Distributed Programming in Java

Session: 10

JavaMail

Objective

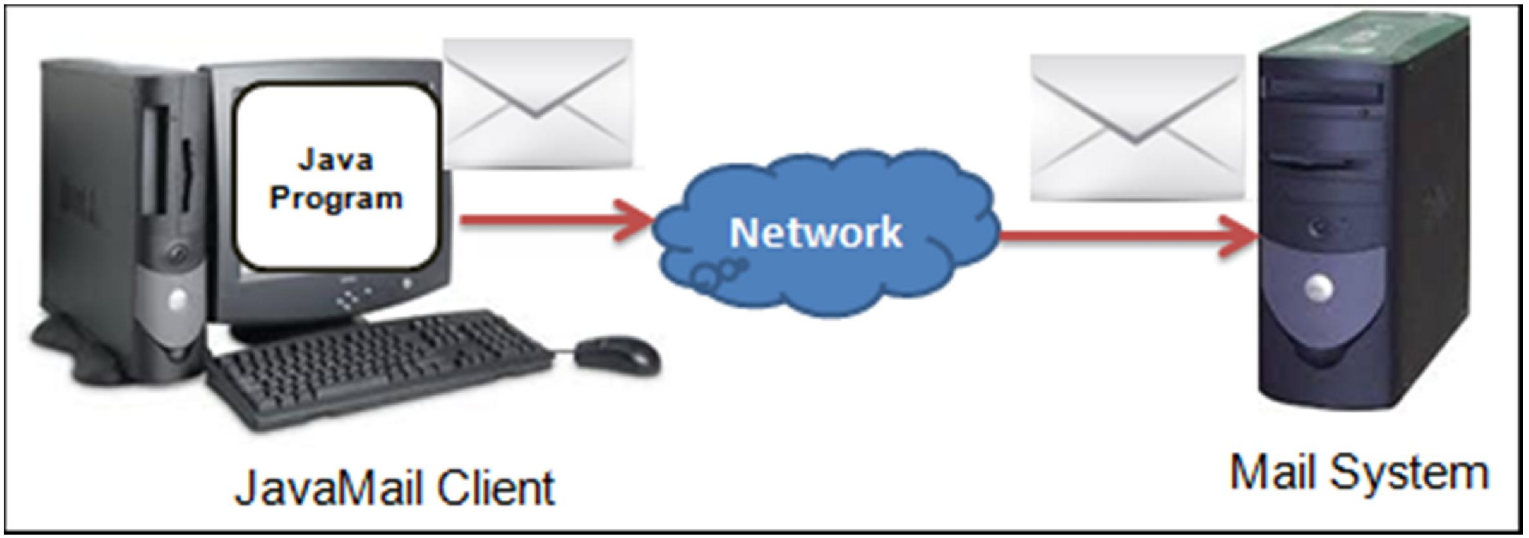
- Describe the JavaMail API
- Describe the various mail protocols
- Explain the concept of a Session
- Describe how to create a Message
- Identify the steps for creating and sending a message
- Discuss the steps required for reading a message
- Discuss the steps required for replying to a message

Introduction

- Java provides a platform independent and a protocol independent API named `JavaMail`.
- `JavaMail` is used to compose, read, and write mails through Java application.
- `JavaMail` API
  - Is not a part of the core package.
  - Is a standard extension and has to be downloaded separately.
  - Using it, you can create programs of type Mail User Agent (MUA).
  - Examples of MUA are Microsoft Outlook or Eudora. Yahoo and Hotmail are examples of Web-based MUAs.
  - Core classes of `JavaMail` API are available in the package namely, `javax.mail` and `javax.mail.Internet`.

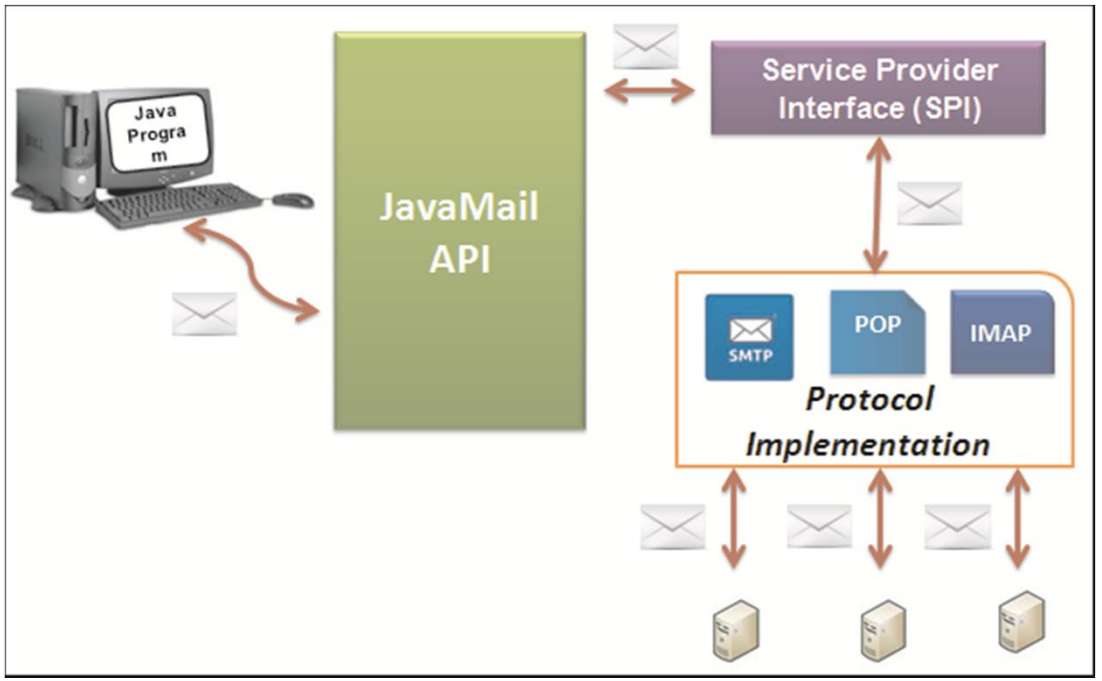
Use of JavaMail

- The `JavaMail` API is not responsible for the actual transporting, delivering, and forwarding messages.
- The main use of `JavaMail` API is that it provides a platform independent interface to different service providers providing their own messaging system.
- Figure shows the use of `JavaMail` API to manage mails on the mailing system.



JavaMail Architecture

- The `JavaMail` API interacts with Service Provider Interface (SPI) to send and receive mails.
- Further, the SIP provides implementation of the specific protocol, such as:
  - Simple Mail Transfer Protocol (SMTP)
  - Post Office Protocol (POP)
  - Internet Message Access Protocol (IMAP)
- It acts as a provider to email service for the Java application.
- Figure shows the `JavaMail` architecture.



Mail Protocols

- The various mail protocols used with `JavaMail` are:
  - SMTP
    - This protocol is used for delivery of email.
    - When you use the `JavaMail` API in your mail based program, the program communicates with the SMTP server of the Internet Service Provider (ISP).
  - POP
    - The current version of POP is version 3; hence, it is referred to as POP3.
    - This protocol is used for receiving email.
    - The POP3 supports a single mailbox for each user; all the mails are sent to this mailbox.
  - IMAP
    - This protocol is used for receiving email and is a more advanced protocol.
    - The current version of IMAP is 4; hence, it is referred to as IMAP4.
  - MIME
    - This protocol is not used for sending or receiving mail, it defines the content type of the message transferred.
    - This protocol also defines the format of the messages and attachments.

JavaMail API

- The JavaMail API defines three main classes namely, `Session`, `Message`, and `Transport`. These classes model the mail system.

Session

- Is a lasting connection between a user and a host, usually a server.
- Starts when the service is first provided and closes when the service is ended.
- A user may have multiple active sessions simultaneously.
- Represents the complete state of a service.
- The `javax.mail` package
  - Provides a class `Session` to define a basic mail session.
  - Once a session is made available, it is through this session that all subsequent operations of mailing system are carried out.

Methods of Session [1-2]

- `public static Session getInstance(Properties prop)`
  - The method creates a `Session` object based on the initial values provided in the `Properties` object.
  - The `JavaMail` specification specifies that the following properties - `mail.store.protocol`, `mail.transport.protocol`, `mail.host`, `mail.user`, and `mail.from` should be specified in the `Properties` object.

- `public static Session getDefaultInstance()`
  - The method returns a default `Session` object.
  - If a default `Session` object is not available, it creates a new one, installs it as the default and then returns it.

Methods of Session [2-2]

- `public static Session getDefaultInstance(Properties prop, Authenticator auth)`
  - The method is used to retrieve the default `Session` object.
  - If a default `Session` is not available a new `Session` object is created and set as a default.
- `public static Session getInstance(Properties prop, Authenticator auth)`
  - The method is used to retrieve a new `Session` object.
- `public Properties getProperties()`
  - The `getProperties()` method is used to retrieve the `Properties` object associated with the specified `Session` object.
- `public String getProperty(String name)`
  - The `getProperty()` method is used to retrieve the value of the specified property name associated with the `Session` object.
  - The method returns a null value if there are no properties.

Creating a Message

- Once a `Session` object is created, you have to create a `Message` object from that `Session` object.
- This `Message` object can then be sent to a mail recipient.
- The `Message` object has several parts such as address 'from', 'to' and 'replyto', and other parts like 'date' and 'subject'.
- The `javax.mail.Message` is an abstract class and is not used directly; its subclass which provides concrete implementation is used.
- The `javax.mail.Message` class implements a `Part` interface which specifies the attributes and content type of the message.

Methods of the Message Class [1-3]

- `setReplyTo(Address[] addresses)`: The method is used to set the 'Reply-To' field of the message. If the argument address is null, then the 'Reply-To' header is removed from the message.

Code Snippet

```
try  {          . . .
    Address[] addresses = {new InternetAddress("mike@gmail.com"),
    new InternetAddress("alex@gmail.com") };
    Session session = Session.getInstance(props, null);
    // Construct the message
    Message msg = new MimeMessage(session);
    // Set the addresses of "Reply-To" header
    msg.setReplyTo(addresses);
. . .
} catch(AddressException ex) {
    System.out.println("Exception : " + ex.getMessage());
} catch(MessagingException ex) {
    System.out.println("Exception : " + ex.getMessage()); }
. . .
```

Methods of the Message Class [2-3]

- `public Address[] getAllRecipients()`: The method is used to retrieve all the recipient addresses of the message. It extracts the address of 'TO', 'CC', and 'BCC' recipients and returns a null value if recipient headers are not present in the message.
- If recipient header is present but does not contain any addresses then the method might return an empty array.
- Code Snippet shows how to retrieve the addresses of all recipients.

Code Snippet

```
. . .
String strFrom = "mails.vincent@gmail.com";
String strTo =  "mails.mike@gmail.com";
String strCc = "alex@gmail.com";
String strBcc =  "john@gmail.com";
try  {
    // Retrieve the session instance with the desired attributes
    Session session = Session.getInstance(props, null);
```

Methods of the Message Class [3-3]

```
    Message msg = new MimeMessage(session);
    msg.setFrom(new InternetAddress(strFrom));
    msg.setRecipients(Message.RecipientType.TO, InternetAddress.
        parse(strTo,  false));
    msg.setRecipients(Message.RecipientType.CC, InternetAddress.
        parse(strCc,  false));
    msg.setRecipients(Message.RecipientType.BCC, InternetAddress.
        parse(strBcc,  false));
    Address[] recipientAddresses = msg. getAllRecipients();
    for(int i = 0; i < recipientAddresses.length; i++) {
        System.out.println("Addresses are : " +
        recipientAddresses[i]);
    }
} catch(AddressException ex) {
    System.out.println("Exception : " + ex.getMessage());
} catch(MessagingException ex) {
    System.out.println("Exception : " + ex.getMessage());
}
. . .
```

MimeMessage Class

- The `javax.mail.internet.MimeMessage` class is a subclass of the `Message` class and is used to create a `Message` object.
- This `Message` object represents a MIME style email message.
- To create a `Message` object, you first create an empty `MimeMessage` object and then set the appropriate attributes and contents.
- Steps to use `MimeMessage` object:
  - Create an empty `Message` object from a session
  - Set the attributes required
  - Set the content type of the message
  - Set the content of the message

Determining Size of a Message [1-2]

- The `getSize()` method is used to retrieve the size of the message contents in bytes.
- If the size of the message contents cannot be determined a value of `-1` is returned.
- The `getSize()` method may not return the exact size of the contents if the message has encoded contents.
- Code Snippet shows how to retrieve the number of new messages in the folder.

Code Snippet

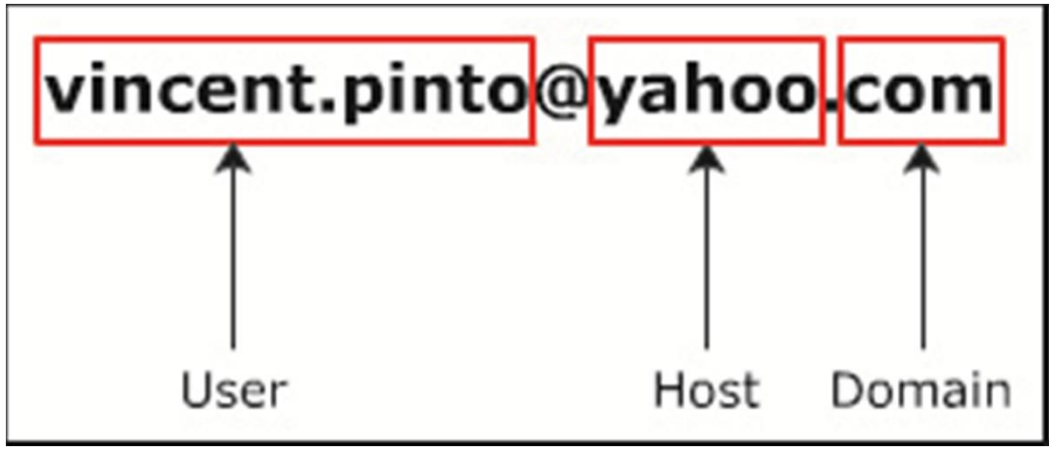
```
. . .
String strHost =”10.1.1.23”;
String strUser = “johnk@aptech.ac.in”;
String strPass = “ WinJohn400”;
int  openMode; Store store;
try {
    // Retrieve the session instance with the desired
       attributes
    Properties props = System.getProperties();
    props.put(“mail.pop3.host”, strHost);
    Session session = Session.getInstance(props);
```

Determining Size of a Message [2-2]

```
Message msg = new MimeMessage(session);
Folder folder = null;
// Retrieve the store
store = session.getStore("pop3");
// Establish connection with the mail host
store.connect(strHost, strUser, strPass);
if (store != null) {
    folder = store.getDefaultFolder()
if (folder == null) throw new Exception("No default folder");
// Retrieve the INBOX folder
folder = folder.getFolder("INBOX");
// Open the folder in read only mode
folder.open(Folder.READ_ONLY);
// Retrieve the open mode
openMode = folder.getMode();
if (openMode == Folder.READ_ONLY) {
        System.out.println("Open mode : READ_ONLY");
}
int  newMessages = folder.getNewMessageCount();
```

Concept of Address

- The address is an Internet email address of a user for a specific domain.
- Generally the address is of the form `user@host.domain`.
- When you use the services of a mailing system such as yahoo or rediff, you have to create an account with this service provider.
- This account is to identify one user from another and is unique for each user.
- In the address, the user name has to be unique followed by a `@` sign, the host is the service provider like yahoo.
- The domain identifies the type organization for example, `.com` represents commercial organization, `.edu` represents educational institution, and so on.
- Figure displays the concept of address.



Creating Address [1-2]

- The `javax.mail` provides an `Address` class which is an `abstract` class.
- This address is required by the `Message` class object to specify the parts like sender and its recipients.
- The `javax.mail.internet.InternetAddress` is generally used to create the address required for sender and its recipients.
- The `InternetAddress` is a subclass of `javax.mail.Address` class.
- Constructor: `public InternetAddress(String address) throws AddressException`
- Code Snippet shows how to create an address using the one argument constructor.

Code Snippet

```
String strFrom = "mails.vincent@gmail.com";
try {
InternetAddress fromAddress = new InternetAddress(strFrom);
} catch(AddressException ex) {
    System.out.println("Exception : " + ex.getMessage());
}
```

Creating Address [2-2]

- **Parse() method**
  - The static method `parse()` is generally used to parse several address separated by commas.
  - When you send or reply to a mail there can be several recipients.
  - Comma delimited email addresses are provided in To, Cc, and Bcc.
  - This method returns an array of addresses by parsing the string containing the addresses.

    Syntax:

    ```
    public static InternetAddress[] parse(String addressList)
    throws AddressException
    ```

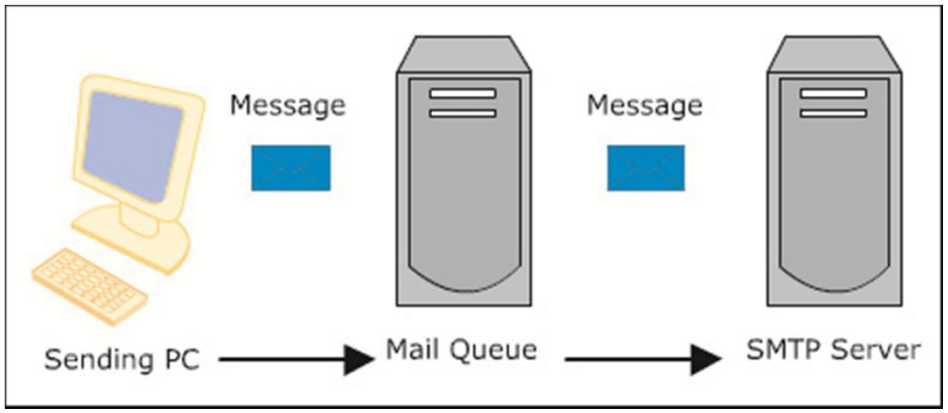
  - Code Snippet shows how to parse a comma delimited list of addresses to create an array of `InternetAddress` class.

Code Snippet

```
String strCc = "mails.mike@gmail.com, fred@gmail.com,
    mike@gmail.com";
try {
    InternetAddress[] ccAddress = InternetAddress.parse(strCc,
    false);
} catch(AddressException ex) {
    System.out.println("Exception : " + ex.getMessage());
}
```

Concept of Transport Class

- Once the `Message` object is created and its required attributes are set, the final step involves sending the message.
- The `javax.mail.Transport` class is used to send the message to all its recipients.
- **The `javax.mail.Transport`**
  - Is an abstract class.
  - Has static methods to send messages to all the recipients.
  - Contains class static methods that throw a `SendFailedException` exception if any of the recipient address is found to be invalid.
  - Successful sending of the message to its recipients does not imply that the message has reached its ultimate recipient.
  - Failures can still occur in the later stages of delivery.
  - Figure displays the concept of `Transport` class.



Methods of Transport Class [1-3]

- `send(Message msg)`: The method takes one argument, an object of the `Message` class, and sends the message to all the recipients specified in the `Message` object.
- Code Snippet shows how to send a message to all its recipients.

Code Snippet

```
try  {
    Session session = Session.getInstance(props, null);
    Message msg = new MimeMessage(session); // Construct
    msg.setFrom(new InternetAddress(strFrom));
    msg.setRecipients(Message.RecipientType.TO,InternetAddress.pars
            e(strTo, false)); //Set the recipients of the message
    msg.setSubject(strSubject);  // Set the subject
    msg.setText(strText);  // Send the text to the recipients
    Transport.send(msg);
} catch(AddressException ex) {
    System.out.println("Exception : " + ex.getMessage());
} catch(MessagingException ex) {
    System.out.println("Exception : " + ex.getMessage());
}
```

Methods of Transport Class [2-3]

- `send(Message msg, Address[] addresses)`: The method takes two arguments, an object of the `Message` class, and an array of `Address` class. This method sends the message to the addresses in the array specified. The method ignores all the addresses of recipients in the `Message` object.
- Code Snippet shows how to send a message to all the recipients from an `Address` array.

Code Snippet

```
Address[] addresses = {new InetAddress("mike@gmail.com"),  new
    InetAddress("mike@gmail.com"), };
try  {
    Session session = Session.getInstance(props, null);
    // Construct the message
    Message msg = new MimeMessage(session);
    // Set the sender of the message
    msg.setFrom(new InternetAddress(strFrom));
    // Set the recipients of the message
    msg.setRecipients(Message.RecipientType.TO,InternetAddress.parse(s
        trTo, false));
    // Set the subject of the message
    msg.setSubject(strSubject);
```

Methods of Transport Class [3-3]

```
    // Set the text of the message
    msg.setText(strText);
    // Send the message to the recipients
    Transport.send(msg, addresses);
} catch(AddressException ex) {
    System.out.println("Exception : " + ex.getMessage());
} catch(MessagingException ex) {
    System.out.println("Exception : " + ex.getMessage());
}
```

Store

- A mail service provider maintains a store which contains the folder where your messages are kept.
- To retrieve a message from email service, you have to gain access to the store first. The POP3 protocol maintains its own store.
- Similarly, the IMAP maintains its own store.
- Once you have created a `Session` object, you gain access to the store by specifying the protocol.
- The `Session` class has a method `getStore()` which takes one argument, the protocol implemented by the store you want to gain access.
- The `Store` object returned by the `getStore()` method is used to establish a connection with your user id and password.

Accessing Message Store [1-3]

- The `javax.mail.Store` class represents a message store and the access protocol, for storing and retrieving messages.
- To gain access to the message store, perform the following steps:
  - **Retrieve the `Store` object from session**
    - Code Snippet shows how to retrieve the `Store` object from a session.

Code Snippet

```
// Retrieve the system properties
Properties props = System.getProperties();
// Assigns the mail.transport.protocol attribute as pop3
props.put("mail.transport.protocol", "pop3");
// Set the port to 995
props.setProperty("mail.pop3.port", "995");
// Retrieve the system properties
Properties props = System.getProperties();
// Assigns the mail.transport.protocol attribute as pop3
props.put("mail.transport.protocol", "pop3");
// Set the port to 995
props.setProperty("mail.pop3.port", "995");
```

Accessing Message Store [2-3]

```
try {
// Retrieve the session instance
Session session = Session.getInstance(props);
// Retrieve the store
store = session.getStore("pop3");
} catch(NoSuchProviderException ex) {
System.out.println("Exception : " + ex.getMessage());
```

- **Connect to the mail host with authentication**
  - Code Snippet shows how to connect to the mail host.

Code Snippet

```
String strHost = "pop.gmail.com";
String strUser = "mails.vincent@gmail.com";
String strPass = "vincent";
// Retrieve the system properties
Properties props = System.getProperties();
// Assign the mail.transport.protocol attribute as pop3
props.put("mail.transport.protocol", "pop3");
// Set the port to 995
props.setProperty("mail.pop3.port", "995"); try {
// Retrieve the session object with properties
```

Accessing Message Store [3-3]

```
Session session = Session.getInstance(props);
// Retrieve the store
store = session.getStore("pop3");
// Establish connection with the mail host
store.connect(strHost, strUser, strPass);
} catch(MessagingException ex) {
System.out.println("Exception : " + ex.getMessage());
```

Folder

- `Folder` is the namespace where your messages are available.
- For the protocol POP3 there is only one folder INBOX.
- The IMAP protocol supports multiple folders.
- Once `Store` object is connected you can retrieve the folder.
- The messages are then retrieved from the folder.
- The `Folder` class represents a folder for the message.
- `Folders` can contain other folders or messages or both.
- The folder names are implementation dependent.
- The `create()` method is used to create a folder and the `exists()` method is used to check whether a folder exists or not.
- Some of the methods of the `Folder` class are as follows:
  - `getMode()`
  - `getNewMessageCount()`

Methods of Folder Class

- `public int getMode()`: The method is used to retrieve the open mode of the folder. The mode specifies whether the folder supports 'ready-only' or 'read-write' operations. If the open mode cannot be determined a value of -1 is returned.

- `public int getNewMessageCount() throws MessagingException`: The method is used to retrieve the number of new messages in the folder. Some implementations of the folder may not support this operation because it is a costly operation, in which case a -1 is returned.

Steps to Read Message from the Folder [1-5]

- To read a message from the folder, perform the following steps:
  - Retrieve the default folder
  - From the default folder retrieve the `INBOX` folder
  - Open the `INBOX` folder
  - Retrieve the messages

Steps to Read Message from the Folder [2-5]

- **Retrieve the default folder**
  - The `Store` class has a method `getDefaultFolder()` which returns the reference to the default folder.
  - The default folder is an object of the `Folder` class.
  - Code Snippet shows how to retrieve the default folder.

Code Snippet

```
try {
    Folder folder;// Retrieve the session object with properties
    Session session = Session.getInstance(props);
    // Retrieve the store
    store = session.getStore("pop3");
    // Establish connection with the mail host
    store.connect(strHost, strUser, strPass);
    if (store != null) {
         // Retrieve the default folder
         folder = store.getDefaultFolder();}
 }
 catch(MessagingException ex) {
     System.out.println("Exception : " + ex.getMessage());
 }
```

Steps to Read Message from the Folder [3-5]

- **From the default folder retrieve the INBOX folder**
  - The `Folder` class has the method `getFolder()` to retrieve the named folder; in case of POP3 it is INBOX.
  - Code Snippet shows how to retrieve the named folder like INBOX.

Code Snippet

```
try {
    Folder folder;
    Session session = Session.getInstance(props);
    store = session.getStore("pop3");
    store.connect(strHost, strUser, strPass);
    if (store != null) {
        // Retrieve the default folder
        folder = store.getDefaultFolder();
        // Retrieve the INBOX folder
        folder = folder.getFolder("INBOX");
    }
} catch(MessagingException ex) {
    System.out.println("Exception : " + ex.getMessage());
}
```

Steps to Read Message from the Folder [4-5]

- **Open the INBOX folder**
  - Once you have the reference of the named folder like INBOX, you use the `open()` method of the folder with the desired mode.
  - Code Snippet shows how to open the named folder like INBOX with read only mode.

Code Snippet

```
try {
     Folder folder;
     Session session = Session.getInstance(props);
     store = session.getStore("pop3");
     store.connect(strHost, strUser, strPass);
     if (store != null) {
          // Retrieve the default folder
          folder = store.getDefaultFolder();
          folder = folder.getFolder("INBOX");
          // Open the folder in read only mode
          folder.open(Folder.READ_ONLY);
     }
} catch(MessagingException ex) {
     System.out.println("Exception : " + ex.getMessage());
}
```

Steps to Read Message from the Folder [5-5]

- **Retrieve the messages**
  - Once the folder is opened you can retrieve messages with the `getMessages()` method of the `Folder` class.
  - This method returns an array of `Message` class, which contains your messages.

Code Snippet

```
try {
    // Retrieve the messages from the folder
    Message[] msgs = folder.getMessages();
    for (int i = 0; i < msgs.length; i++)  {
        String subject = msgs[i].getSubject();
        System.out.println("Subject : " + subject);
    }
} catch(MessagingException ex) {
    System.out.println("Exception : " + ex.getMessage());
}
```

Exception Class

- Some of the important exceptions related to `JavaMail` API are:
  - `NoSuchProviderException`
  - `MessagingException`
  - `AddressException`
  - `SendFailedException`

Creating a Message and Mailing [1-3]

To create and send a message using `JavaMail` API, perform the following steps:

- Create and initialize a Properties object.
- Create a Session object with the initialized properties.
- Create a Message object from the session.
- Set the various attributes of the message.
- Send the message with a `Transport` object.

Code Snippet

```
String strHost = "pop.gmail.com";
String strTo  = "mike@gmail.com";
String strFrom = "mails.vincent.com";
String strCc = "alex@gmail.com";
String strBcc = "fred@gmail.com";
String strSubject = "Weekend party";
String strText = "Hi all, you are all invited to a party this Saturday";
// Retrieve the system properties
java.util.Properties props = System.getProperties();
```

Creating a Message and Mailing [2-3]

```
// Assign the attribute mail.smtp.host as host
props.put("mail.smtp.host", strHost);
// Assign the attribute mail.transport.protocol as protocol
props.put("mail.transport.protocol", "smtp");
// Assign the attribute mail.smtp.port as port number
props.put("mail.smtp.port", "25");
try {
    // Create the session object
    Session session = Session.getInstance(props, null);
    // Construct the message object from the session
    Message msg = new MimeMessage(session);
    // Set the From address
    msg.setFrom(new InternetAddress(strFrom));
    // Set the To address
    msg.setRecipients(Message.RecipientType.TO,InternetAddress.pa
    rse(strTo, false));
    // Set the Cc address
    msg.setRecipients(Message.RecipientType.CC,InternetAddress.pa
    rse(strCc, false));
```

Creating a Message and Mailing [3-3]

```
    // Set the Bcc address
    msg.setRecipients(Message.RecipientType.BCC,InternetAddress.p
    arse(strBcc, false));
     // Set the subject of the message
     msg.setSubject(strSubject);
    // Set the message text as plain text
    msg.setText(strText);
    // Set the current date
    msg.setSentDate(new Date());
    // Send the message to its recipients
    Transport.send(msg);
    System.out.println("\nMail was sent successfully.");
} catch (MessagingException ex) {
        System.out.println("Exception : " + ex.getMessage());
}
```

Reading the Message [1-3]

To read a message from a folder, perform the following steps:

- Create and initialize a Properties object
- Create a `Session` object with the initialized properties
- Create a `Store` object
- Use the `Store` object to connect to the host
- Retrieve the default folder
- Retrieve the named folder from the default folder
- Open the folder
- Retrieve the messages from the folder

Code Snippet

```
JTextField txtName;
JPasswordField pswPassword;
String strHost = "pop.gmail.com";
String strUser = txtUserName.getText().trim();
String strPass = pswPassword.getText().trim();
Store store = null; Folder folder = null;
```

Reading the Message [2-3]

```
try {  // Retrieve the system properties
   Properties props = System.getProperties();
   // Assign the attribute mail.pop3.host as host
   props.put("mail.pop3.host", strHost);
   // Assign the attribute mail.transport.protocol as protocol
   props.put("mail.transport.protocol", "pop3");
   // Assign the attribute mail.pop3.port as port number
   props.put("mail.pop3.port", "995");
   // Create the session object
   Session session = Session.getInstance(props);
   // Retrieve the store
   store = session.getStore("pop3");
   // Establish connection with the mail host
   store.connect(strHost, strUser, strPass);
   folder = store.getDefaultFolder();
   // Retrieve the INBOX folder
   folder = folder.getFolder("INBOX");
   // Open the folder in read only mode
   folder.open(Folder.READ_ONLY);
```

Reading the Message [3-3]

```
    // Retrieve the messages from the folder
    Message[] msgs = folder.getMessages();
    msgBodies = new String[msgs.length];
    for (int i = 0; i < msgs.length; i++) {
        // Retrieve subject of the message String
        subject=msgs[i].getSubject();
        System.out.println("Subject : " + subject);
}
} catch (MessagingException ex) {
        System.out.println("Exception : " + ex.getMessage();}
```

Replying to the Message [1-2]

To reply to a message, perform the following steps:

- Retrieve the messages
- Select the message to reply and edit the text
- Use the `reply()` method of `Message` class

Code Snippet

```
try {
    Message message;
    // TextArea to display and edit the message text
    JTextArea txaText;  int index;
    // Retrieve the message and display them in a GUI
    . . .
    // Allow to select a message from the GUI and edit the text
    message = msgs[index];
    // Retrieve the edited text
    strText = txaText.getText();
    // Reply to the message
    MimeMessage reply = (MimeMessage)message.reply(false);
```

Replying to the Message [2-2]

```
    // Set the sender's address
    reply.setFrom(new
        InternetAddress("mails.vincent@gmail.com"));
    // Set the message
    reply.setText(strText);
    // Send the message
    Transport.send(reply);
} catch(AddressException ex) {
    System.out.println("Exception : " + ex.getMessage());
} catch(MessagingException ex) {
    System.out.println("Exception : " + ex.getMessage());
}
```

Summary

- The JavaMail API is used to read, compose, and send electronic messages.
- The JavaMail API is not responsible for the actual transporting, delivering, and forwarding messages.
- In the context of a mailing system a session is a lasting connection between a user and a host, usually a server.
- MimeMessage class object represents a MIME style email message.
- Once the Message object is created and its required attributes are set, the final step involves sending the message.
- Creating and sending emails using JavaMail involves creating and sending a message, reading a message, and replying to a message.