Fundamentals of the JavaMail API

Skill Level: Introductory

John Zukowski (jaz@zukowski.net)
President
JZ Ventures, Inc.

22 Aug 2001

Looking to incorporate mail facilities into your platform-independent Java™ solutions? Look no further than the JavaMail API, which offers a protocol-independent model for working with IMAP, POP, SMTP, MIME, and all those other Internet-related messaging protocols.

Section 1. About this tutorial

Should I take this tutorial?

Looking to incorporate mail facilities into your platform-independent Java solutions? Look no further than the JavaMail API, which offers a protocol-independent model for working with IMAP, POP, SMTP, MIME, and all those other Internet-related messaging protocols. With the help of the JavaBeans Activation Framework (JAF), your applications can now be mail-enabled through the JavaMail API.

Concepts

After completing this module you will understand the:

- Basics of the Internet mail protocols SMTP, POP3, IMAP, and MIME
- Architecture of the JavaMail framework
- Connections between the JavaMail API and the JavaBeans Activation Framework

Objectives

By the end of this module you will be able to:

- Send and read mail using the JavaMail API
- Deal with sending and receiving attachments
- Work with HTML messages
- Use search terms to search for messages

Prerequisites

Instructions on how to download and install the JavaMail API are contained in the course. In addition, you will need a development environment such as the JDK 1.1.6+ or the Java 2 Platform, Standard Edition (J2SE) 1.2.x or 1.3.x.

A general familiarity with object-oriented programming concepts and the Java programming language is necessary. The Java language essentials tutorial can help.

Section 2. Introducing the JavaMail API

What is the JavaMail API?

The JavaMail API is an optional package (standard extension) for reading, composing, and sending electronic messages. You use the package to create *Mail User Agent* (MUA) type programs, similar to Eudora, pine, and Microsoft Outlook. The API's main purpose is not for transporting, delivering, and forwarding messages; this is the purview of applications such as sendmail and other Mail Transfer Agent (MTA) type programs. MUA-type programs let users read and write e-mail, whereas MUAs rely on MTAs to handle the actual delivery.

The JavaMail API is designed to provide protocol-independent access for sending and receiving messages by dividing the API into two parts:

- The first part of the API is the focus of this course -- basically, how to send and receive messages independent of the provider/protocol.
- The second part speaks the protocol-specific languages, like SMTP, POP, IMAP, and NNTP. With the JavaMail API, in order to communicate with a server, you need a *provider* for a protocol. The creation of protocol-specific providers is not covered in this course because Sun provides a sufficient set for free.

Section 3. Reviewing related protocols

Introduction

Before looking into the JavaMail API specifics, let's step back and take a look at the protocols used with the API. There are basically four that you'll come to know and love:

- SMTP
- POP
- IMAP
- MIME

You will also run across NNTP and some others. Understanding the basics of all the protocols will help you understand how to use the JavaMail API. While the API is designed to be protocol agnostic, you can't overcome the limitations of the underlying protocols. If a capability isn't supported by a chosen protocol, the JavaMail API doesn't magically add the capability on top of it. (As you'll soon see, this can be a problem when working with POP.)

SMTP

The Simple Mail Transfer Protocol (SMTP) is defined by RFC 821. It defines the mechanism for delivery of e-mail. In the context of the JavaMail API, your JavaMail-based program will communicate with your company or Internet Service Provider's (ISP's) SMTP server. That SMTP server will relay the message on to the SMTP server of the recipient(s) to eventually be acquired by the user(s) through POP or IMAP. This does not require your SMTP server to be an open relay, as authentication is supported, but it is your responsibility to ensure the SMTP server is configured properly. There is nothing in the JavaMail API for tasks like configuring a server to relay messages or to add and remove e-mail accounts.

POP

POP stands for Post Office Protocol. Currently in version 3, also known as POP3, RFC 1939 defines this protocol. POP is the mechanism most people on the Internet use to get their mail. It defines support for a single mailbox for each user. That is all it does, and that is also the source of a lot of confusion. Much of what people are familiar with when using POP, like the ability to see how many new mail messages they have, are not supported by POP at all. These capabilities are built into

programs like Eudora or Microsoft Outlook, which remember things like the last mail received and calculate how many are new for you. So, when using the JavaMail API, if you want this type of information, you have to calculate it yourself.

IMAP

IMAP is a more advanced protocol for receiving messages. Defined in RFC 2060, IMAP stands for Internet Message Access Protocol, and is currently in version 4, also known as IMAP4. When using IMAP, your mail server must support the protocol. You can't just change your program to use IMAP instead of POP and expect everything in IMAP to be supported. Assuming your mail server supports IMAP, your JavaMail-based program can take advantage of users having multiple folders on the server and these folders can be shared by multiple users.

Due to the more advanced capabilities, you might think IMAP would be used by everyone. It isn't. It places a much heavier burden on the mail server, requiring the server to receive the new messages, deliver them to users when requested, *and* maintain them in multiple folders for each user. While this does centralize backups, as users' long-term mail folders get larger and larger, everyone suffers when disk space is exhausted. With POP, saved messages get offloaded from the mail server.

MIME

MIME stands for Multipurpose Internet Mail Extensions. It is not a mail transfer protocol. Instead, it defines the content of what is transferred: the format of the messages, attachments, and so on. There are many different documents that take effect here: RFC 822, RFC 2045, RFC 2046, and RFC 2047. As a user of the JavaMail API, you usually don't need to worry about these formats. However, these formats do exist and are used by your programs.

NNTP and others

Because of the split of the JavaMail API between provider and everything else, you can easily add support for additional protocols. Sun maintains a list of third-party providers that take advantage of protocols for which Sun does not provide out-of-the-box support. You'll find support for NNTP (Network News Transport Protocol) [newsgroups], S/MIME (Secure Multipurpose Internet Mail Extensions), and more.

Section 4. Installing JavaMail

Introduction

There are two versions of the JavaMail API commonly used today: 1.2 and 1.1.3. All the examples in this course will work with both. While 1.2 is the latest, 1.1.3 is the version included with the 1.2.1 version of the Java 2 Platform, Enterprise Edition (J2EE), so it is still commonly used. The version of the JavaMail API you want to use affects what you download and install. All will work with JDK 1.1.6+, Java 2 Platform, Standard Edition (J2SE) version 1.2.x, and J2SE version 1.3.x.

Note: After installing Sun's JavaMail implementation, you can find many example programs in the `demo` directory.

Installing JavaMail 1.2

To use the JavaMail 1.2 API, download the JavaMail 1.2 implementation, unbundle the `javamail-1_2.zip` file, and add the `mail.jar` file to your CLASSPATH. The 1.2 implementation comes with an SMTP, IMAP4, and POP3 provider besides the core classes.

After installing JavaMail 1.2, install the JavaBeans Activation Framework.

Installing JavaMail 1.1.3

To use the JavaMail 1.1.3 API, download the JavaMail 1.1.3 implementation, unbundle the `javamail1_1_3.zip` file, and add the `mail.jar` file to your CLASSPATH. The 1.1.3 implementation comes with an SMTP and IMAP4 provider, besides the core classes.

If you want to access a POP server with JavaMail 1.1.3, download and install a POP3 provider. Sun has one available separate from the JavaMail implementation. After downloading and unbundling `pop31_1_1.zip`, add `pop3.jar` to your CLASSPATH, too.

After installing JavaMail 1.1.3, install the JavaBeans Activation Framework.

Installing the JavaBeans Activation Framework

All versions of the JavaMail API require the JavaBeans Activation Framework. The framework adds support for typing arbitrary blocks of data and handling it accordingly. This doesn't sound like much, but it is your basic MIME-type support found in many browsers and mail tools today. After downloading the framework, unbundle the `jaf1_0_1.zip` file, and add the `activation.jar` file to your CLASSPATH.

For JavaMail 1.2 users, you should now have added `mail.jar` and `activation.jar` to your CLASSPATH.

For JavaMail 1.1.3 users, you should now have added `mail.jar`, `pop3.jar`, and `activation.jar` to your CLASSPATH. If you have no plans of using POP3, you don't need to add `pop3.jar` to your CLASSPATH.

If you don't want to change the CLASSPATH environment variable, copy the jar files to your `lib/ext` directory under the Java Runtime Environment (JRE) directory. For instance, for the J2SE 1.3 release, the default directory would be `C:\jdk1.3\jre\lib\ext` on a Windows platform.

Using JavaMail with the Java 2 Enterprise Edition

If you use J2EE, there is nothing special you have to do to use the basic JavaMail API; it comes with the J2EE classes. Just make sure the `j2ee.jar` file is in your CLASSPATH and you're all set.

For J2EE 1.2.1, the POP3 provider comes separately, so download and follow the steps to include the POP3 provider as shown in the previous section "Installing JavaMail 1.1.3." J2EE 1.3 users get the POP3 provider with J2EE so do not require the separate installation. Neither installation requires you to install the JavaBeans Activation Framework.

Exercise

Exercise 1. How to set up a JavaMail environment

Section 5. Reviewing the core classes

Introduction

Before taking a how-to approach at looking at the JavaMail classes in depth, this section walks you through the core classes that make up the API: `Session`, `Message`, `Address`, `Authenticator`, `Transport`, `Store`, and `Folder`. All these classes are found in the top-level package for the JavaMail API, `javax.mail`, though you'll frequently find yourself using subclasses found in the `javax.mail.internet` package.

Session

The `Session` class defines a basic mail session. It is through this session that everything else works. The `Session` object takes advantage of a `java.util.Properties` object to get information like mail server, username,

password, and other information that can be shared across your entire application.

The constructors for the class are private. You can get a single default session that can be shared with the `getDefaultInstance()` method:

```
Properties props = new Properties();
// fill props with any information
Session session = Session.getDefaultInstance(props, null);
```

Or, you can create a unique session with `getInstance()`:

```
Properties props = new Properties();
// fill props with any information
Session session = Session.getDefaultInstance(props, null);
```

In both cases, the `null` argument is an `Authenticator` object that is not being used at this time.

In most cases, it is sufficient to use the shared session, even if working with mail sessions for multiple user mailboxes. You can add the username and password combination in at a later step in the communication process, keeping everything separate.

Message

Once you have your `Session` object, it is time to move on to creating the message to send. This is done with a type of `Message` . Because `Message` is an abstract class, you must work with a subclass, in most cases `javax.mail.internet.MimeMessage` . A `MimeMessage` is an e-mail message that understands MIME types and headers, as defined in the different RFCs. Message headers are restricted to US-ASCII characters only, though non-ASCII characters can be encoded in certain header fields.

To create a `Message`, pass along the `Session` object to the `MimeMessage` constructor:

```
MimeMessage message = new MimeMessage(session);
```

Note: There are other constructors, like for creating messages from RFC822-formatted input streams.

Once you have your message, you can set its parts, as `Message` implements the `Part` interface (with `MimeMessage` implementing `MimePart`). The basic mechanism to set the content is the `setContent()` method, with arguments for the content and the mime type:

```
message.setContent("Hello", "text/plain");
```

If, however, you know you are working with a `MimeMessage` and your message is plain text, you can use its `setText()` method, which only requires the actual content, defaulting to the MIME type of text/plain:

```
message.setText("Hello");
```

For plain text messages, the latter form is the preferred mechanism to set the content. For sending other kinds of messages, like HTML messages, use the former.

For setting the subject, use the `setSubject()` method:

```
message.setSubject("First");
```

Address

Once you've created the `Session` and the `Message`, as well as filled the message with content, it is time to address your letter with an `Address` . Like `Message`, `Address` is an abstract class. You use the `javax.mail.internet.InternetAddress` class.

To create an address with just the e-mail address, pass the e-mail address to the constructor:

```
Address address = new InternetAddress("president@whitehouse.gov");
```

If you want a name to appear next to the e-mail address, you can pass that along to the constructor, too:

```
Address address = new InternetAddress("president@whitehouse.gov", "George Bush");
```

You will need to create address objects for the message's *from* field as well as the *to* field. Unless your mail server prevents you, there is nothing stopping you from sending a message that appears to be from anyone.

Once you've created the addresses, you connect them to a message in one of two ways. For identifying the sender, you use the `setFrom()` and `setReplyTo()` methods.

```
message.setFrom(address)
```

If your message needs to show multiple *from* addresses, use the `addFrom()` method:

```
Address address[] = ...;
message.addFrom(address);
```

For identifying the message recipients, you use the `addRecipient()` method. This method requires a `Message.RecipientType` besides the address.

```
message.addRecipient(type, address)
```

The three predefined types of address are:

- Message.RecipientType.TO
- Message.RecipientType.CC
- Message.RecipientType.BCC

So, if the message was to go to the vice president, sending a carbon copy to the first lady, the following would be appropriate:

```
Address toAddress = new InternetAddress("vice.president@whitehouse.gov");
Address ccAddress = new InternetAddress("first.lady@whitehouse.gov");
message.addRecipient(Message.RecipientType.TO, toAddress);
message.addRecipient(Message.RecipientType.CC, ccAddress);
```

The JavaMail API provides no mechanism to check for the validity of an e-mail address. While you can program in support to scan for valid characters (as defined by RFC 822) or verify the MX (mail exchange) record yourself, these are all beyond the scope of the JavaMail API.

Authenticator

Like the `java.net` classes, the JavaMail API can take advantage of an `Authenticator` to access protected resources via a username and password. For the JavaMail API, that resource is the mail server. The JavaMail `Authenticator` is found in the `javax.mail` package and is different from the `java.net` class of the same name. The two don't share the same `Authenticator` as the JavaMail API works with Java 1.1, which didn't have the `java.net` variety.

To use the `Authenticator`, you subclass the abstract class and return a `PasswordAuthentication` instance from the `getPasswordAuthentication()` method. You must register the `Authenticator` with the session when created. Then, your `Authenticator` will be notified when authentication is necessary. You could pop up a window or read the username and password from a configuration file (though if not encrypted is not secure), returning them to the caller as a `PasswordAuthentication` object.

```
Properties props = new Properties();
```

```
// fill props with any information
Authenticator auth = new MyAuthenticator();
Session session = Session.getDefaultInstance(props, auth);
```

Transport

The final part of sending a message is to use the `Transport` class. This class speaks the protocol-specific language for sending the message (usually SMTP). It's an abstract class and works something like `Session`. You can use the *default* version of the class by just calling the static `send()` method:

```
Transport.send(message);
```

Or, you can get a specific instance from the session for your protocol, pass along the username and password (blank if unnecessary), send the message, and close the connection:

```
message.saveChanges(); // implicit with send()
Transport transport = session.getTransport("smtp");
transport.connect(host, username, password);
transport.sendMessage(message, message.getAllRecipients());
transport.close();
```

This latter way is best when you need to send multiple messages, as it will keep the connection with the mail server active between messages. The basic `send()` mechanism makes a separate connection to the server for each method call.

Note: To watch the mail commands go by to the mail server, set the debug flag with `session.setDebug(true)`.

Store and folder

Getting messages starts similarly to sending messages with a `Session`. However, after getting the session, you connect to a `Store`, quite possibly with a username and password or `Authenticator`. Like `Transport`, you tell the `Store` what protocol to use:

```
// Store store = session.getStore("imap");
Store store = session.getStore("pop3");
store.connect(host, username, password);
```

After connecting to the `Store`, you can then get a `Folder`, which must be opened before you can read messages from it:

```
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
Message message[] = folder.getMessages();
```

For POP3, the only folder available is the `INBOX`. If you are using IMAP, you can have other folders available.

Note: Sun's providers are meant to be smart. While `Message message[] = folder.getMessages();` might look like a slow operation reading every message from the server, only when you actually need to get a part of the message is the message content retrieved.

Once you have a `Message` to read, you can get its content with `getContent()` or write its content to a stream with `writeTo()`. The `getContent()` method only gets the message content, while `writeTo()` output includes headers.

```
System.out.println(((MimeMessage)message).getContent());
```

Once you're done reading mail, close the connection to the folder and store.

```
folder.close(aBoolean);
store.close();
```

The boolean passed to the `close()` method of folder states whether or not to update the folder by removing deleted messages.

Moving on

Essentially, understanding how to use these seven classes is all you need for nearly everything with the JavaMail API. Most of the other capabilities of the JavaMail API build off these seven classes to do something a little different or in a particular way, like if the content is an attachment. Certain tasks, like searching, are isolated and are discussed later.

Section 6. Using the JavaMail API

Introduction

You've seen how to work with the core parts of the JavaMail API. In the following sections you'll find a how-to approach for connecting the pieces to do specific tasks.

Sending messages

Sending an e-mail message involves getting a session, creating and filling a message, and sending it. You can specify your SMTP server by setting the `mail.smtp.host` property for the `Properties` object passed when getting the `Session`:

```
String host = ...;
String from = ...;
String to = ...;

// Get system properties
Properties props = System.getProperties();

// Setup mail server
props.put("mail.smtp.host", host);

// Get session
Session session = Session.getDefaultInstance(props, null);

// Define message
MimeMessage message = new MimeMessage(session);
message.setFrom(new InternetAddress(from));
message.addRecipient(Message.RecipientType.TO,
  new InternetAddress(to));
  message.setSubject("Hello JavaMail");
  message.setText("Welcome to JavaMail");

  // Send message
  Transport.send(message);
```

You should place the code in a try-catch block, as setting up the message and sending it can throw exceptions.

Exercise:
Exercise 2. How to send your first message

Fetching messages

For reading mail, you get a session, get and connect to an appropriate store for your mailbox, open the appropriate folder, and get your messages. Also, don't forget to close the connection when done.

```
  String host = ...;
  String username = ...;
  String password = ...;

  // Create empty properties
  Properties props = new Properties();

  // Get session
  Session session = Session.getDefaultInstance(props, null);

  // Get the store
  Store store = session.getStore("pop3");
  store.connect(host, username, password);

  // Get folder
  Folder folder = store.getFolder("INBOX");
  folder.open(Folder.READ_ONLY);

  // Get directory
```

```
Message message[] = folder.getMessages();

for (int i=0, n=message.length; i<n; i++) {
   System.out.println(i + ": " + message[i].getFrom()[0]
        + "\t" + message[i].getSubject());
        }

        // Close connection
        folder.close(false);
        store.close();
```

What you do with each message is up to you. The above code block just displays whom the message is from and the subject. Technically speaking, the list of *from* addresses could be empty and the `getFrom()[0]` call could throw an exception.

To display the whole message, you can prompt the user after seeing the from and subject fields, and then call the message's `writeTo()` method if the user wants to see it.

```
        BufferedReader reader = new BufferedReader (
          new InputStreamReader(System.in));

        // Get directory
        Message message[] = folder.getMessages();
        for (int i=0, n=message.length; i<n; i++) {
          System.out.println(i + ": " + message[i].getFrom()[0]
            + "\t" + message[i].getSubject());

          System.out.println("Do you want to read message? " +
            "[YES to read/QUIT to end]");
          String line = reader.readLine();
          if ("YES".equals(line)) {
            message[i].writeTo(System.out);
          } else if ("QUIT".equals(line)) {
            break;
          }
        }
```

Exercise:
Exercise 3. How to check for mail

Deleting messages and flags

Deleting messages involves working with the `Flags` associated with the messages. There are different flags for different states, some system-defined and some user-defined. The predefined flags are defined in the inner class `Flags.Flag` and are listed below:

- `Flags.Flag.ANSWERED`
- `Flags.Flag.DELETED`
- `Flags.Flag.DRAFT`
- `Flags.Flag.FLAGGED`
- `Flags.Flag.RECENT`

- `Flags.Flag.SEEN`
- `Flags.Flag.USER`

Just because a flag exists doesn't mean the flag is supported by all mail servers or providers. For instance, except for deleting messages, the POP protocol supports none of them. Checking for *new* mail is not a POP task but a task built into mail clients. To find out what flags are supported, ask the folder with `getPermanentFlags()`.

To delete messages, you set the message's `DELETED` flag:

```
message.setFlag(Flags.Flag.DELETED, true);
```

Open up the folder in `READ_WRITE` mode first though:

```
folder.open(Folder.READ_WRITE);
```

Then, when you are done processing all messages, close the folder, passing in a true value to *expunge* the deleted messages.

```
folder.close(true);
```

There is an `expunge()` method of `Folder` that can be used to delete the messages. However, it doesn't work for Sun's POP3 provider. Other providers may or may not implement the capabilities. It will more than likely be implemented for IMAP providers. Because POP only supports single access to the mailbox, you have to close the folder to delete the messages with Sun's provider.

To unset a flag, just pass false to the `setFlag()` method. To see if a flag is set, check it with `isSet()`.

Authenticating yourself

You learned that you can use an `Authenticator` to prompt for username and password when needed, instead of passing them in as strings. Here you'll actually see how to more fully use authentication.

Instead of connecting to the `Store` with the host, username, and password, you configure the `Properties` to have the host, and tell the `Session` about your custom `Authenticator` instance, as shown here:

```
// Setup properties
Properties props = System.getProperties();
props.put("mail.pop3.host", host);

// Setup authentication, get session
```

```
Authenticator auth = new PopupAuthenticator();
Session session = Session.getDefaultInstance(props, auth);

// Get the store
Store store = session.getStore("pop3");
store.connect();
```

You then subclass `Authenticator` and return a `PasswordAuthentication` object from the `getPasswordAuthentication()` method. The following is one such implementation, with a single field for both. (This isn't a Project Swing tutorial; just enter the two parts in the one field, separated by a comma.)

```
import javax.mail.*;
import javax.swing.*;
import java.util.*;

public class PopupAuthenticator extends Authenticator {

  public PasswordAuthentication getPasswordAuthentication() {
    String username, password;

    String result = JOptionPane.showInputDialog(
      "Enter 'username,password'");

    StringTokenizer st = new StringTokenizer(result, ",");
    username = st.nextToken();
    password = st.nextToken();

    return new PasswordAuthentication(username, password);
  }

}
```

Because the `PopupAuthenticator` relies on Swing, it will start up the event-handling thread for AWT. This basically requires you to add a call to `System.exit()` in your code to stop the program.

Replying to messages

The `Message` class includes a `reply()` method to configure a new `Message` with the proper recipient and subject, adding "Re: " if not already there. This does not add any content to the message, only copying the *from* or *reply-to* header to the new recipient. The method takes a boolean parameter indicating whether to reply to only the sender (false) or reply to all (true).

```
MimeMessage reply = (MimeMessage)message.reply(false);
reply.setFrom(new InternetAddress("president@whitehouse.gov"));
reply.setText("Thanks");
Transport.send(reply);
```

To configure the *reply-to* address when sending a message, use the `setReplyTo()` method.

Exercise:
Exercise 4. How to reply to mail

Forwarding messages

Forwarding messages is a little more involved. There is no single method to call, and you build up the message to forward by working with the parts that make up a message.

A mail message can be made up of multiple parts. Each part is a `BodyPart`, or more specifically, a `MimeBodyPart` when working with MIME messages. The different body parts get combined into a container called `Multipart` or, again, more specifically a `MimeMultipart`. To forward a message, you create one part for the text of your message and a second part with the message to forward, and combine the two into a multipart. Then you add the multipart to a properly addressed message and send it.

That's essentially it. To copy the content from one message to another, just copy over its `DataHandler`, a class from the JavaBeans Activation Framework.

```
// Create the message to forward
Message forward = new MimeMessage(session);

// Fill in header
forward.setSubject("Fwd: " + message.getSubject());
forward.setFrom(new InternetAddress(from));
forward.addRecipient(Message.RecipientType.TO,
  new InternetAddress(to));

// Create your new message part
BodyPart messageBodyPart = new MimeBodyPart();
messageBodyPart.setText(
  "Here you go with the original message:\n\n");

// Create a multi-part to combine the parts
Multipart multipart = new MimeMultipart();
multipart.addBodyPart(messageBodyPart);

// Create and fill part for the forwarded content
messageBodyPart = new MimeBodyPart();
messageBodyPart.setDataHandler(message.getDataHandler());

// Add part to multi part
multipart.addBodyPart(messageBodyPart);

// Associate multi-part with message
forward.setContent(multipart);

// Send message
Transport.send(forward);
```

Working with attachments

Attachments are resources associated with a mail message, usually kept outside of the message like a text file, spreadsheet, or image. As with common mail programs like Eudora and pine, you can *attach* resources to your mail message with the JavaMail API and get those attachments when you receive the message.

Sending attachments:

Sending attachments is quite like forwarding messages. You build up the parts to make the complete message. After the first part, your message text, you add other parts where the `DataHandler` for each is your attachment, instead of the shared handler in the case of a forwarded message. If you are reading the attachment from a file, your attachment data source is a `FileDataSource` . Reading from a URL, it is a `URLDataSource` . Once you have your `DataSource`, just pass it on to the `DataHandler` constructor, before finally attaching it to the `BodyPart` with `setDataHandler()`. Assuming you want to retain the original filename for the attachment, the last thing to do is to set the filename associated with the attachment with the `setFileName()` method of `BodyPart`. All this is shown here:

```
// Define message
Message message = new MimeMessage(session);
message.setFrom(new InternetAddress(from));
message.addRecipient(Message.RecipientType.TO,
  new InternetAddress(to));
message.setSubject("Hello JavaMail Attachment");

// Create the message part
BodyPart messageBodyPart = new MimeBodyPart();

// Fill the message
messageBodyPart.setText("Pardon Ideas");

Multipart multipart = new MimeMultipart();
multipart.addBodyPart(messageBodyPart);

// Part two is attachment
messageBodyPart = new MimeBodyPart();
DataSource source = new FileDataSource(filename);
messageBodyPart.setDataHandler(new DataHandler(source));
messageBodyPart.setFileName(filename);
multipart.addBodyPart(messageBodyPart);

// Put parts in message
message.setContent(multipart);

// Send the message
Transport.send(message);
```

When including attachments with your messages, if your program is a servlet, your users must upload the attachment besides telling you where to send the message. Uploading each file can be handled with a form encoding type of `multipart/form-data`.

```
<FORM ENCTYPE="multipart/form-data"
    method=post action="/myservlet">
  <INPUT TYPE="file" NAME="thefile">
  <INPUT TYPE="submit" VALUE="Upload">
</FORM>
```

Note: Message size is limited by your SMTP server, not the JavaMail API. If you run into problems, consider increasing the Java heap size by setting the ms and mx parameters.

Exercise:
Exercise 5. How to send attachments

Getting attachments:
Getting attachments out of your messages is a little more involved then sending them because MIME has no simple notion of attachments. The content of your message is a `Multipart` object when it has attachments. You then need to process each `Part`, to get the main content and the attachment(s). Parts marked with a disposition of `Part.ATTACHMENT` from `part.getDisposition()` are clearly attachments. However, attachments can also come across with no disposition (and a non-text MIME type) or a disposition of `Part.INLINE`. When the disposition is either `Part.ATTACHMENT` or `Part.INLINE`, you can save off the content for that message part. Just get the original filename with `getFileName()` and the input stream with `getInputStream()`.

```
Multipart mp = (Multipart)message.getContent();

for (int i=0, n=multipart.getCount(); i<n; i++) {
  Part part = multipart.getBodyPart(i));

  String disposition = part.getDisposition();

  if ((disposition != null) &&
       ((disposition.equals(Part.ATTACHMENT) ||
          (disposition.equals(Part.INLINE))) {
    saveFile(part.getFileName(), part.getInputStream());
  }
}
```

The `saveFile()` method just creates a `File` from the filename, reads the bytes from the input stream, and writes them off to the file. In case the file already exists, a number is added to the end of the filename until one is found that doesn't exist.

```
// from saveFile()
File file = new File(filename);
for (int i=0; file.exists(); i++) {
  file = new File(filename+i);
}
```

The code above covers the simplest case where message parts are flagged appropriately. To cover all cases, handle when the disposition is null and get the MIME type of the part to handle accordingly.

```
if (disposition == null) {
  // Check if plain
  MimeBodyPart mbp = (MimeBodyPart)part;
  if (mbp.isMimeType("text/plain")) {
     // Handle plain
  } else {
     // Special non-attachment cases here of image/gif, text/html, ...
  }
  ...
}
```

Processing HTML messages

Sending HTML-based messages can be a little more work than sending plain text

message, though it doesn't have to be that much more work. It all depends on your specific requirements.

Sending HTML messages:
If all you need to do is send the equivalent of an HTML file as the message and let the mail reader worry about fetching any embedded images or related pieces, use the `setContent()` method of `Message`, passing along the content as a `String` and setting the content type to `text/html`.

```
String htmlText = "<H1>Hello</H1>" +
  "<img src=\"http://www.jguru.com/images/logo.gif\">";
message.setContent(htmlText, "text/html"));
```

On the receiving end, if you fetch the message with the JavaMail API, there is nothing built into the API to display the message as HTML. The JavaMail API only sees it as a stream of bytes. To display the message as HTML, you must either use the Swing `JEditorPane` or some third-party HTML viewer component.

```
if (message.getContentType().equals("text/html")) {
    String content = (String)message.getContent();
    JFrame frame = new JFrame();
    JEditorPane text = new JEditorPane("text/html", content);
    text.setEditable(false);
    JScrollPane pane = new JScrollPane(text);
    frame.getContentPane().add(pane);
    frame.setSize(300, 300);
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    frame.show();
}
```

Including images with your messages:
On the other hand, if you want your HTML content message to be complete, with embedded images included as part of the message, you must treat the image as an attachment and reference the image with a special `cid` URL, where the `cid` is a reference to the `Content-ID` header of the image attachment.

The process of embedding an image is quite similar to attaching a file to a message, the only difference is you have to tell the `MimeMultipart` that the parts are related by setting its subtype in the constructor (or with `setSubType()`) and set the `Content-ID` header for the image to a random string which is used as the `src` for the image in the `img` tag. The following demonstrates this completely.

```
String file = ...;

// Create the message
Message message = new MimeMessage(session);

// Fill its headers
message.setSubject("Embedded Image");
message.setFrom(new InternetAddress(from));
message.addRecipient(Message.RecipientType.TO,
  new InternetAddress(to));

// Create your new message part
BodyPart messageBodyPart = new MimeBodyPart();
String htmlText = "<H1>Hello</H1>" +
```

```
  "<img src=\"cid:memememe\">";
messageBodyPart.setContent(htmlText, "text/html");

// Create a related multi-part to combine the parts
MimeMultipart multipart = new MimeMultipart("related");
multipart.addBodyPart(messageBodyPart);

// Create part for the image
messageBodyPart = new MimeBodyPart();

// Fetch the image and associate to part
DataSource fds = new FileDataSource(file);
messageBodyPart.setDataHandler(new DataHandler(fds));
messageBodyPart.setHeader("Content-ID","memememe");

// Add part to multi-part
multipart.addBodyPart(messageBodyPart);

// Associate multi-part with message
message.setContent(multipart);
```

Exercise:
Exercise 6. How to send HTML messages with images

Section 7. Searching with SearchTerm

Introduction

The JavaMail API includes a filtering mechanism found in the `javax.mail.search` package to build up a `SearchTerm` . Once built, you then ask a `Folder` what messages match, retrieving an array of `Message` objects:

```
SearchTerm st = ...;
Message[] msgs = folder.search(st);
```

There are 22 different classes available to help you build a search term.

- AND terms (class `AndTerm`)
- OR terms (class `OrTerm`)
- NOT terms (class `NotTerm`)
- SENT DATE terms (class `SentDateTerm`)
- CONTENT terms (class `BodyTerm`)
- HEADER terms (`FromTerm` / `FromStringTerm`, `RecipientTerm` / `RecipientStringTerm`, `SubjectTerm`, etc..)

Essentially, you build up a logical expression for matching messages, then search. For instance the following term searches for messages with a (partial) subject string

of ADV or a *from* field of `friend@public.com`. You might consider periodically running this query and automatically deleting any messages returned.

```
SearchTerm st =
  new OrTerm(
      new SubjectTerm("ADV:"),
      new FromStringTerm("friend@public.com"));
Message[] msgs = folder.search(st);
```

Section 8. Exercises

About the exercises

These exercises are designed to provide help according to your needs. For example, you might simply complete the exercise given the information and the task list in the exercise body; you might want a few hints; or you may want a step-by-step guide to successfully complete a particular exercise. You can use as much or as little help as you need per exercise. Moreover, because complete solutions are also provided, you can skip a few exercises and still be able to complete future exercises requiring the skipped ones.

Each exercise has a list of any prerequisite exercises, a list of skeleton code for you to start with, links to necessary API pages, and a text description of the exercise goal. In addition, there is help for each task and a solutions page with links to files that comprise a solution to the exercise.

Exercise 1. How to set up a JavaMail environment

In this exercise you will install Sun's JavaMail reference implementation. After installing, you will be introduced to the demonstration programs that come with the reference implementation.

Task 1:
Download the latest version of the JavaMail API implementation from Sun.

Task 2:
Download the latest version of the JavaBeans Activation Framework from Sun.

Task 3:
Unzip the downloaded packages. You get a ZIP file for all platforms for both packages.

> **Help for task 3:**
> You can use the `jar` tool to unzip the packages.

Task 4:
Add the `mail.jar` file from the JavaMail 1.2 download and the `activation.jar` file from the JavaBeans Activation Framework download to your `CLASSPATH`.

Help for task 4:
Copy the files to your extension library directory. For Microsoft Windows, using the default installation copy, the command might look like the following:

```
cd \javamail-1.2
copy mail.jar \jdk1.3\jre\lib\ext
cd \jaf-1.0.1
copy activation.jar \jdk1.3\jre\lib\ext
```

If you don't like copying the files to the extension library directory, detailed instructions are available from Sun for setting your CLASSPATH on Windows NT.

Task 5:
Go into the `demo` directory that comes with the JavaMail API implementation and compile the `msgsend` program to send a test message.

Help for task 5:

```
javac msgsend.java
```

Task 6:
Execute the program passing in a *from* address with the `-o` option, your SMTP server with the `-M` option, and the *to* address (with no option). You'll then enter the subject, the text of your message, and the end-of-file character (CTRL-Z) to signal the end of the message input.

Help for task 6:
Be sure to replace the *from* address, SMTP server, and *to* address.

```
java msgsend -o from@address -M SMTP.Server to@address
```

If you are not sure of your SMTP server, contact your system administrator or check with your Internet Service Provider.

Task 7:
Check to make sure you received the message with your normal mail reader (Eudora, Outlook Express, pine, ...).

Exercise 1. How to set up a JavaMail environment: Solution

Upon successful completion, the JavaMail reference implementation will be in your CLASSPATH.

Exercise 2. How to send your first message

In the last exercise you sent a mail message using the demonstration program provided with the JavaMail implementation. In this exercise, you'll create the program yourself.

For more help with exercises, see About the exercises.

Prerequisites:

- Exercise 1. How to set up a JavaMail environment

Skeleton code:

- MailExample.java

Task 1:
Starting with the skeleton code, get the system `Properties`.

Help for task 1:

```
Properties props = System.getProperties();
```

Task 2:
Add the name of your SMTP server to the properties for the `mail.smtp.host` key.

Help for task 2:

```
props.put("mail.smtp.host", host);
```

Task 3:
Get a `Session` object based on the `Properties`.

Help for task 3:

```
Session session = Session.getDefaultInstance(props, null);
```

Task 4:
Create a `MimeMessage` from the session.

Help for task 4:

```
MimeMessage message = new MimeMessage(session);
```

Task 5:

Set the *from* field of the message.

Help for task 5:

```
message.setFrom(new InternetAddress(from));
```

Task 6:
Set the *to* field of the message.

Help for task 6:

```
message.addRecipient(Message.RecipientType.TO,
      new InternetAddress(to));
```

Task 7:
Set the subject of the message.

Help for task 7:

```
message.setSubject("Hello JavaMail");
```

Task 8:
Set the content of the message.

Help for task 8:

```
message.setText("Welcome to JavaMail");
```

Task 9:
Use a `Transport` to send the message.

Help for task 9:

```
Transport.send(message);
```

Task 10:
Compile and run the program, passing your SMTP server, *from* address, and *to* address on the command line.

Help for task 10:

```
java MailExample SMTP.Server from@address to@address
```

Task 11:
Check to make sure you received the message with your normal mail reader

(Eudora, Outlook Express, pine, ...).

Exercise 2. How to send your first message: Solution

The following Java source file represents a solution to this exercise:

- Solution/MailExample.java

Exercise 3. How to check for mail

In this exercise, create a program that displays the *from* address and subject for each message and prompts to display the message content.

For more help with exercises, see About the exercises.

Prerequisites:

- Exercise 1. How to set up a JavaMail environment

Skeleton Code

- GetMessageExample.java

Task 1:
Starting with the skeleton code, get or create a `Properties` object.

Help for task 1:

```
Properties props = new Properties();
```

Task 2:
Get a `Session` object based on the `Properties`.

Help for task 2:

```
Session session = Session.getDefaultInstance(props, null);
```

Task 3:
Get a `Store` for your e-mail protocol, either `pop3` or `imap`.

Help for task 3:

```
Store store = session.getStore("pop3");
```

Task 4:

Connect to your mail host's store with the appropriate username and password.

Help for task 4:

```
store.connect(host, username, password);
```

Task 5:
Get the folder you want to read. More than likely, this will be the `INBOX`.

Help for task 5:

```
Folder folder = store.getFolder("INBOX");
```

Task 6:
Open the folder read-only.

Help for task 6:

```
folder.open(Folder.READ_ONLY);
```

Task 7:
Get a directory of the messages in the folder. Save the message list in an array variable named `message`.

Help for task 7:

```
Message message[] = folder.getMessages();
```

Task 8:
For each message, display the *from* field and the subject.

Help for task 8:

```
System.out.println(i + ": " + message[i].getFrom()[0]
  + "\t" + message[i].getSubject());
```

Task 9:
Display the message content when prompted.

Help for task 9:

```
System.out.println(message[i].getContent());
```

Task 10:
Close the connection to the folder and store.

Help for task 10:

```
folder.close(false);
store.close();
```

Task 11:
Compile and run the program, passing your mail server, username, and password on the command line. Answer YES to the messages you want to read. Just hit ENTER if you don't. If you want to stop reading your mail before making your way through all the messages, enter QUIT.

Help for task 11:

```
java GetMessageExample POP.Server username password
```

Exercise 3. How to check for mail: Solution

The following Java source file represents a solution to this exercise.

- Solution/GetMessageExample.java

Exercise 4. How to reply to mail

In this exercise, create a program that creates a canned reply message and attaches the original message if it's plain text.

For more help with exercises, see About the exercises.

Prerequisites:

- Exercise 3. How to check for mail

Skeleton Code:

- ReplyExample.java

Task 1:
The skeleton code already includes the code to get the list of messages from the folder and prompt you to create a reply.

Task 2:
When answered affirmatively, create a new `MimeMessage` from the original message.

Help for task 2:

```

```

```
MimeMessage reply = (MimeMessage)message[i].reply(false);
```

Task 3:
Set the *from* field to your e-mail address.

Task 4:
Create the text for the reply. Include a canned message to start. When the original message is plain text, add each line of the original message, prefix each line with the "> " characters.

> **Help for task 4:**
> To check for plain text messages, check the messages MIME type with `mimeMessage.isMimeType("text/plain")`.

Task 5:
Set the message's content, once the message content is fully determined.

Task 6:
Send the message.

Task 7:
Compile and run the program, passing your mail server, SMTP server, username, password, and *from* address on the command line. Answer YES to the messages you want to send replies. Just hit ENTER if you don't. If you want to stop going through your mail before making your way through all the messages, enter QUIT.

> **Help for task 7:**

```
java ReplyExample POP.Server SMTP.Server username password from@address
```

Task 8:
Check to make sure you received the message with your normal mail reader (Eudora, Outlook Express, pine, ...).

Exercise 4. How to reply to mail: Solution

The following Java source file represents a solution to this exercise.

- Solution/ReplyExample.java

Exercise 5. How to send attachments

In this exercise, create a program that sends a message with an attachment.

For more help with exercises, see About the exercises.

Prerequisites:

- Exercise 2. How to send your first message

Skeleton Code:

- AttachExample.java

Task 1:
The skeleton code already includes the code to get the initial mail session.

Task 2:
From the session, get a `Message` and set its header fields: to, from, and subject.

Task 3:
Create a `BodyPart` for the main message content and fill its content with the text of the message.

Help for task 3:

```
BodyPart messageBodyPart = new MimeBodyPart();
messageBodyPart.setText("Here's the file");
```

Task 4:
Create a `Multipart` to combine the main content with the attachment. Add the main content to the multipart.

Help for task 4:

```
Multipart multipart = new MimeMultipart();
multipart.addBodyPart(messageBodyPart);
```

Task 5:
Create a second `BodyPart` for the attachment.

Task 6:
Get the attachment as a `DataSource`.

Help for task 6:

```
DataSource source = new FileDataSource(filename);
```

Task 7:
Set the `DataHandler` for the message part to the data source. Carry the original filename along.

Help for task 7:

```
messageBodyPart.setDataHandler(new DataHandler(source));
messageBodyPart.setFileName(filename);
```

Task 8:
Add the second part of the message to the multipart.

Task 9:
Set the content of the message to the multipart.

Help for task 9:

```
message.setContent(multipart);
```

Task 10:
Send the message.

Task 11:
Compile and run the program, passing your SMTP server, *from* address, *to* address, and filename on the command line. This will send the file as an attachment.

Help for task 11:

```
java AttachExample SMTP.Server from@address to@address filename
```

Task 12:
Check to make sure you received the message with your normal mail reader (Eudora, Outlook Express, pine, ...).

Exercise 5. How to send attachments: Solution

The following Java source file represents a solution to this exercise.

- Solution/AttachExample.java

Exercise 6. How to send HTML messages with images

In this exercise, create a program that sends an HTML message with an image attachment where the image is displayed within the HTML message.

For more help with exercises, see About the exercises.

Prerequisites:

- Exercise 5. How to send attachments

Skeleton code:

- logo.gif
- HtmlImageExample.java

Task 1:
The skeleton code already includes the code to get the initial mail session, create the main message, and fill its headers (to, from, subject).

Task 2:
Create a `BodyPart` for the HTML message content.

Task 3:
Create a text string of the HTML content. Include a reference in the HTML to an image (<img src="...">) that is local to the mail message.

Help for task 3:
Use a `cid` URL. The content-id will need to be specified for the image later.

```
String htmlText = "<H1>Hello</H1>" +
  "<img src=\"cid:memememe\">";
```

Task 4:
Set the content of the message part. Be sure to specify the MIME type is `text/html`.

Help for task 4:

```
messageBodyPart.setContent(htmlText, "text/html");
```

Task 5:
Create a `Multipart` to combine the main content with the attachment. Be sure to specify that the parts are related. Add the main content to the multipart.

Help for task 5:

```
MimeMultipart multipart = new MimeMultipart("related");
multipart.addBodyPart(messageBodyPart);
```

Task 6:
Create a second `BodyPart` for the attachment.

Task 7:
Get the attachment as a `DataSource`, and set the `DataHandler` for the message part to the data source.

Task 8:
Set the `Content-ID` header for the part to match the image reference specified in the HTML.

Help for task 8:

```
messageBodyPart.setHeader("Content-ID","memememe");
```

Task 9:
Add the second part of the message to the multipart, and set the content of the message to the multipart.

Task 10:
Send the message.

Task 11:
Compile and run the program, passing your SMTP server, *from* address, *to* address, and filename on the command line. This will send the images as an inline image within the HTML text.

Help for task 11:

```
java HtmlImageExample SMTP.Server from@address to@address filename
```

Task 12:
Check if your mail reader recognizes the message as HTML and displays the image within the message, instead of as a link to an external attachment file.

Help for task 12:
If your mail reader can't display HTML messages, consider sending the message to a friend.

Exercise 6. How to send HTML messages with images: Solution

The following Java source files represent a solution to this exercise.

- Solution/logo.gif
- Solution/HtmlImageExample.java

Section 9. Wrapup

In summary

The JavaMail API is a Java package used for reading, composing, and sending e-mail messages and their attachments. It lets you build standards-based e-mail clients that employ various Internet mail protocols, including SMTP, POP, IMAP, and MIME, as well as related protocols such as NNTP, S/MIME, and others.

The API divides naturally into two parts. The first focuses on sending, receiving, and managing messages independent of the protocol used, whereas the second focuses on specific use of the protocols. The purpose of this tutorial was to show how to use the first part of the API, without attempting to deal with protocol providers.

The core JavaMail API consists of seven classes -- `Session`, `Message`, `Address`, `Authenticator`, `Transport`, `Store`, and `Folder` -- all of which are found in `javax.mail`, the top-level package for the JavaMail API. We used these classes to work through a number of common e-mail-related tasks, including sending messages, retrieving messages, deleting messages, authenticating, replying to messages, forwarding messages, managing attachments, processing HTML-based messages, and searching or filtering mail lists.

Finally, we provided a number of step-by-step exercises to help illustrate the concepts presented. Hopefully, this will help you add e-mail functionality to your platform-independent Java applications.

Resources

Learn

- Sun's JavaMail FAQ addresses the use of JavaMail in applets and servlets, as well as prototol-specific questions.
- Tutorial author John Zukowski maintains jGuru's JavaMail FAQ.
- Want to see how others are using JavaMail? Check out Sun's list of third-party products.
- Benoit Marchal shows how to use Java and XML to produce plain text and HTML newsletters in this two-part series, "Managing e-zines with JavaMail and XSLT" Part 1 (developerWorks, March 2001) and Part 2 (developerWorks, April 2001).

Get products and technologies

- Download the JavaMail 1.2 API from the JavaMail API home page.
- The JavaBeans Activation Framework is required for versions 1.2 and 1.1.3 of the JavaMail API.

Discuss

- Participate in the discussion forum for this content.
- The JavaMail-interest mailing list is a Sun-hosted discussion forum for developers.

About the author

John Zukowski
Formerly with jGuru.com, John Zukowski does strategic Java consulting for JZ Ventures, Inc. His latest book is titled Java Collections from Apress.