage message = (TextMessage) newMessage;
13    System.out.println("Message received ");
14    System.out.println(message.getText());
15    message.acknowledge ();
16    connection.close();
17
18  } catch (JMSException e){ }
```

Creating a Publish/Subscribe JMS API Application

1. Look up a `TopicConnection` factory using the J.N.D.I. API.
2. Look up a `Topic` object using the J.N.D.I. API.
3. Create `Connection` and `Session` objects.
4. Create a `TopicPublisher` object.
5. Create one or more `Message` objects.
6. Publish one or more messages using the `TopicPublisher` object.

Creating a Publish/Subscribe JMS API Application

Example publish/subscribe the JMS API code:

```
1
2   try {
3     topicConnectionFactory = (TopicConnectionFactory)
4       jndiContext.lookup("TopicConnectionFactory");
5     topic = (Topic) jndiContext.lookup(topicName);
6     topicConnection =
7       topicConnectionFactory.createTopicConnection();
8     topicSession = topicConnection.createTopicSession(false,
9       Session.AUTO_ACKNOWLEDGE);
10    topicPublisher = topicSession.createPublisher(topic);
11    message = topicSession.createTextMessage();
12    message.setText("This is a simple publish/subscribe message”);
13    topicPublisher.publish(message);
14  } catch (JMSException e) {
15    System.out.println("Exception occurred: " + e.toString());
16  }
```

Subscribing to a Topic

```
1
2    try {
3      TopicConnectionFactory factory =(TopicConnectionFactory)
4        jndiContext.lookup("TopicConnectionFactory");
5      topic = (Topic) jndiContext.lookup(topicName);
6      TopicConnection connection = factory.createTopicConnection ();
7      TopicSession session = connection.createTopicSession(false,
8        TopicSession.CLIENT_ACKNOWLEDGE );
9      subscriber = session.createSubscriber(topic);
10     subscriber.setMessageListener (new MessageListener(){
11       public void onMessage(Message newMessage){
12         try {
13           TextMessage message = (TextMessage) newMessage;
14           System.out.println("Message received ");
15           System.out.println(message.getText());
16           message.acknowledge ();
17         } catch (Exception e) {}
18       }
19     });
20     connection.start();
21   } catch (JMSException e){ }
```

When Is the JMS API the Right Solution?

The JMS API might be the right solution for projects with one or both of the following characteristics:

- Messages need to be sent between disparate systems.
- Sender applications do not require receiver applications to be running.

Issues That the JMS API Does Not Address

- Does not specify security considerations
- Does not provide standardized error messages for system errors or errors referring to specific messages
- Does not provide the ability to invoke methods on the server and get a return value as a response

Advantages of the JMS API in the Enterprise

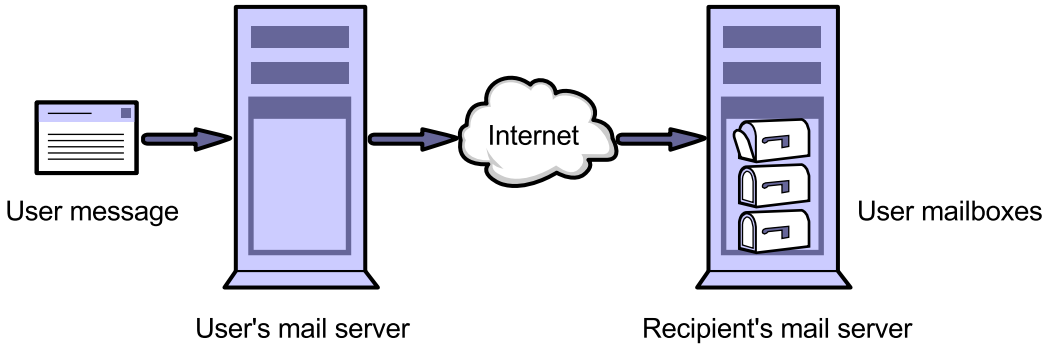
- Minimizes the client application's dependence on a particular server implementation
- Can use a JMS API provider to access legacy systems
- Enables client-side applications to perform responsively to the user
- Facilitates many-to-many communication

JavaMail API

Understanding Email

- The use of email has grown tremendously over the past several years.
- Internet email uses several standard protocols:
  - Simple Mail Transfer Protocol (SMTP)
  - Post Office Protocol 3 (POP3)
  - Internet Message Access Protocol (IMAP)
  - Multipurpose Internet Mail Extensions (MIME)
- Many applications exist to assist user access to email through these protocols.
- Many business systems require the functionality to automatically generate and send email.

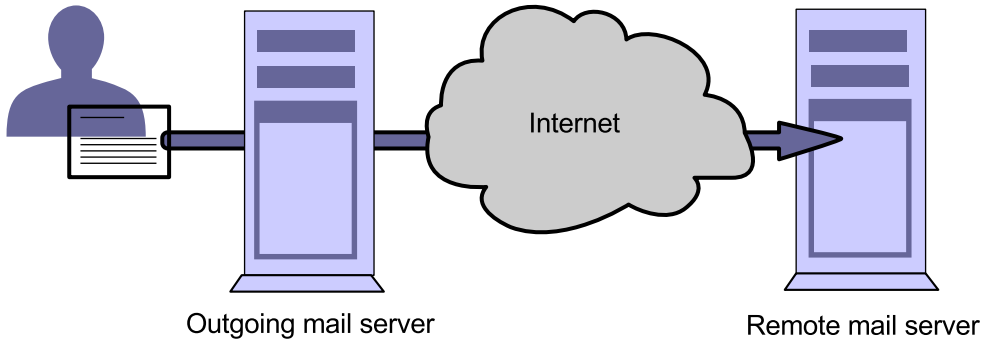
Email Being Sent to a Specific User's Mailbox



Describing SMTP

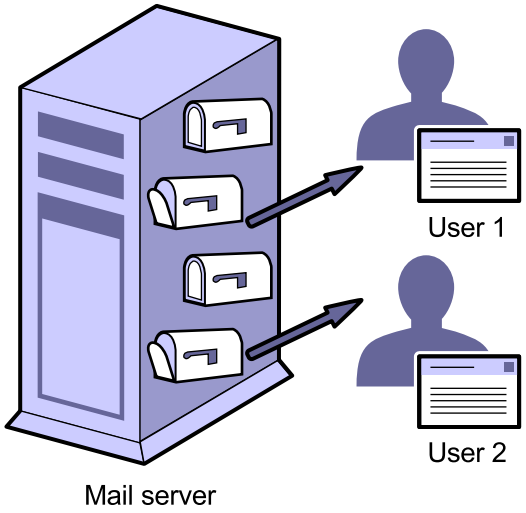
SMTP enables an email application to send mail messages to a recipient's mail server, using the following steps:

1. A user types an email message into an email application and then clicks the Send button.
2. The email message is transferred from the mail application to the sender's mail server.
3. The email is then sent from the sender's server through the Internet to the recipient's email server.



Describing POP3

- Defines a single mailbox for each user on a particular mail server
- Enables users to download messages to a local mail application (for example Netscape™ Mail)



Describing IMAP

- Stores the messages on the server
- Messages are only downloaded to the user's application when read
- Places a larger burden on the mail server because messages are stored for longer

Describing MIME

- Defines standard formats of files sent as an email message or attachment to an email
- Used by web browser applications to determine how to display data (for example, `gif` and `jpeg` images)
- Used by web servers to notify web browsers of the format of the data being sent

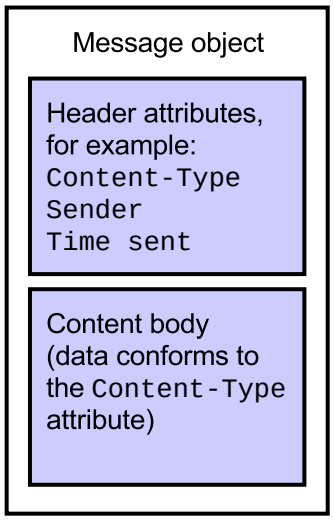
Type of Content	**Example MIME Types**
Audio	`audio/midi` **audio/x-pn-realaudio**
Images	`image/gif` `image/jpeg`
Text	`text/html` `text/plain`
Video	`video/mpeg` `video/quicktime`

Describing the JavaMail API

- Enables a Java application to access an email system
- Can be used to automatically create email messages and send them to users
- Enables access to the email systems using the standard protocols including SMTP, POP, and IMAP
- Can also be used to add attachments to email
- Provides the Java technology classes and interfaces in the package `javax.mail`
- Part of the J2EE platform

Describing the Core JavaMail API Classes

- `javax.mail.Session` – Defines the basic connection (session) with a mail server
- `javax.mail.Message` – Encapsulates the email message to send



- `javax.mail.Address` – Represents the email address of the sender or recipient of the message
- `javax.mail.Transport` – Uses a mail protocol (usually SMTP) to send the message to the server

Using JavaMail to Send Email

1. Create a `Session` object with the mail server.
2. Create a `Message` object to send.
3. Set the recipients and subject of the message.
4. Set the body of the message.
5. Add attachments, if desired.
6. Send the message using the `Transport` class.

Mail Session Properties

Property Name	Description
`mail.from`	Sender's email address.
`mail.`*`protocol`*`.host` for example, `mail.smtp.host`	Specifies the protocol-specific default mail server. This overrides the `mail.host` property.
`mail.`*`protocol`*`.user`	Specifies the protocol-specific default user name for connecting to the mail server. This overrides the `mail.user` property.
`mail.user`	Specifies the default user name to provide when connecting to a mail server. The `Store` and `Transport` object's `connect` methods use this property to obtain the user name if the protocol-specific user name property is absent.

Sending an Email Using the JavaMail API

```
1   Properties props = new Properties();
2   // code to fill props with information regarding mail server, such as
3   // props.put("mail.smtp.host", mailServer);
4   Session session = Session.getInstance(props, null);
5
6   Message msg = new MimeMessage(session);
7   msg.setFrom();
8   msg.setRecipients(Message.RecipientType.TO,
9     InternetAddress.parse("test@sun.com"), false));
10  msg.setSubject("Test Message");
11  String msgText = "This is a test of the emergency email system.";
12  msg.setText(msgText);
13  msg.setHeader("X-Mailer", mailer);
14
15  Transport.send(msg);
```

Using the JavaMail API to Receive Email

1. Create a `Session` object with the mail server.
2. Get a `Store` object from the server and connect to the `Store` object with your user name and password.
3. Get a `Folder` object from the `Store` object and open the `Folder` object.
4. Get an array of `Message` objects from the `Folder` object.

Code to Retrieve an Email Message Using the JavaMail API

```
1   Session session = Session.getDefaultInstance(properties, null);
2   Store store = session.getStore("pop3");
3   store.connect("<serverName>", "<username>", "<password>");
4   Folder folder = store.getFolder("INBOX");
5   folder.open(Folder.READ_ONLY);
6   Message messages[] = folder.getMessages();
7
8   for (int i=0; i < messages.length; i++) {
9     System.out.println(i + ": "+ messages[i].getFrom()[0]
10      + "\t" + messages[i].getSubject());
11  }
12  folder.close(false);
13  store.close();
```