JavaMail™ Guide for Service Providers

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300   fax 650 969-9131

Revision 01, August 1998

JavaMail Guide for Service Providers

Please Recycle

Adobe PostScript™

August 1998

Contents

Chapter 1: Introduction 1

Chapter 2: Messages 3
- The Structure of a Message 3
  - Simple Messages 4
  - Multipart Messages 4
- Messages and the JavaBeans Activation Framework 5
  - The DataSource 6
  - The DataContentHandler 6
- Message Subclasses 7
  - Creating a Message Subclass 7
    - Message Attributes 7
    - Setting Message Content 8
    - Accessing Message Content 8
  - Creating a `MimeMessage` Subclass 9
    - Creating the Subclass 9
    - Headers 10
    - Content 12
    - Special Cases: Protocols that Provide Preparsed Data 13

Chapter 3: Message Storage and Retrieval 15
- Store 15
  - Authentication 15
    - The `protocolConnect` Method 16
    - The `connect` Method 16
  - Folder Retrieval 16
- Folders 17
  - Folder Naming 18
  - Folder State 18
  - Messages Within a Folder 19
    - Getting Messages 20
    - Searching Messages 21
    - Getting Message Data in Bulk 22
  - Folder Management 23
    - Appending and Copying Messages 23

Expunging Messages 23
Handling Message Flags 24

Chapter 4: Message Transport 25
Transport 25
The `sendMessage` Method 25
The `protocolConnect` Method 26
Address 27

Chapter 5: Events 29

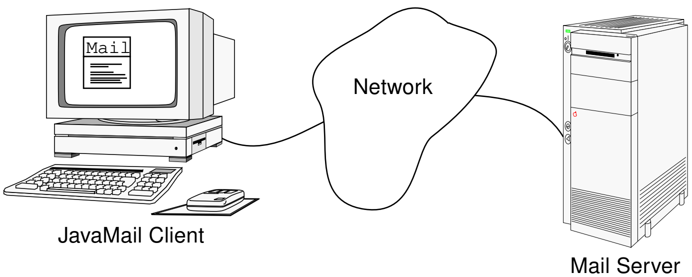
Chapter 6: Packaging 31

Chapter 1:

Introduction

JavaMail provides a common, uniform API for managing electronic mail. It allows service-providers to provide a standard interface to their standards-based or proprietary messaging systems using the Java programming language. Using this API, applications access message stores, and compose and send messages.

FIGURE 1-1



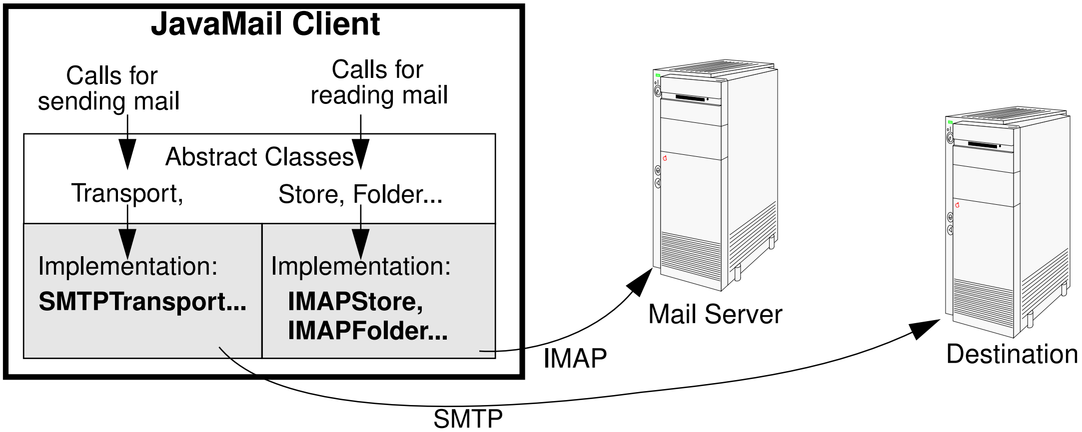
The JavaMail API is composed of a set of abstract classes that model the various pieces of a typical mail system. These classes include,

- `Message`—Abstract class that represents an electronic mail message. JavaMail implements the RFC822 and MIME Internet messaging standards. The `MimeMessage` class extends `Message` to represent a MIME-style email message.
- `Store`—Abstract class that represents a database of messages maintained by a mail server and grouped by owner. A `Store` uses a particular access protocol.
- `Folder`—Abstract class that provides a way of hierarchically organizing messages. `Folders` can contain messages and other folders. A mail server provides each user with a default folder, and users can typically create and fill subfolders.
- `Transport`—Abstract class that represents a specific transport protocol. A `Transport` object uses a particular transport protocol to send a message.

As a service provider, you implement abstract classes in terms of your specific protocol or system. For example, an IMAP provider implements the JavaMail API using the IMAP4 protocol. Clients then use your implementation to manipulate their electronic mail.

The following figure shows a client using an IMAP4 implementation to read mail, and an SMTP implementation to send mail. (They can be from the same or different vendors.)

FIGURE 1-2



This Service Provider's Guide shows you how to develop and package a JavaMail service provider for your clients. It is meant to be used in conjunction with the Javadoc provided with the JavaMail API and the JavaMail API Specification.

This guide covers:

- Creating messages
- Storing and retrieving messages
- Sending a message
- Communicating with a client (for example, notifying the client of new mail)
- Packaging your implementation

The descriptions of the first three tasks show how to subclass the appropriate abstract classes and implement their abstract methods. In addition, the task descriptions point out the methods that have default implementations that you might choose to override for the sake of efficiency.

Chapter 2: Messages

Messages are central to any electronic mail system. This chapter discusses how to implement them for your provider.

If your provider allows only for the common case of creating and sending MIME style messages, then your provider can use the pre-written JavaMail message implementation: the `javax.mail.internet.MimeMessage` class. Implementations that furnish a protocol like `SMTP` fall into this category.

If your implementation does not fall into the previous category, you will have to implement your own `Message` subclass. This chapter

- Explains the structure of a `Message` object
- Explains how `Message` objects use the JavaBeans™ Activation Framework
- Shows you how to develop a `Message` subclass

The Structure of a Message

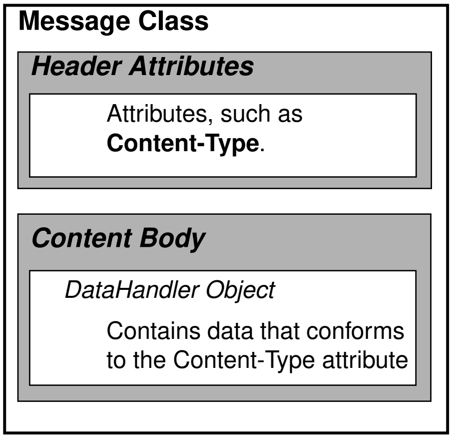
The `Message` class models an electronic mail message. It is an abstract class that implements the `Part` interface.

The `Message` class defines a set of attributes and content for an electronic mail message. The attributes, which are name-value pairs, specify addressing information and define the structure of the message's content (its content type). Messages can contain a single content object or, indirectly, multiple content objects. In either case, the content is held by a `DataHandler` object.

Simple Messages

A simple message has a single content object, which is wrapped by a `DataHandler` object. The following figure shows the structure of a `Message` object:

FIGURE 2-1 Structure of a Simple Message



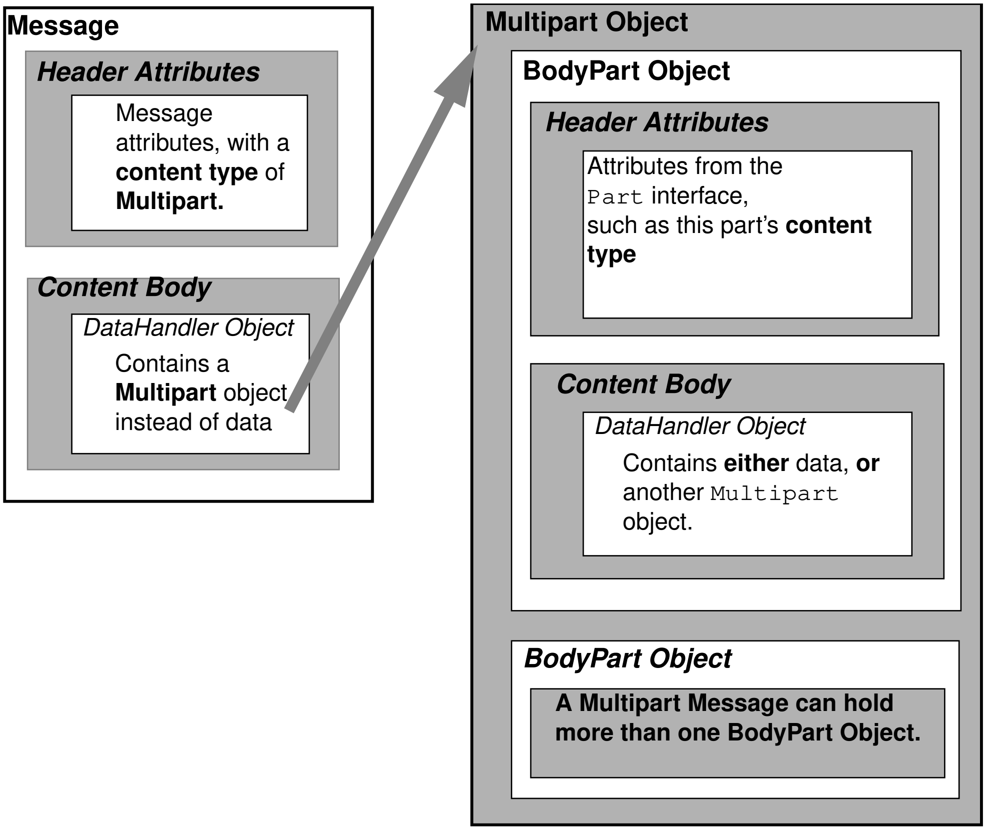
Multipart Messages

In addition to the simple structure shown above, messages can also contain multiple content objects. In this case the `DataHandler` object contains a `Multipart` object, instead of merely a a single block of content data.

A `Multipart` object is a container of `BodyPart` objects. The structure of a `BodyPart` object is similar to the structure of a `Message` object, because they both implement the `Part` interface.

Each `BodyPart` object contains attributes and content, but the attributes of a `Bodypart` object are limited to those defined by the `Part` interface. An important attribute is the content-type of this part of the message content. The content of a `BodyPart` object is a `DataHandler` that contains either data or another `Multipart` object. The following figure shows this structure:

FIGURE 2-2 Structure of a Message with Multiple Content Types



Messages and the JavaBeans Activation Framework

As shown in FIGURE 2-1 on page 4, the content of a message is represented by a `DataHandler` object. The `DataHandler` class is part of the JavaBeans Activation Framework (JAF). Documentation on the JAF can be obtained from the world-wide web at
`http://java.sun.com/beans/glasgow/jaf.html`.

The `DataHandler` class provides a consistent interface to data, independent of its source and format. The data can be from message stores, local files, URLs or objects in the Java programming language.

The DataSource

A DataHandler object accepts data in the form of an object in the Java programming language directly. For data from message stores, files or URLs, however, a `DataHandler` depends on objects that implement the `DataSource` interface to provide data access. A `DataSource` object provides access to data in the form of an input stream.The `DataSource` interface is also part of the JAF. JavaMail provides the following `DataSource` objects:

- `javax.mail.MultipartDataSource`
- `javax.mail.internet.MimePartDataSource`

The DataContentHandler

`DataHandler` objects return the content of a message as an object in the Java programming language. They use objects that implement the `DataContentHandler` interface to translate message content between the streams provided by `DataSource` objects and objects in the Java programming language. The `getContent` and `writeTo` methods of the text/plain `DataContentHandler` show this:

```
public class text_plain implements DataContentHandler {
  // This method creates a String from a "text/plain"
  // data source
  public Object getContent(DataSource dataSource) {
    InputStream inputStream = dataSource.getInputStream();
    ByteArrayOutputStream outputStream =
      new ByteArrayOutputStream();
    int c;

    while ((c = inputStream.read()) != -1)
      outputStream.write(c)
    // get the character set from the content-type
    String charset = getCharSet(dataSource.getContentType());
    return new String(outputStream.toByteArray(), charset);
  }

  // This method creates a byte stream from a String
  public void writeTo(Object object, String type,
                      OutputStream outputStream) {
    OutputStreamWriter writer =
      new OutputStreamWriter(outputStream, getCharset(type));
    String string = (String)object;
    writer.write(string, 0, string.length());
    writer.flush();
  }
}
```

DataContentHandlers are also part of the JAF. The JavaMail implementation in the com.sun.mail.handlers package includes DataContentHandlers for the following two MIME types:

- multipart/mixed (the name of the class is multipart_mixed)
- text/plain (the name of the class is text_plain)

A DataHandler typically finds the correct DataContentHandler for a particular MIME type through the MailCap registry. (The client programmer can also provide the correspondence programmatically.)

Message Subclasses

The following factors determine the message class of your provider:

- If applications will use your provider for interacting with a non-MIME messaging system, create a subclass of the Message class (See "Creating a Message Subclass" on page 7.)
- If applications will use your provider to interact with a message store that supports MIME messages, create a subclass of the MimeMessage class. (See "Creating a MimeMessage Subclass" on page 9.)
- If applications will use your provider to send MIME messages then use the MimeMessage class without subclassing it.

Creating a Message Subclass

When you subclass the Message class, you must implement methods that manage attributes, that retrieve content, and that set content.

Message Attributes

Your implementation is expected to support the attributes in the Message class and its Part interface by implementing their get and set methods. If your messaging system does not allow the modification of an attribute, have the method that sets it throw the IllegalWriteException.

In addition to supporting the predefined attributes, you can also expose attributes specific to your implementation. To make a system-specific attribute available in your subclass, add a field that represents the attribute and provide accessor methods for it.

Setting Message Content

The `Message` class provides a number of abstract methods for setting message content. These will be used by clients preparing an outgoing message.

Some methods take message data directly, and expect your implementation to wrap the data in a `DataHandler` object:

```
public void setContent(java.lang.Object obj,
                       java.lang.String type)
public void setText(java.lang.String text)
```

To wrap the data, use the `DataHandler` constructor that requires an object and a data type. You can then call the same method that clients call when they have wrapped their data in a `DataHandler` object themselves:

```
public void setDataHandler(javax.activation.DataHandler dh)
```

This method is abstract.

Accessing Message Content

The `Message` class provides three methods for getting the message content:

- `public javax.activation.DataHandler getDataHandler()`
- `public java.lang.Object getContent()`
- `public java.io.InputStream getInputStream()`

They are used by clients to get a message from a folder. To implement these methods:

1. **Optional: provide a cache for the `DataHandler` object**
   Caching the `DataHandler` can improve performance if it reduces the number of times you must access the store.

```
public MyMessage extends Message {
  // field for caching the data handler
  private DataHandler dh;
  ...
}
```

2. **Implement the abstract `getDataHandler` method**
   Return the appropriate `DataHandler` object. For example:

```
public MyMessage extends Message {
  ...
  public DataHandler getDataHandler() throws MessagingException {
    if (dh == null)
      dh = new DataHandler(new SomeDataSource(this));
    return dh;
  }
}
```

   Note that you must provide a `DataSource` object to the `DataHandler`. For example, the `MimeMessage` subclass uses the `MimePartDataSource` class.

3. **Have the `getInputStream` and `getContent` methods delegate to the JAF.**
   Implement the `getInputStream` and `getContent` methods to call the corresponding `DataHandler` methods.
   For example:

```
public InputStream getInputStream() throws IOException {
  return getDataHandler().getInputStream();
}
public Object getContent() throws IOException {
  return getDataHandler().getContent();
}
```

Using the techniques described above ensures that you make proper use of the JAF. The `javax.mail.internet` package is implemented this way.

Creating a `MimeMessage` *Subclass*

The `javax.mail.internet` package provides a complete implementation of the internet standards that define the structure of an email message: RFC822 and MIME (RFC2045 - 2047). It defines a subclass of the `Message` class called `MimeMessage`. The `MimeMessage` class adds:

1. Methods to get and set MIME-specific attributes.
2. The ability to parse a MIME-style input stream into its header and content.
3. The ability to generate a MIME-style bytestream.

This section explains enough about the `MimeMessage` class and the `javax.mail.internet` package to enable you to implement a subclass of `MimeMessage`. The upcoming sections use a POP3 implementation, `POP3Message`, as an example.

Creating the Subclass

A newly created message, when it represents a message from a message store, should be a lightweight object that is filled with data only as that data is required. Your constructor, therefore, should create a message object that does not immediately load its data. For example:

```
public class POP3Message extends MimeMessage {
    // Keep track of whether data has been loaded
    boolean loaded = false;
     ...
    public POP3Message(POP3Folder folder, int messageNumber) {
        // This MimeMessage constructor returns an empty message object
        super(folder, messageNumber);
    }
    ...
}
```

Objects that return messages, such as folders, can use this constructor. For example, the folder class that is part of the POP3 service provider, `POP3Folder`, could get a particular POP3 message:

```
public Message getMessage(int messageNumber) {
  POP3Message message;
  ...
    message = new POP3Message(this, messageNumber);
  ...
  return message;
}
```

The next sections, which discuss how to manage Message headers and content, describe a way to load the message's data on demand.

Headers

The `MimeMessage` constructor holds its headers in a `javax.mail.internet.InternetHeaders` object. This object, when constructed with an `InputStream`, reads lines from the stream until it reaches the blank line that indicates end of header. It stores the lines as RFC822 header-fields. After the `InternetHeaders` object reads from the input stream, the stream is positioned at the start of the message body.

The POP3 implementation uses this constructor to load the message headers when one is requested:

```
public class POP3Message extends MimeMessage {
  //Keep track of whether the Message data has been loaded
  boolean loaded = false;
  int hdrSize;
  ...
  public String[] getHeader(String name) {
    //Get the headers on demand from the message store
    load();
    // Don't need to reimplement getting the header object's contents
    return super.getHeader(name);
  }

  /*** Reimplement all variants of getHeader() as above ***/
  ...

  private synchronized void load() {
    if (!loaded) {
      // Get message data (headers and content) and cache it
      content = POP3Command("RETR", msgno);
      // Open a stream to the cache
      InputStream is = new ByteArrayInputStream(content);
      // Setup "headers" field by getting the header data
      headers = new InternetHeaders(is);
      // Save header size to easily access msg content from cache
      hdrSize = content.length - is.available();
      loaded = true;
```

```
        }
    }
}
```

Internationalization of Headers

RFC 822 allows only 7bit US-ASCII characters in email headers. MIME (RFC 2047) defines techniques to encode non-ASCII text in various portions of a RFC822 header, so that such text can be safely transmitted across the internet. These encoding techniques convert non-ASCII characters into sequences of ASCII characters. At the receiving end these characters must be decoded to recreate the original text.

The RFCs specify which standard headers allow such encoding. For example, the `Subject` header permits encoded characters, but the `Content-ID` header does not.

The `MimeMessage.getHeader` methods obtain the named header from the `InternetHeaders` object without decoding it; they return raw data. Similarly, the `MimeMessage.setHeader` method sets the named header without encoding it. The specialized methods such as `getSubject`, `setSubject`, `getDescription` and `setDescription` do apply MIME and RFC822 semantics to the header value.

If your `MimeMessage` subclass adds new headers that require encoding and decoding, your implementation of those headers' supporting get and set methods is responsible for doing this. The MimeUtility class provides a variety of static methods to help you, such as the `decodeText` and `encodeText` methods.

The `decodeText` method decodes a raw header and returns its value as a Unicode string. An example of its use:

```
public String getMyHeader() throws MessagingException {
    String rawvalue = getHeader("MyHeader", null);
    try {
        return MimeUtility.decodeText(rawvalue);
    } catch (UnsupportedEncodingException ex) {
        return rawvalue;
    }
}
```

The `encodeText` method encodes a raw header and sets its value as a Unicode string. An example of its use:

```
public void setMyHeader(String rawHeader) throws MessagingException
{
        try {
            setHeader("MyHeader",
                      MimeUtility.encodeText(rawHeader);
    } catch (UnsupportedEncodingException uex) {
        throw new MessagingException("Encoding error", uex);
    }
}
```

Content

The `getDataHandler` method returns a `DataHandler` object that wraps (contains) the message's content data. As shown in the discussion on how to "Implement the abstract getDataHandler method" on page 8, this is done by instantiating a `DataHandler` object with a suitable `DataSource` object. (The `DataHandler` object uses the `DataSource` object to provide a stream to a `DataContentHandler` object. The `DataContentHandler` object translates message content from the stream to an object in the Java programming language.)

For a `MimeMessage` object, the `DataSource` object is a `MimePartDataSource`. The `MimePartDataSource` provides an input stream that decodes any MIME content-transfer encoding that is present. The `MimePartDataSource` class:

1. Creates a `DataSource` from a `MimePart`

   The constructor `MimePartDataSource(MimePart part)` stores the part object internally and delegates to its methods. For example, the `MimePartDataSource` object's `getContentType` method just calls the part's `getContentType` method.

2. Implements the `DataSource` interface's `getInputStream` method

   The `MimePartDataSource` uses the `part` available to it to decode any MIME content-transfer encoding that is present. To do this the `getInputStream` method:

   a. Fetches the data stream using the `part`'s `getContentStream` protected method.

   b. Checks whether the part has any encoding (using the `getEncoding` method)

   c. If it finds any encoding, it attempts to decode the bytes

   d. Returns the decoded bytes.

When you subclass the `MimeMessage` class, you only need to override the `getContentStream` method to work with your protocol. When the `MimePartDataSource` class's `getInputStream` method is run, your subclass's `getContentStream` method provides the protocol-specific data stream, and the `MimeMessage` implementations of the remaining calls decode the content.

The POP3Message example follows. All but the `getContentStream` method is unchanged from the example in "Headers" on page 10.

```
public class POP3Message extends MimeMessage {
  boolean loaded = false;
  int hdrSize;
  ...
  protected synchronized InputStream getContentStream() {
    load();
    return new ByteArrayInputStream(content, hdrSize,
                                    content.length - hdrSize);
```

```
    }
    ...
    private synchronized void load() {
      if (!loaded) {
        // Get message data (headers and content) and cache it
        content = POP3Command("RETR", msgno);
        // Open a stream to the cache
        InputStream is = new ByteArrayInputStream(content);
        // Setup "headers" field by getting the header data
        headers = new InternetHeaders(is);
        // Save header size to easily access msg content from cache
        hdrSize = content.length - is.available();
        loaded = true;
      }
    }
  }
```

When clients call the `POP3Message` class's `getDataHandler`, `getContent`, or `getInputStream` methods of a `MimeMessage`, the bytes they receive are already decoded.

Multipart MIME Messages

As discussed in "The Structure of a Message" on page 3, messages can have a single content object or, indirectly, multiple content objects. The `DataHandler` of a MIME multipart message contains an object of class `MimeMultipart` from the `javax.mail.internet` package. Invoking the `getContent` method on a MIME multipart message typically returns this class.

You typically do not have to subclass the `MimeMultipart` class. The multipart `DataContentHandler` provided by the `com.sun.mail.handlers` package creates this object internally when it is given a DataSource. It is directed to do so by this entry in mailcap file in the JavaMail distribution:

```
multipart/*;; x-java-content-handler=\
com.sun.mail.handlers.multipart_mixed
```

Note that classes in the `com.sun.mail` package, and its subpackages, are not part of the JavaMail API. They are separate implementations of the API.

Special Cases: Protocols that Provide Preparsed Data

Some protocols, such as IMAP, provide preparsed data. In this case, you override most `MimeMessage` methods to avoid parsing the input stream. A `MimeMessage` subclass for a protocol like IMAP acts like an interface that models the MIME API.

For example, if multipart IMAP content is retrieved in the same way as multipart MIME content, the `DataContentHandler` re-parses the multipart data that has already been parsed at the server. To avoid the extra parse, you create a special type of

`DataSource` - the `MultipartDataSource`, and pass that to the `DataHandler`. The `DataHandler` passes it on to the `MultipartDataContentHandler`, which avoids the parse if its `DataSource` is already of type `MultipartDataSource`.

So, an IMAPMessage's getDataHandler() method may be:

```
public javax.activation.DataHandler getDataHandler()
  throws MessagingException {
  if (dh != null)
    return dh;
  if (myType.equals("multipart")) {
    dh = new DataHandler(new IMAPMultipartDataSource(this));
  } else {
    dh = new DataHandler(new IMAPDataSource(this));
  }
  return dh;
}
```

The `IMAPMultipartDataSource` is a subclass of `MultipartDataSource`.

Chapter 3:

Message Storage and Retrieval

Users interact with message stores to fetch and manipulate electronic mail messages. This chapter discusses how to implement the classes that allow clients this access. If you are creating a JavaMail service provider that allows a client to send mail, but does not interface with a mail store, you do not have to implement this functionality.

To provide message storage and retrieval, you must implement some abstract classes:

- `Store`, which models the message database and the access protocol used to retrieve messages. Its implementation is discussed in "Store" on page 15.
- `Folder`, which represents a node in the message storage hierarchy used to organize messages. Its implementation is discussed in "Folders" on page 17.

Store

The `Store` class models a message database and its access protocol. A client uses it to connect to a particular message store, and to retrieve folders (groups of messages).

To provide access to a message store, you must extend the `Store` class and implement its abstract methods. In addition, you must override the default implementation of at least one method that handles client authentication. The next sections cover how to write these methods. They begin with authentication, since it precedes retrieval when the provider is used.

Authentication

JavaMail provides a framework to support both the most common style of authentication, (username, passphrase), and other more sophisticated styles such as a challenge-response dialogue with the user. To furnish the (username, passphrase) style authentication in your provider, override the `protocolConnect` method. To use another style of authentication, you must override the version of the `connect` method that takes no arguments.

The `protocolConnect` *Method*

The `Store` class provides a set of methods that establish a connection with a message store. Establishing a connection typically involves setting up a network connection to a host and authenticating the user with the message store installed on that host. The `protocolConnect` method handles these tasks.

The signature of the `protocolConnect` method is:

```
boolean protocolConnect(String host, int port,
                        String user, String password)
```

The method returns `true` if the connection and authentication succeed. If the connection fails, it throws a `MessagingException`. If the authentication fails, it returns `false`.

The default implementation of `protocolConnect` returns `false`, indicating that the authentication failed. You should provide an implementation that connects to the given host at the specified port, and performs the service-specific authentication using the given username and passphrase. The simplest implementation, for message stores that do not require authentication, merely has this method return `true`. An example of such a message store is one that is local file-based.

Note that clients do not call the `protocolConnect` method directly. Instead, the `protocolConnect` method is invoked when clients call one of the `connect` methods

The `connect` *Method*

To provide authentication schemes more sophisticated than (username, passphrase), you must override the version of the `connect` method that takes no arguments.

The `connect` method takes no arguments and uses an `Authenticator` object to obtain information from the user if the information is not already available. (The client provides the `Authenticator` object.)

It then uses that information to connect to the message store and authenticate the user. Finally, if the connection is successful, it delivers an `OPENED ConnectionEvent`. (For more information about events, see Chapter 5: Events.)

Folder Retrieval

A message store stores messages, and often allows users to further group their messages. These groups of messages are called folders, and are represented by the abstract class, `Folder`. The `Store` class provides abstract methods for allowing the user to retrieve a folder:

- `getDefaultFolder`
- `getFolder`

If you are unfamiliar with folders, please read "Folders" on page 17 before continuing with this section.

The `getDefaultFolder` method must return the default folder. The returned folder must be in a closed state. (This is the default initial state of a folder.)

The `getFolder` methods return the specified folders:

```
Folder getFolder(String name)
Folder getFolder(URLName urlname)
```

These methods return the requested folders whether or not they exist in the store. (This is similar to the `java.io.File` API.) Do not validate the folder's existence in the `getFolder` methods.

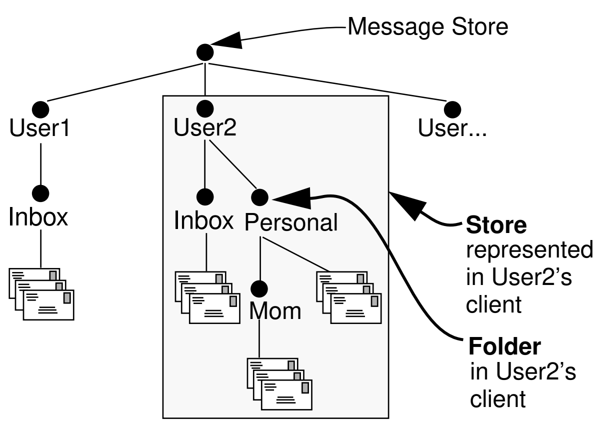
The folders returned by the `getFolder` methods must be in a closed state. (The default initial state of a folder is closed.)

Note – The `Store` object should not cache `Folder` objects. Invoking the `getFolder` method multiple times on the same folder name must return that many distinct `Folder` objects.

Folders

The `Folder` class models a node in a mail storage hierarchy. Folders can contain messages or subfolders or both. The following figure illustrates this:

FIGURE 3-1 Message Store Containing Folders



Each user has a folder that has the case-insensitive name `INBOX`. Providers must support this name. Folders have two states: they can be closed (operations on a closed folder are limited) or open.

Since `Folder` is an abstract class, you must extend it and implement its abstract methods. In addition, some of its methods have default implementations that, depending on your system, you may want to override for performance purposes. This section covers many of the abstract methods that you must implement, and the methods whose default implementations you might want to override. It groups them in the following way:

- "Folder Naming": `getName`, `getFullName`, `getSeparator`
- "Folder State": `open`, `close`
- "Messages Within a Folder": `getMessage`, `getMessages`, `search`, `fetch`
- "Folder Management": `getPermanentFlags`, `setFlags`, `appendMessages`, `copyMessages`, `expunge`

Folder Naming

Each folder has a name. One such name is `INBOX`; You must support that name in a case-insensitive fashion. Typically, mail systems allow users to create and name other folders for organizing their messages, leading to a tree-like organization of electronic mail in the message store.

The `Folder` class has two abstract methods that return the name of a folder:

- `getName`
- `getFullName`

A folder's full name is the combination of its name and its ancestors' names. Each level in the hierarchy of the folder's full name is separated from the next by the hierarchy delimiter character. The hierarchy delimiter character is returned by the method `getSeparator`.

The `getSeparator` method is an abstract method; implement it to return a `Folder`'s hierarchy delimiter. If your message store does not support a hierarchical storage structure, the `getSeparator` method must return the `NUL` character (`\u0000`).

Folder State

Folders can be in one of two states: open or closed. Initially a folder is closed. The operations allowed on a closed folder are limited; in particular, very few message related operations are allowed. Having the initial state of folders be closed allows them to be created as light-weight objects, with no dedicated server connection required. For example, an IMAP provider can designate a single server connection as the "common" connection for all closed folders.

Folders are opened using the method:

```
public abstract void open(int mode)
```

where mode is either READ_ONLY or READ_WRITE. These modes have the intuitive meanings: only a folder opened in READ_WRITE mode can be modified. If this folder is opened successfully, you must deliver an OPENED connection event.

The effect of opening multiple connections to the same folder on a specific Store is implementation dependent. You can choose to allow multiple readers but only one writer, or you could allow multiple writers as well as readers.

Once a folder is open, a variety of message specific methods, such as getMessage, can be invoked on it. Implement the open method such that these operations can be successfully conducted after the method returns. For example, an IMAP provider might want to open a new connection to the server and issue the SELECT command to select this folder.

Open folders can be closed with the method:

```
public abstract void close(boolean expunge)
```

The close method on an open folder typically has the Folder object get rid of its message-cache, if it maintains one, and generally reset itself, so that its a light-weight object again. Invoking the close method on a closed folder should not have any effect.

The close method must indicate that the folder is closed even if it terminates abnormally by throwing a MessagingException. That is, you must still deliver a CLOSED connection event and make the Folder object such that calls to the isOpen method return false.

Messages Within a Folder

Folders can be viewed as presenting a resizable array of messages to a client. They allow the client to access a message based on its index within this array. The index is the message's sequence-number. Sequence numbers begin at one (1) and continue, incrementing by one, through the total number of messages in the folder. A Folder implementation typically employs a suitable collection class, such as Vector, to store messages.

This section discusses the abstract methods that the Folder class provides to clients for retrieving messages and the information they contain. It groups these methods as follows:

- "Getting Messages"
- "Searching Messages"
- "Getting Message Data in Bulk"

Getting Messages

The Folder class provides two methods to get one or more messages from a folder: `getMessage` and `getMessages`.

The `getMessage` *Method*

The signature of the getMessage method is:

```
public abstract Message getMessage(int index)
```

Note that the `getMessage` method is abstract; you must provide an implementation for it. It returns the `Message` object with the given sequence number. Message numbers begin at one (1).

It is important that your implementation of the `getMessage` method does not return a completely filled (also called a *heavy-weight*) `Message` object. The client's expectation is that a `Message` object is just a "reference" to the message, so instead you should create Message objects that are almost empty (also called *light-weight* messages). The client will fill it as content is needed by the user. For example, an IMAP implementation might create IMAPMessage objects that initially contain only the appropriate IMAP Sequence number or IMAP UID.

The `getMessages` *Methods*

The `getMessages` methods have the following signatures:

```
public Message[] getMessages()
public Message[] getMessages(int[] msgnums)
public Message[] getMessages(int start, int end)
```

The `getMessages` methods have default implementations that use `getMessage` to return the requested messages. The `getMessages` method, when given no parameters, returns all of the `Message` objects in the folder.

Note – `Folder` implementations should cache `Message` objects. This insures that if a client calls the `getMessage` method multiple times, the implementation will efficiently return the same `Message` object unless the client calls the `expunge` method.

Clients that use message-numbers as their references to messages will invoke the `getMessage` method quite often to get at the appropriate `Message` object. Creating a new `Message` object each time, instead of caching the messages, would be expensive in terms of memory and time.

Searching Messages

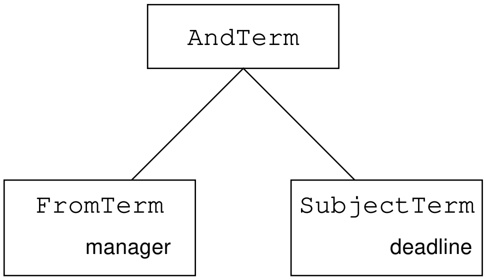
The `Folder` class provides search methods that return the messages that match the given search criteria. The signatures of the methods are shown below. The first `search` method shown, which takes only a `SearchTerm` argument, applies the search criteria to each message in the folder. The second `search` method shown applies the search criteria to the specified messages.

```
Message[] search(SearchTerm term)
Message[] search(SearchTerm term, Message[] msgs)
```

Default implementations are provided for both `search` methods. The default implementations do client-side searching by applying the given `SearchTerm` on each `Message` and returning those messages that match.

If your message store supports server-side searching, you can override the default implementation to take advantage of it. Your implementation should

1. **Construct a search expression corresponding to the SearchTerm provided.**
   The client uses `SearchTerm` objects to construct a tree of terms that represent a search criteria. For example, the tree shown below represents a search for messages from "manager" that contain the word "deadline" in the subject:

   Traverse the search-tree specified by the given `SearchTerm` to construct the search expression for the server-side search.

2. **Use the constructed search expression on the server.**
   For example, an IMAP provider will convert the SearchTerm into an IMAP search sequence and pass it on to the IMAP server.

   If the `SearchTerm` is too complex, or contains a subclass of the `SearchTerm` class that the client has defined, you can either throw a `SearchException` or use the default implementation of client-side searching by calling `super.search`

3. **Return light-weight `Message` objects.**
   To repeat the example given on page 20, an IMAP implementation might return `IMAPMessage` objects that initially contain only the appropriate IMAP Sequence number or IMAP UID.

Getting Message Data in Bulk

As mentioned in the previous sections, a `Message` object should start out as a light-weight reference to the corresponding message. The client invokes methods to fill in the message as the data is required.

To help deliver message content, certain server-based message access protocols, such as IMAP, allow batch fetching of message attributes for a range of messages in a single request. The `Folder` class provides a `fetch` method to allow service providers to take advantage of this capability; check your service provider's documentation.

The `fetch` method takes an array of `Message` objects and a `FetchProfile` object as arguments. Its signature is:

```
public void fetch(Message[] msgs, FetchProfile fp)
```

The `FetchProfile` object lists the message attributes to be obtained for the messages. The `fetch` method, if a service provider supports getting message data in bulk, gets the requested attributes and stores them in the `Message` objects.

If your access protocol allows batch fetching of message attributes, then you should override this method to allow clients to take advantage of it. The default implementation of the `fetch` method returns without doing any work.

When clients call the `fetch` method, they provide a `FetchProfile` with the names of the items to be obtained. The items can be pre-defined `FetchProfile` attributes, the names of header-fields, or both. The currently defined attributes are:

- `ENVELOPE`—This includes the common "toplevel" attributes of a message. These are generally the main addressing attributes - From, To, Cc, Bcc, Reply-To, Subject and SentDate. GUI Mailers usually display a subset of these items in header-list window, so a provider must attempt to include at least these items. For example, an IMAP provider will include the `ENVELOPE` data item.
- `CONTENT_INFO`—This specifies information about the content of the message, including the ContentType, ContentDisposition, Size and LineCount. For example, an IMAP provider will include the `BODYSTRUCTURE` data item.
- `FLAGS`—This specifies the flags for a message. (More information on flags is provided in "Handling Message Flags")

Your implementation of the `fetch` method should support the `FetchProfile` attributes appropriate for your system.

Folder Management

This section discusses the abstract methods that the `Folder` class provides to clients for manipulating a group of messages in a folder. It groups these methods as follows:

- "Appending and Copying Messages"
- "Expunging Messages"
- "Handling Message Flags"

Appending and Copying Messages

The `Folder` object provides an abstract `appendMessages` method. Your implementation should add the given messages onto the end of this folder's messages and deliver a `MessageCountEvent` if possible. Note that the append operation is valid on a closed folder.

Some or all of the `Message` objects might belong to the same `Store` as this `Folder`. If your system can optimize the append operation by doing server-side copies, you might want to check for and handle this special case.

The `copyMessages` method copies the specified messages from this folder to the destination folder. There is a default implementation for this method that uses the `appendMessages` method to do the copy. If your system supports server-side copy, make sure that this operation employs that optimization, either by overriding this method or by implementing the `appendMessages` method to handle this case.

Expunging Messages

The `Folder` object provides an abstract `expunge` method. Your implementation should:

- Remove all messages that are marked deleted (i.e, have their `DELETED` flag set) from the folder and set the values of those messages' `expunged` fields to true.
- Renumber the messages in the folder that occur after an expunged message so that their sequence-numbers match their index within the folder. For example, if messages A and C are removed due to the `expunge` method being invoked, the remaining messages (B, D and E) are renumbered suitably:

A (1)	B (2)	C (3)	D (4)	E (5)	Folder *before* invoking the expunge method

B (1)	D (2)	E (3)	Folder *after* invoking the expunge method

- Send one or more `MessageCountEvents` to notify listeners about the removal of the messages. When you call the `notifyMessageRemovedListeners` method, its `boolean` argument, `removed`, must be set to true.

Only the `getMessageNumber` and `isExpunged` methods are valid on an expunged `Message` object.

Some messaging systems support shared folders that can be accessed and modified from multiple sessions at the same time. In such cases, multiple open `Folder` objects can correspond to the same physical folder. An `expunge` operation on one of those `Folder` objects removes all deleted messages from the physical folder. However, the other folders must *not* remove the corresponding `Message` objects from *their* lists. They should mark those messages as `expunged`, so that any direct method on those `Messages` will fail. They may also fire `MessageCountEvents` (with the `removed` boolean flag set to false) to notify listeners about the removal. In essence, those `Folders` will continue to present the same, unchanged array of `Messages` to their clients. The array is purged and messages are renumbered only when the `expunge` method is directly invoked.

Refer to Section 6.2.3 in the JavaMail 1.1 Specification document for the rationale for this behavior.

Handling Message Flags

Flags are indicators of message state stored with a message. The set of flags associated with a message is represented by a `Flags` object; individual flags are represented by the `Flags.Flag` object. The flags represented by the `Flags.Flag` class include `ANSWERED`, `DELETED`, `DRAFT`, `FLAGGED`, `RECENT`, `SEEN`, and `USER`. (The `USER` flag means that this folder supports user-defined flags.)

The `Folder` class provides the `getPermanentFlags` method and the `setFlags` methods for handling flags. This section first covers the `getPermanentFlags` method, then the `setFlags` methods.

The `getPermanentFlags` method is abstract:

```
public abstract Flags getPermanentFlags()
```

Your implementation should return a `Flags` object that contains every `Flags.Flag` that your system supports.

The `setFlags` methods have default implementations to set the specified flags on a given range of messages. They set the flags on each `Message` object individually (after obtaining the message by calling the `getMessage` method, if necessary) and send the appropriate `MessageChangedEvent`.

Most message stores provide a call in their API to efficiently set flags on a group of messages. If your system does this, consider overriding the default implementations of the `setFlags` methods to make use of the server-side optimization.

If you override the `setFlags` methods, be sure that the methods that operate on sequence-numbers do *not* abort the operation if any sequence-number refers to an expunged message. Instead of aborting, your implementation should continue operating on the rest of the messages.

Chapter 4:

Message Transport

The JavaMail API provides the ability for users to send electronic mail messages. This chapter describes how to furnish a JavaMail service provider of a message transport system. If you are creating a JavaMail service provider that allows a client to access a mail server but does not handle sending mail, you do not have to implement this functionality.

To provide a message transport system, you must,

- Provide a `Transport` implementation (See "Transport" on page 25.)
- Provide an `Address` subclass (See "Address" on page 27.)

Transport

The function of the `Transport` class is to send (transport) messages; it is an abstract class. To implement a specific transport protocol:

- Subclass the `Transport` class;
- Implement the `Transport` class's abstract method, `sendMessage;`
- Override the default implementation of the `Transport` class's `protocolConnect` method.

The `sendMessage` Method

The `Transport` class provides static methods that applications use to send messages. The default implementations of these methods call the abstract method `sendMessage` to do the actual transmission. The `sendMessage` method has the following signature:

```
public abstract
      void sendMessage(Message m, Address[] address)
            throws MessagingException
```

To implement this method, follow these steps:

1. **Check the type of the given message.**
   Typically, a service provider handles only certain types of messages. For example, an SMTP provider typically sends `MimeMessages`. In the face of an unknown message type, you can have `sendMessage` either fail and throw a `MessagingException`, or you can try to coerce it into a known type and send it.

2. **Transmit the message**
   Get the byte stream of the message, and transmit the message using its `writeTo` method.

3. **Send a `TransportEvent`**
   The `TransportEvent` indicates the delivery status of the message. The possible event types are `MESSAGE_DELIVERED`, `MESSAGE_NOT_DELIVERED` and `MESSAGE_PARTIALLY_DELIVERED`. For more information about events, see Chapter 5: Events.

4. **Throw an exception if the delivery is unsuccessful**
   If the delivery fails, completely or partially, you must throw a suitable `MessagingException` or `SendFailedException`.

The `protocolConnect` *Method*

The `Transport` class provides methods for applications to call that establish a connection with a transport. The methods, called `connect`, have default implementations that establish a connection by calling the `protocolConnect` method. You must override the `protocolConnect` method.

The signature of the `protocolConnect` method looks like this:

```
protected
   boolean protocolConnect(String host, int port,
                            String user, String password)
```

The method returns `true` if the connection and authentication succeed. If the connection fails, it throws a `MessagingException`. If the authentication fails, it returns `false`.

The default implementation of `protocolConnect` returns `false`, indicating that the authentication failed. You should provide an implementation that connects to the given `host` at the specified `port`, and performs the transport-specific authentication using the given `username` and `password`.

Address

The `Address` class is an abstract class. Subclasses provide specific implementations. Every `Address` subclass has a type-name, which identifies the address-type represented by that subclass. For example, the `javax.mail.internet.InternetAddress` subclass has the type-name: `rfc822`.

The type-name is used to map address-types to `Transport` protocols. These mappings are set in the `address.map` registry. For example, the default address.map in the JavaMail package contains the following entry:

```
rfc822=smtp
```

Setting up the address-type to transport-protocol mapping is covered in Chapter 6: Packaging, Step 5.

The Address-type to Transport mapping is used by JavaMail to determine the `Transport` object to be used to send a message. The `getTransport(Address)` method on `Session` does this, by searching the `address.map` for the transport-protocol that corresponds to the type of the given address object. For example, invoking the `getTransport(Address)` method with an `InternetAddress` object, will return a Transport object that implements the `smtp` protocol.

An `Address` subclass may also provide additional methods that are specific to that address-type. For example, one method that the `InternetAddress` class adds is the `getAddress` method.

Chapter 5: Events

The `Store`, `Folder` and `Transport` classes use events to communicate state changes to applications. The documentation for the methods of these classes specify which events to generate. A compliant provider must broadcast these events.

To broadcast an event, call the appropriate `notify`*`Event`*`Listeners` method. For example, to manage `MessageCountEvents` for new mail notification, your `Folder` subclass should call the `notifyMessageAddedListeners(msgs)` method. (It is best to use the default implementations of the `Notify`*`Event`*`Listeners` methods, because they dispatch their events in an internal event-dispatcher thread. Using a separate thread like this avoids deadlocks from breakage in the locking hierarchy.)

Every event generated by the `Store`, `Folder` and `Transport` classes also has associated `addListener` and `removeListener` methods. Like the `notify`*`Event`*`Listeners` methods, these methods already have useful implementations. A programmer using your service provider implementation calls the appropriate `add`*`Event`*`Listener` and `remove`*`Event`*`Listener` methods to control which event notifications are received.

Chapter 6: Packaging

Provider software must be packaged for use by JavaMail clients. To do this:

1. **Choose a suitable name for your package**
   The recommended way of doing this is to reverse your company domain name, and then add a suitable suffix. For example, Sun's IMAP provider is named `com.sun.mail.imap`.

2. **Make sure that your key classes are public**
   If you provide access to a message store, your `Store` subclass must be a `public` class. If you provide a way to send messages, your `Transport` subclass must be a `public` class. (This allows JavaMail to instantiate your classes.)

3. **Bundle your provider classes into a suitably named `jar` file**
   The name of the `jar` file should reflect the protocol you are providing. For example, an NNTP provider may have a `jar` file named `nntp.jar`. Refer to a suitable Java programming language book for details on how to create jar files.

   Because your `jar` file must be included in an application's `classpath` so that it can be found by the application's classloader, include the name of your `jar` file in the documentation for your provider. Mention that the application's classpath should be updated to include the location of the `jar` file.

4. **Create a registry entry for the protocol your implementation provides**
   A registry entry is a set of attributes that describe your implementation. There are five attributes that describe a protocol implementation. Each attribute is a name-value pair whose syntax is *`name=value`*. The attributes are separated by semicolons (`;`). The attributes are:

TABLE 6-1 Attributes in a JavaMail Resource File

Attribute Name	Description of the Attribute Value
protocol	Name assigned to the protocol, such as `imap`
type	Type of protocol: either the value `store` or the value `transport`
class	Full name of the class, including its package, that implements this protocol
vendor	Optional entry: a string identifying yourself or your organization as the vendor
version	Optional entry: a string identifying the version number of this implementation

For example, an entry for a `POP3` provider from `FOOBAR.com` looks like this:

```
protocol=pop3; type=store; class=com.foobar.pop3.POP3Store;
vendor=FOOBAR
```

The users or administrators of a JavaMail application place your registry entry into a registry, either manually or using a configuration tool. This installs your provider into the client's JavaMail system.

A registry is comprised of resource files. The name of the file that holds your entry is called `javamail.providers`. JavaMail searches resource files in the following order:

1. *java.home*`/lib/javamail.providers`
2. `META-INF/javamail.providers`

In the documentation that you provide to users, provide your registry entry and request that it be placed in one of the two files listed above.

5. **Create any mapping from an address type to your protocol**
If you are providing an implementation that allows applications to send mail, you must create a mapping between the types of addresses that your implementation can deliver and your protocol. The mapping has the format *`addressType=protocol`*, where

- `addressType` is the string returned by your `Address` subclass's `getType` method
- `protocol` is the value of the protocol attribute that you provided in the Step 4.

The users or administrators of a JavaMail application place your mapping into the `address.map` registry, either manually or using a configuration tool. As stated previously, a registry is comprised of resource files. The name of the file that holds your mapping is called `javamail.address.map`. JavaMail searches resource files in the following order:

1. *java.home*`/lib/javamail.address.map`
2. `META-INF/javamail.address.map`

In the documentation that you provide to users, provide your mapping, and request that it be placed in one of the two files listed above.