Messaging Access SDK Guide

Messaging Access SDK
Java Version

Version 3.5

Recycled and Recyclable Paper

The Team: Messaging Access SDK Project Team, Manager: Gena Cunanan
Engineering: Soon Shin, Derek Tumulak, Prasad Yendluri
Marketing: Faten Hellal
Publications: Sharon Williams
Version 3.5

Part Number

Printed in the United States of America. 00 99 98 5 4 3 2 1

Netscape Communications Corporation, 501 East Middlefield Road, Mountain View, CA 94043

This guide describes the Java version of the Netscape Messaging Access SDK 3.5, a development kit for writing messaging applications.

June 8, 1998

Contents

About This Guide

The *Messaging Access SDK Guide* is the developer's guide and reference to the Netscape Messaging Access software development kit (SDK), version 3.5, for writing messaging applications.

The *Messaging Access SDK Guide* tells you how to tap the capabilities of familiar and powerful Internet Access Protocols, POP3, IMAP4, SMTP, and MIME, in your messaging applications.

This guide describes the Java version of the Messaging Access SDK.

Note For system requirements and installation information, see the `ReadMe` file that is available on the Netscape web site. §

This chapter includes the following sections:

- Who Should Read This Guide
- What's in This Guide
- Conventions Used in This Guide
- Where to Find More Information

[Top]

Who Should Read This Guide

The *Messaging Access SDK Guide* is designed for developers who want to create messaging applications based on the standard Internet protocols SMTP, POP3, MIME, and IMAP4.

[Top]

What's in This Guide

This is the guide to read if you want to write messaging applications using a variety of messaging access APIs. This guide comes in two versions, which document the C and Java versions of the Messaging Access SDK. You are reading the Java version of the Guide.

- Organization
- Quick Reference to Tasks

[Top]

Organization

To provide quick access to conceptual information, task-based development information, and reference information, the guide consists of these parts:

- Using the Messaging Access SDK--Messaging Access SDK basics and development information.
- Messaging Access SDK Java Reference--Reference to the Java interfaces of the POP3, IMAP4, SMTP, and MIME Protocol APIs, and the Java Convenience API of the Messaging Access SDK. This reference is in JavaDocs format.

[Top] [What's in This Guide]

Quick Reference to Tasks

To help you find the information you need more quickly, look in the column on the left for the task you want to perform. Click the title in the column on the right to go directly to the appropriate chapter.

If you want to do this:	See this chapter:
Learn more about Netscape Access APIs.	Chapter 1, "Introducing the Messaging Access SDK."
Understand how the Access APIs work together in a Messaging Access SDK application.	Chapter 1, "Introducing the Messaging Access SDK."
Send email messages.	Chapter 2, "Sending Mail with SMTP."
Encode/decode and parse messages. Add attachments to messages.	Chapter 3, "Building and Parsing MIME Messages."
Retrieve manage messages on the server.	Chapter 4, "Receiving Mail with IMAP4."
Retrieve messages, messages attributes, and parts of messages.	Chapter 5, "Receiving Mail with POP3."
Find out which Internet Protocol commands are called by Messaging Access SDK methods.	Chapter 6, "Reference to Protocols."
Find out what you need to know about using multithreading in your messaging applications.	Appendix A, "Writing Multithreaded Applications with the Messaging Access SDK."

To look up the Java classes and methods you need for your messaging application, see Part 2, "Messaging Access SDK Java Reference."

[Top] [What's in This Guide]

Conventions Used in This Guide

Fonts. All program code listings, URLs, and other program names appear in Courier, a monospace font. Placeholders, which you replace with your own value, are in italicized Courier font.

Note Formats. This guide emphasizes information with several types of note formats:

Note Information of interest to the developer but not essential to understanding the surrounding topic. §

Warning Information that can affect the development decisions you make or the development environment you choose. Don't miss these notes. §

Terminology. This guide uses the word *command* to represent Internet Protocol commands, and the word *method* to represent the Messaging Access SDK Java implementation that calls this command. For example, the Messaging Access SDK `IMAP4Client.close` method sends the CLOSE IMAP4 protocol command.

[Top]

Where to Find More Information

For information for developers, see the Netscape DevEdge site.

For information about the Java programming language, see the Sun Java web site.

This guide tells you how to use each Protocol API for Messaging Access SDK tasks. Internet Protocols are introduced and described in RFC (Request for Comments) documents from the Network Working Group. For further information about the Protocols, see the following RFCs.

SMTP RFCs

- RFC 821: “Simple Mail Transfer Protocol,” August 1982
- RFC 1854:”SMTP Service Extension for Command Pipelining,” October 1995
- RFC 1869: “SMTP Service Extensions,” November 1995
- RFC 1891: “SMTP Service Extension for Delivery Status Notifications,” January 1996
- RFC 2197: “SMTP Service Extension for Command Pipelining,” September 1997

MIME RFCs

- RFC 2045: “Multipurpose Internet Mail Extensions (MIME), Part One: Format of Internet Message Bodies,” November 1996
- RFC 2046: “Multipurpose Internet Mail Extensions (MIME), Part Two: Media Types,” November 1996
- RFC 2047: “MIME (Multipurpose Internet Mail Extensions), Part Three: Message Header Extensions for Non-ASCII Text,” November 1996
- For MIME headers: RFC 822: “Standard for the Format of ARPA Internet Text Messages,” August 1982

IMAP4 RFCs

- RFC 2060: “Internet Message Access Protocol - Version 4rev1,” December 1996

- RFC 2086: “IMAP4 ACL extension,” January 1997
- Internet Draft**:** “IMAP4 Namespace,” December 1997

POP3 RFCs

- RFC 1939: “Post Office Protocol - Version 3,” May 1996

Note An index to RFCs is available through the Internet FAQ Consortium. §

[Top]

1 *Using the Messaging Access SDK*

Chapter 5 Receiving Mail with POP3

This chapter is an overview of using POP3 (Post Office Protocol 3) to download messages to a client.

[Top]

Chapter

1

Introducing the Messaging Access SDK

The Netscape Messaging Access software development kit (SDK) provides a set of Protocol Level APIs that the developer can use to write messaging applications and extend applications with messaging services. This chapter is an overview of the Messaging Access SDK, version 3.5.

The Messaging Access SDK provides SMTP, MIME, POP3, and IMAP4 APIs in the Java and C programming languages, for a variety of platforms.

The *Messaging Access SDK Guide* provides developers with a complete set of software libraries, sample code, and documentation for building mail-enabled applications.

[Top]

How the Protocol APIs Work Together

The Messaging Access SDK provides implementations of the Internet messaging protocols, SMTP, IMAP4, MIME, and POP3. These Protocol APIs are designed to work together, yet have the ability to operate independently of each other.

SMTP (Simple Mail Transfer Protocol). SMTP sends non-encoded or MIME-encoded messages. You can use MIME to prepare to send messages in formats other than text, to encode messages, and to include attachments. For more information, see Chapter 2, “Sending Mail with SMTP.”

MIME (Multipurpose Internet Mail Extension). MIME builds and encodes messages with attachments for sending with SMTP, and parses and decodes received messages. The encoded MIME message is passed to SMTP.

The MIME API consists of the MIME encoder and the MIME parser. The MIME encoder is used to build MIME messages with attachments and encode them for sending over SMTP. You can use the MIME API to parse and decode messages when they are received through IMAP4 or POP3. For more information, see Chapter 3, “Building and Parsing MIME Messages.”

IMAP4 (Internet Message Access Protocol, version 4). IMAP4 is used to retrieve and manage messages remotely. The user can save messages on the server or locally. In addition, the user can manipulate items on the server (for example, create or delete mailboxes). IMAP4 supports multiuser mailboxes. For more information, see Chapter 4, “Receiving Mail with IMAP4.”

POP3 (Post Office Protocol, version 3). POP3 connects to the server and retrieves messages. POP3 is simpler than IMAP4 and provides a subset of its capabilities. It supports one user per mailbox. For more information, see Chapter 5, “Receiving Mail with POP3.”

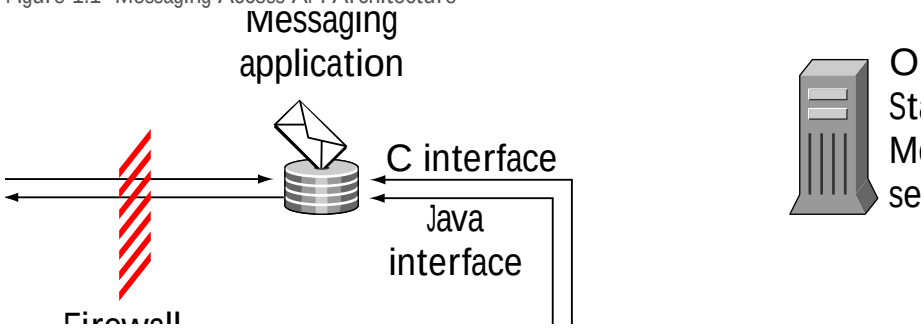
For a quick reference to the Internet Protocol commands called by Messaging Access SDK methods, see Chapter 6, “Reference to Protocols.”

The Protocol APIs are designed to co-exist in the same client environment and match each other's interfaces where applicable. For example, the message data chunks returned by POP3 and IMAP4 APIs can be fed to the MIME SDK API to parse the message contents. In the same way, the encoded message byte-stream returned by the MIME API can be passed to the SMTP API to transmit the message. At the same time, the APIs are designed to allow customers to use only the API required by their application.

Applications written with the Messaging Access SDK Protocol APIs can work with any messaging system that implements the Internet messaging protocols, primarily the Netscape Messaging Servers. Protocol APIs are self-contained and are intended to coexist with all other SuiteSpot SDKs; they are independent of Netscape Server releases.

Each Messaging Access SDK Protocol API is designed to follow its Internet standard specification. API invocations result in the exchange of standard protocol elements with the server. Any information exchange with the server conforms to one of the standard protocols.

Figure 1.1 Messaging Access API Architecture



The Messaging Access SDK is designed to parse and format protocol elements and make these available to the programmer through Java classes or C data structures and the methods and functions that access them.

The Messaging Access SDK Protocol APIs are built to be thin and are optimized for performance and memory. Each Protocol API includes a `sendCommand` (or pass-through) interface, which programmers can use to send protocol elements that are not directly supported by the API. The IMAP4 API has further conveniences, such as transparently handling unilateral and unsolicited responses from the server and making these available at the API level through a callback mechanism.

For more information about the `sendCommand` API, see the reference entries for `SMTPClient.sendCommand`, `POP3Client.sendCommand`, and `IMAP4Client.sendCommand`.

For more information about using callbacks, see the section about callback mapping in each of these chapters: Sending Mail with SMTP, Receiving Mail with IMAP4, Receiving Mail with POP3, and Building and Parsing MIME Messages.

[Top]

The Messaging Access SDK, Java Version

The Messaging Access SDK (Software Development Kit) comes in a zip file on Unix and a self-extracting executable on MS Windows.

You can download the SDK at this URL.

For the latest installation information, see the ReadMe file for the SDK.

The Messaging Access SDK download file contains the following directories and files:

- `packages` directory - JAR file that contains the Java classes that make up the Messaging Access SDK. You do not have to unzip this file.
- `examples` directory - Sample code that demonstrates selected parts of the Messaging Access SDK.
- `ReadMeJ.htm` - Links to current installation information, development notes, system requirements, information about using the SDK that may be more current than this guide, and Netscape licensing information.

On Unix, unzip the downloaded file using a utility that preserves the file hierarchy, for example, gzip. On MS Windows, simply execute the self-extracting executable. For links to the latest information about installation, see `ReadMeJ.htm`, included in the SDK.

[Top]

Supported Platforms

The Messaging Access SDK supports the MS Windows and Unix platforms listed in Table 1.1.

Table 1.1 Supported Platforms

Platforms	Supported Versions
Solaris	2.5.1, 2.6
Windows NT	4.0 with SP 3
Windows 95	
AIX	4.21
IRIX	6.2
OSF/DEC Unix	4.0d
HP-UX	11.0

[Top]

SDK Sink Classes for Java

The SMTP, IMAP4, and POP3 response sinks and the MIME data sink are Java interfaces that contain callback methods for each client call. These Java interfaces contain method definitions and opaque data.

- The method definitions are patterns for the implementation of callbacks; they do not have implementations and do nothing in themselves.
- The opaque data represents client data that is set by the application and always returned to the application when the callbacks are made.

For easy reference, each protocol chapter includes a table that shows how its methods are mapped to callbacks. See the individual protocol chapters under "Method Callback Mapping."

When you start a session with SMTP, IMAP4, or POP3, you first create (initialize) the response sink. Then you create the client, which calls the response sink methods. To use the client object, you must implement the response sink interface, as the constructor for the client class takes a sink object as a parameter. The response sink receives and processes all the available server response data whenever the `processResponses` call of the client class is issued. This call reads in responses from the server and invokes the appropriate callback method for all responses that are available at the time of execution.

When you start a session with the MIME dynamic parser, you first create and initialize the data sink, and then you create the parser. The parser makes callbacks to its data sink based on the kind of data it finds in the input stream. For example, if it finds a header, it makes the header callback. For the other protocols, the callback comes from the server and callbacks tend to be tied to individual methods.

For MIME, the kind of callback is dependent upon the kind of data that is in the input stream. There are no particular correspondences between functions and data sink callback prototypes, as there are in the other protocols.

As a convenience, SMTP, IMAP4, and POP3 provide sink classes that implement the response sink interface. For example, the `SMTPSink` class implements the `ISMTPSink` interface. You can save a step by extending the sink class, or you can implement your own class based on the interface.

For more information, see the section about implementing the response sink in each of these chapters: Sending Mail with SMTP, Receiving Mail with IMAP4, Receiving Mail with POP3, and Building and Parsing MIME Messages.

Note SMTP, IMAP4, and POP3 commands are asynchronous. After sending a command, the application does not have to wait to issue the next one, but can do something else. §

[Top]

SDK Exceptions for Java

In general, the Messaging Access SDK uses standard Java exception handling for dealing with unexpected occurrences during processing. In addition, it provides several exception classes.

Messaging Access SDK Exceptions

Protocol Exceptions handle internal errors in the protocol implementation of the Messaging Access SDK. These exceptions can be thrown when an error is detected within the SDK (on the client side) or when data received from the server results in a parsing error. Exception classes: `SMTPException`, `MIMEException`, `POP3Exception`, `IMAP4Exception`, `IMException`.

Protocol Server Exceptions handle server response errors. These exceptions are caused when the server sends an error saying that some part of the operation failed or is not supported. This can happen even when all relevant code executes properly and everything operates normally on the client side.

Server Exceptions are thrown from the `error` callback on the response sink when an API call that is mapped directly to the RFC fails. It is up to the developer to determine whether or not the `error` callback will throw this exception. As an SDK default, the SMTP, IMAP4, and POP3 sink classes that represent the response sink classes throw an exception whenever the `error` callback is called. Server Exception classes: `SMTPServerException`, `POP3ServerException`, `IMAP4ServerException`.

Standard Java Exceptions

`IOException` exceptions handle I/O errors, which usually occur when the user tries to make API calls before connecting or if the connection is lost unexpectedly. The user can respond by catching the `IOException` and examining the reason for the failure. The user can try to re-establish the connection by calling `connect` again.

`InterruptedIOException` exceptions occur when a time-out occurs. This is a recoverable condition; the application can wait and reissue the method.

[Top]

Compiling with the Java SDK

Follow these steps to compile the Java version of the Messaging Access SDK. These instructions assume you have already set up your Java development environment. If not, you can download the JDK 1.1.5 (Java Development Kit) from the Java web site.

The Java version of the Netscape Messaging SDK comes with two Java Archive (JAR) files.

- The `proapi.jar` file contains the Java classes for the protocol APIs.
- The `coapi.jar` file contains the Java classes for the convenience APIs.

When you install the Netscape Messaging Access SDK, these files are copied into the `packages` subdirectory under the `install-root` directory.

Setting the CLASSPATH

You must set the `CLASSPATH` environment variable to include the fully qualified path names for the `proapi.jar` and the `coapi.jar` files.

On Unix platforms, set the `CLASSPATH` as follows:

If you are using `ksh`*:*

```
%

CLASSPATH=$CLASSPATH:<fully-qualified-path>/proapi.jar:
                     <fully-qualified-path>/coapi.jar

% export CLASSPATH
```

For convenience, consider adding the lines above to your `.profile` file.

If you are using `csh`*:*

```
% set classpath=($classpath <fully-qualified-path>/proapi.jar
                            <fully-qualified-path>/coapi.jar)
```

For convenience, consider adding the lines above to your `.cshrc` file.

On MS Windows platforms, set the `CLASSPATH` as follows:

```
C:\> set

CLASSPATH=%CLASSPATH%;<fully-qualified-path>\proapi.jar;
                      <fully-qualified-path>\coapi.jar
```

If you are using MS Windows 95, consider adding these lines to your `AUTOEXEC.BAT` file.

If you are using MS Windows NT, make these environment variable changes in the Control Panel. To do this, start the Control Panel, select System, and then edit the environment variables.

Note If you are not using the Sun JDK, you may need to make other environment changes as specified by the vendor. For example, if you are using Symantec Visual Cafe 2.0, you must set the CLASSPATH in the `sc.ini` file in the `bin` directory. §

If your environment requires it, set your PATH variable to include the target directory for building your Java application.

After you have set up your environment, you can build an application that uses the Messaging Access SDK API by invoking the `javac` compiler on the Java files you want to compile. For example, if you are using the Sun JDK, you could use this code:

```
cd <build-directory>;  javac *.java
```

If you are using another environment, such as Symantec Visual Cafe, follow the build procedure for that environment.

[Top]

Chapter

2

Sending Mail with SMTP

This chapter is an overview of using SMTP (Simple Mail Transfer Protocol) to create and send email messages.

- The SMTP Protocol
- SMTP Callback Mapping
- Creating a Response Sink
- Creating a Client
- Connecting to a Server
- Determining ESMTP Support
- Setting the Mailer
- Setting the Recipient
- Sending the Message
- Sending Messages with Convenience APIs
- Ending the Session

[Top]

The SMTP Protocol

SMTP (Simple Mail Transport Protocol) allows clients to deliver mail messages to SMTP servers. To retrieve these messages, the client uses the IMAP4 or POP3 protocol. Servers can use SMTP to move messages from one server to another before delivering them to a mailbox.

The SMTP client always starts the session, but either client or server can end it. The client starts the session by connecting to the server. The server acknowledges the message with a greeting. The client responds, and, in subsequent commands, specifies the message sender and recipients and sends the message.

SMTP commands are made up of a keyword, followed by any parameters the method has. Commands receive a three-digit response code, described in SMTP Response Codes. SMTP commands include only the U.S. ASCII character set, a subset of ASCII that includes the values 00h-7Fh (0d-127d).

The responses returned by SMTP commands are made up of a three-digit numeric code followed by descriptive text. The client application can detect and handle the response or display the message to the user for interpretation. For more information, see SMTP Response Codes.

If your server supports Extended SMTP (ESMTP), which is provided in an update to the existing SMTP specification, your mail application can take advantage of ESMTP elements, such as pipelining, the `bdat` command, data chunking, and DSN. For more information, see Determining ESMTP Support.

During a single SMTP session, the client can send multiple unrelated, independently addressed messages. Because of this, the SMTP client can increase efficiency by batching messages and sending them together using pipelining. For more information, see Pipelining Commands.

The SMTP server waits for SMTP messages on the “well-known” TCP port 25. Many mail applications allow the user to specify a different port.

For a table of SDK-supported SMTP protocol commands, see Supported SMTP Internet Protocol Commands. For detailed information about SMTP, consult one of the RFCs listed, with links, in SMTP RFCs.

[Top]

Steps in an SMTP Session

Generally, a messaging application follows these steps when using SMTP to send mail. These steps are listed below with links to more detailed descriptions.

Step	Section with details
Initialize the response sink.	Creating a Response Sink
Initialize the client.	Creating a Client
Connect to the server.	Connecting to a Server
Determine Extended SMTP (ESMTP) features supported by the server.	Determining ESMTP Support
Set the mailer.	Setting the Mailer
Set the recipients.	Setting the Recipient
Send the message.	Sending the Message
End the SMTP session.	Ending the Session

[Top]

SMTP Response Codes

When the client sends an SMTP command, the response that comes back contains a standard three-digit response code followed by descriptive text. This section is an overview of SMTP responses. For detailed information, see RFC 821.

The response contains the three digit code, a space, and one or more lines of text that describes the response. If the response is multi-line, each subsequent line also contains the three digit code, a hyphen, and text. The final line contains the code, a space, and text.

This table lists some of the most common SMTP reply codes. In general, response codes in the 100 to 300 range are considered successful; those in the 400 to 500 range are considered unsuccessful.

Table 2.1 SMTP Reply Codes

Code	Text of Response
211	system status, or system help reply
214	help message
220	<domain> service ready
221	<domain> service closing transmission channel
250	request mail action okay, completed
251	user not local, will forward to <forward-path >
354	start mail input; and with <CRLF>.<CRLF>
421	<domain> servers not available, closing transmission channel
450	requested mail action not taken: mailbox unavailable
451	requested action aborted: local error in processing
452	requested action not taken: insufficient system storage
500	syntax error, command unrecognized
501	syntax error in parameters or arguments
502	command not implemented
503	bad sequence of commands
504	command parameter not implemented
550	requested action not taken: mailbox unavailable
551	user not local; please try <forward-path>
552	requested mail action aborted: exceeded storage allocation
553	requested action not taken: mailbox name not allowed
554	transaction failed

The first digit of the SMTP reply code basically tells whether the response is positive or negative.

Table 2.2 SMTP Reply Codes, Digit 1

Digit 1	Meaning
1yz	Positive Preliminary Reply
2yz	Positive Completion Reply
3yz	Positive Intermediate Reply
4yz	Transient Negative Completion Reply
5yz	Permanent Negative Completion Reply

The information described by the second and third digits is noted here. For the meanings of specific numbers, see RFC 821.

- The second digit supplies response categories, such as Syntax or Connections, that identify the general type of failure.
- The third digit provides more information to help distinguish between responses with the same first two digits. For example, note the variations in the 55x codes in Table 2.1, SMTP Reply Codes. In all of these codes, the command failed, but for different reasons the code was able to identify, such as “mailbox unavailable” (550) or “unavailable or unable to find user” (551).

[Top]

SMTP in the Messaging Access SDK

The SMTP class hierarchy is made up of the following classes.

- `netscape.messaging.smtp.ISMTPSink`. Interface for the SMTP response sink. See Creating a Response Sink.

- `netscape.messaging.smtp.SMTPClient`. Represents the SMTP client. See Creating a Client.
- `netscape.messaging.smtp.SMTPSink`. Convenience implementation of the `ISMTPSink` interface. See Creating a Response Sink.
- `netscape.messaging.smtp.SMTPException`. Exception thrown when an SMTP API error condition is detected by the Messaging Access SDK. Extends `IOException`.
- `netscape.messaging.smtp.SMTPServerException`. Exception thrown from the `error` callback on the response sink when the server sends an error. Extends `IOException` and `SMTPException`.

[Top]

SMTP Callback Mapping

Callbacks are associated with many SMTP methods. For general information about the response sink and callbacks, see "SDK Sink Classes for Java."

The `ISMTPSink` interface contains callbacks for each client call. The client's `processResponses` method invokes the interface method that corresponds to the client call. Methods with multi-line responses map to more than one callback. The second callback provides a notification that the operation is complete.

If a server error occurs, the error callback is invoked.

Table 2.3 shows which SMTP methods are mapped to callbacks in the `ISMTPSink` interface. Table 2.4 shows methods that do not map to callbacks.

Table 2.3 Methods with Callbacks

`SMTPClient` Methods	Possible Callbacks on `ISMTPSink`
`bdat`	`bdat, error`
`connect`	`connect, error`

SMTPClient Methods	Possible Callbacks on ISMTPSink
data	data, error
ehlo	ehlo, ehloComplete, error
expand	expand, expandComplete, error
help	help, helpComplete, error
mailFrom	mailFrom, error
noop	noop, error
quit	quit, error
rcptTo	rcptTo, error
reset	reset, error
send	send, error
sendCommand	sendCommand, sendCommandComplete, error
sendStream	send, error
verify	verify, error

Table 2.4 Methods without Callbacks

Methods Without Callbacks	
disconnect	setChunkSize
free	setPipelining
get_option	setResponseSink
initialize	setTimeout
processResponses	set_option

[Top] [SMTP Callback Mapping]

Creating a Response Sink

The first step in starting an SMTP session is to create the SMTP response sink, which is defined by the `ISMTPSink` interface. The response sink contains the callback methods for the SMTP client. For general information about the response sink, see "SDK Sink Classes for Java."

The `ISMTPSink` interface contains callbacks for each client call. You must implement this interface in order to use the SMTP client object. The constructor for the `SMTPClient` class takes an `ISMTPSink` object as a parameter.

To create your own response sink class, you can implement the `ISMTPSink` interface, using this syntax:

```
public class newSMTPSink extends Object
   implements ISMTPSink
```

As a convenience, the Messaging SDK provides the `SMTPSink` class, which implements the `ISMTPSink` interface. `SMTPSink` implements all the interfaces in `ISMTPSink`. By default, the implementation does nothing, except provide the `error` callback, which throws an exception. You can save a step by extending this class, using this syntax:

```
public class newSMTPSink extends SMTPSink{
   }
```

The following section of code creates a response sink.

```
SMTPSink l_smtpSink;

l_smtpSink = new SMTPSink();
```

After you create the response sink, the next step is Creating a Client.

[Top]

Creating a Client

The SMTP client uses an `SMTPClient` object to communicate with the server. To create the `SMTPClient` object and set the response sink for the client's use, call the `SMTPClient.SMTPClient` class constructor, which takes an existing response sink. Use this syntax:

```
public SMTPClient(ISMTPSink in_sink)
```

The following section of code creates a client.

```
/* Create sink first, as described in Creating a Response Sink */

SMTPClient l_client;

SMTPSink l_smtpSink;

l_client = new SMTPClient( l_smtpSink );
```

After you initialize the client, the next step is Connecting to a Server.

[Top]

Connecting to a Server

Before sending mail, the client must connect with the server through a service port. To connect to the server, call either of two `SMTPClient.connect` methods, depending on whether or not you want to specify the connection port. The SDK methods perform some error-checking.

To connect to the server using the default port for the SMTP protocol (port 25), use this form of `connect` and supply the identifier of the server:

```
public synchronized void connect(String in_server) throws IOException
```

To specify the server port to use for the server connection, use the other form of `connect`:

```
public synchronized void connect(String in_server,
                                 int in_port) throws IOException
```

On connecting, the server sends a greeting message to client. The client responds by identifying itself with the EHLO command.

Note For this method's callback mapping, see SMTP Callback Mapping. §

The following section of code connects the client to the server.

```
/* After Creating a Response Sink and Creating a Client */

l_client.connect( "smtpserver.com" );

l_client.processResponses();
```

During the connect process, you can enable pipelining if your server supports it. See Pipelining Commands. To find out which extensions are supported by the server, see Determining ESMTP Support. After connecting to the server, the next step is Setting the Mailer

To disconnect the client from the server, and close the socket connection, use this `SMTPClient` class method.

```
public synchronized void disconnect() throws IOException
```

You could use this function as part of a Cancel operation while retrieving a message. Remember that you do not call `processResponses` after `disconnect`. If an input or output error occurs, the method throws an `IOException`.

[Top]

Determining ESMTP Support

To retrieve a listing of extensions that are supported by the server, call the `SMTPClient.ehlo` method. This method returns a multiline message listing the Extended SMTP (ESMTP) features, such as pipelining or DSN, that the server supports. This is similar to the functionality of the `IMAP4Client.capability` command. Use this syntax:

```
public synchronized void ehlo(String in_domain)
                              throws IOException
```

This method calls the EHLO SMTP protocol command, which can be issued in any session state, but is usually issued after connecting to the server.

In Messaging Server 3.5, the developer must determine ESMTP support; in Messaging Server 4.0, this is optional.

Note For this method's callback mapping, see SMTP Callback Mapping. §

The following section of code finds out which Extended SMTP (ESMTP) features the server supports.

```
/* After Connecting to a Server */

l_client.ehlo( "yourdomain.com" );

l_client.processResponses();
```

You can enable pipelining if your server supports this extension. See Pipelining Commands.

[Top]

Pipelining Commands

Pipelining allows you to group, or batch, methods for execution rather than sending each separately. If pipelining is enabled on your server, commands are stored internally in the client as they are issued. All commands begin to execute when triggered in one of three ways: if the `SMTPClient.processResponses` method is called, if the internal storage area is full, or if a method that cannot be pipelined is issued.

You can enable pipelining anywhere, but it may make sense to do this after invoking the `SMTPClient.ehlo` method. Pipelining may then be enabled if the server supports it. If not, the way the network works does not change. The `ehlo` callback indicates whether pipelining is supported.

Use this syntax to attempt to enable pipelining:

```
public synchronized void setPipelining(
                    boolean in_enablePipelining)
                    throws SMTPException
```

The `in_enablePipelining` parameter is a Boolean value that tells the server to attempt to enable pipelining. The method throws an `SMTPException` if PIPELINING is not supported by the server. This method sends the PIPELINING SMTP protocol command.

Note For this method's callback mapping, see SMTP Callback Mapping. §

Some methods continue to add to the pipelining list. These are `SMTPClient.bdat`, `SMTPClient.mailFrom`, `SMTPClient.rcptTo`, and `SMTPClient.send`. Calling any other method causes the methods on the pipelining list to begin executing.

For example, you could call `mailFrom`, followed by one or more calls to `rcptTo`. These methods are added to the pipelining list and are not executed. If you then call another method, such as `noop`, the commands are sent to the server.

For details about using pipelining, refer to RFC 1854,"SMTP Service Extension for Command Pipelining."

[Top]

Setting the Mailer

Setting the mailer starts the process of delivering a message. If your server supports Extended SMTP, you can implement ESMTP elements in the `in_esmtpParams` parameter. Use the `SMTPClient.mailFrom` method:

```
public synchronized void mailFrom(String in_reverseAddress,
                        String in_esmtpParams)
                        throws IOException
```

This method identifies the sender and provides the sender's fully qualified domain name in the `in_reverseAddress` parameter. It sends the MAIL FROM SMTP protocol command. If an I/O error occurs, the method throws an `IOException`.

Note For this method's callback mapping, see SMTP Callback Mapping. §

The following section of code sets the mailer.

```
/* After Connecting to a Server */
SMTPClient l_client;
l_client.mailFrom( "sender@netscape.com", null );
l_client.processResponses();
```

After you set the mailer, the next step is Setting the Recipient.

[Top]

Setting the Recipient

After setting the mailer, the next step is to set the recipient. Use this `SMTPClient` class method:

```
public synchronized void rcptTo(String in_forwardAddress,
                                String in_esmtpParams)
                                throws IOException
```

This method sets a single recipient, so you must call it again for each recipient of a message. The `in_forwardAddress` parameter contains the recipient's address. If your server supports Extended SMTP, you can pass ESMTP elements in the `in_esmtpParams` parameter. If an input or output error occurs, the method throws an `IOException`.

`SMTPClient.rcpTo` sends the RCPT TO SMTP protocol command.

Note For this method's callback mapping, see SMTP Callback Mapping. §

The following section of code sets the recipient.

```
/* After Setting the Mailer */
l_client.rcptTo( "recipient@netscape.com", null );
l_client.processResponses();
```

After you set the recipient, the next step is Sending the Message.

[Top]

Sending the Message

After setting all of the message recipients, the client can send the message data. To send a message, use `SMTPClient.data`, followed by the `SMTPClient.send` method. First, use this method:

```
public synchronized void data()
                    throws IOException
```

The server responds with a success or failure reply code. See SMTP Response Codes.

The `SMTPClient.send` method delivers data to the server. If you use this method, you must send data with the `SMTPClient.data` method and not with `SMTPClient.bdat`. The `SMTPClient.bdat` method, which can deliver binary data, is not supported on the Netscape Messaging Server and some other servers.

After the `data` method, call `SMTPClient.send`, which sends a message to the server:

```
public synchronized void send(
      InputStream in_inputStream) throws IOException
```

The input stream contains the data to send. When the server responds that it is ready, the client sends the RFC 822 message data line by line.

You can set data chunk size with `SMTPClient.setChunkSize`, or you can use the default (1 K). You can set this at any point before the `SMTPClient.send` method.

Note For the callback mapping for these methods, see SMTP Callback Mapping. §

The following section of code uses `SMPTClient.data` and `SMPTClient.send` to send a message.

```
l_client.data();

l_client.processResponses();

l_client.send( new ByteArrayInputStream( "Hello World!!!" ) );

l_client.processResponses();
```

After you send the message and perform any other SMTP operations you need for the session, the next step is Ending the Session.

To use a more convenient, but less flexible, way to send messages, see Sending Messages with Convenience APIs.

[Top]

Sending Messages with Convenience APIs

The Messaging Access SDK provides two Convenience APIs that combine several message-handling operations in one step. The `ImTransport.sendMessage` and `ImTransport.sendDocuments` methods are for developers who want to mail-enable applications, such as spreadsheets and word processors, whose primary purpose is not messaging. Adding this mail functionality can allow the end user to mail documents or post them to a news group from within the application.

The `ImTransport.sendMessage` method is a convenience for sending messages. It lets the client use SMTP to send a message that is already in MIME format.

This method connects to the SMTP transport at the specified host, and submits a message created with the Netscape MIME API or in any other way. Use this syntax:

```
public String[] sendMessage(String host,
                            String sender,
                            String recipients[],
                            InputStream MIMEMessageStream)
                            throws IMException
```

You provide the names of the host and sender, and the email addresses of the recipients. The input stream contains the MIME message itself.

The Messaging Access SDK also provides an API for building and sending a message in a single step. For more information, see Sending Documents with the Convenience API.

[Top]

Ending the Session

When the client wants to end the session, the client should call `SMTPClient.quit` to notify the server. The server closes the TCP connection and returns a response code. You should always end a session with `quit` instead of just closing the connection. This method sends the QUIT SMTP protocol command:

```
public synchronized void quit() throws IOException
```

Note For this method's callback mapping, see SMTP Callback Mapping. §

The following section of code notifies the server that the client is terminating the session.

```
l_client.quit();

l_client.processResponses();
```

[Top]

Chapter

3

Building and Parsing MIME Messages

This chapter is an overview of using the MIME (Multipurpose Internet Mail Extension) API of the Messaging Access SDK to encode, decode, and parse mail messages, and handle text and non-text attachments.

- The MIME Protocol
- Structure of a MIME Message
- MIME in the Messaging Access SDK
- Steps in a MIME Session
- Building the MIME Message
- Encoding the Message
- Parsing MIME Messages

[Top]

The MIME Protocol

The MIME (Multipurpose Internet Mail Extension) protocol is the solution for sending multipart, multimedia, and binary data over the Internet. MIME is the standard for sending a variety of data types, including video, audio, images, programs, formatted documents, and text, in email messages.

The MIME protocol is made up of the extensions to the Internet mail format documented in RFC 822, “Standard for the Format of ARPA Internet Text Messages,” August 1982. The MIME protocol, documented in a series of MIME RFCs, adds these features:

- the ability to send rich information through the Internet
- the ability to encode and attach binary (non-ASCII) content to messages
- a framework for multipart mail messages that contain differing body parts
- a way to identify the content type associated with a message body part
- a standardized and interpretable set of body part types

MIME messages can include attachments and non-ASCII data. To conform with RFC 822, which requires mail message characters to be in ASCII, MIME uses an encoding algorithm to convert binary data to ASCII characters. For content that requires encoding, MIME specifies two encoding types, either Quoted-Printable or BASE64, which are described more fully in MIME Encoding Types.

In addition to the ability to build multimedia messages in MIME format, the MIME API of the Messaging Access SDK provides a parsing facility for messages. This generic MIME parser takes a MIME-encoded email message and decodes all or parts of it, depending on the preferences of the application. The MIME parser is described in Parsing MIME Messages.

For detailed information about MIME, consult one of the RFCs listed, with links, in MIME RFCs.

[Top]

MIME Encoding Types

MIME messages can include attachments and non-ASCII data. RFC 822 requires mail message characters to be in ASCII, so MIME uses an encoding algorithm to convert binary data to ASCII characters. MIME uses one of two encoding types, Quoted-Printable and BASE64 encoding.

Quoted printable encoding handles content that is mostly composed of ASCII characters, with only a small number that are non-ASCII (for example, Scandinavian characters in the ISO-8859-1 character set). This text is mostly readable on the client before it is encoded. The encoding process ignores ASCII characters and encodes the rest, using a set of rules for representing characters, line breaks, and tabs, and limiting line length.

BASE64 encoding handles binary data. This algorithm works by encoding sets of a octets into encoded characters, and produces 33 percent data expansion.

The MIME API also supports a non-encoding option. For example, no encoding is required for text messages.

For detailed information about MIME encoding, consult one of the RFCs listed, with links, in MIME RFCs.

[Top]

MIME Content Types

MIME types typically have three parts, a type, a subtype and optional content-type parameters. The type is the general content category; the subtype is the specific data format, as shown in these examples:

- `text/plain`: Text content (type) in `plain` format (subtype).
- `image/gif`: An `image` file (type) in `gif` format (subtype).

This table lists the valid MIME content types. A valid subtype can be of any data format type, including numerous experimental formats.

Table 3.1 MIME Content Types

Type	Description	Subtypes
Text	Information in raw text form. Has optional character set (default: `us-ascii`).	plain: includes no formatting information
Audio	Message body contains audio data.	basic
Image	Message body contains an image.	image format name, for example: `gif, jpeg`
Video	Message body contains a time-varying-picture image, possibly with color and sound.	image format name, for example: `mpeg`
Application	Uninterpreted binary data or information to be processed by an application.	octet-stream, postscript
Multipart	Messages with multiple attachments of potentially different media. Subtypes describe how the sub-parts relate.	mixed, alternative, digest, parallel
Message	Identifies a message.	rfc822, partial, external-body

The MIME implementation of the Messaging Access SDK provides methods that create these content types, add them to messages, and encode or decode them.

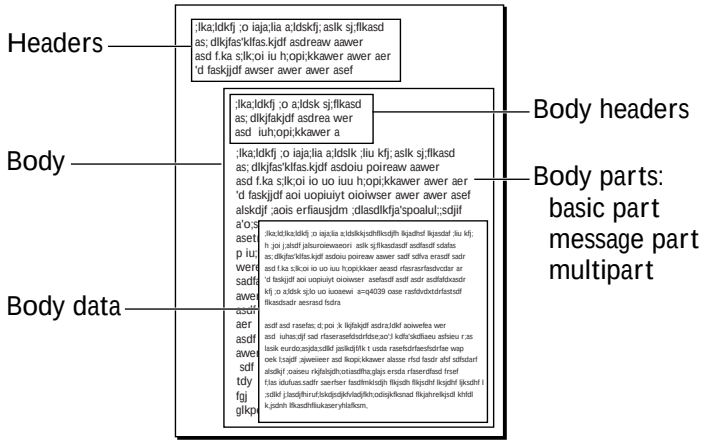
For detailed information about MIME content types, subtypes, and content parameters, consult one of the RFCs listed, with links, in MIME RFCs.

[Top]

Structure of a MIME Message

A MIME message has two main parts, the header and the body.

Figure 3.1 Parts of a MIME message



The message header consists of lines that describe the sender, subject, recipient, date, version of MIME in use, and a variety of other types of information, depending on the needs of the messaging application. This example shows the header lines of a message.

```
Return-Path:<Prasad@netscape.com>
   Received:from netscape.com ([205.217.229.85])by
            dredd.mcom.com (Netscape Messaging Server
            3.0) with ESMTP id AAA24896; Wed, 4 Feb 1998
            20:08:19 -0800
     Sender:prasad
 Message-ID:<34D93795.C1F48C83@netscape.com>
       Date:Wed, 04 Feb 1998 19:52:53 -0800
       From:Prasad Yendluri <Prasad@netscape.com>
  X-Mailer:Mozilla 4.03C-NSCP [en] (X11; U; SunOS 5.5.1
            sun4u)
MIME-Version:1.0
         To:sharonw@netscape.com
    Subject:Information about MIME
Content-Type:multipart/mixed; boundary =
            "------------ BFA9E722569728E3111F0326"
```

For more information, see Adding Message Headers.

The message body consists of body parts of different types, depending on the demands of the data in the message.

- Basic part. Includes all Basic MIME body part types: text, image, audio, video, and application. It does not include the Message-part or Multipart types. This is the simplest part.
- Multipart. Container part made up of two or more sub-body parts. The Multipart type includes several subtypes that describe how the sub-parts relate to each other (mixed, alternative, digest, and parallel).
- Message part. Message used as an attachment, for example, a message forwarded in another message.

For more information, see Building the MIME Message.

[Top]

MIME in the Messaging Access SDK

The MIME class hierarchy is made up of the following classes.

- `class netscape.messaging.mime.Header` (implements `java.lang.Cloneable`). Represents a message header.
- `class netscape.messaging.mime.MIMEBodyPart` (implements `java.lang.Cloneable`). Represents the body part of a message.
- `class netscape.messaging.mime.MIMEBasicPart` (implements `java.lang.Cloneable`). Represents Basic MIME BodyPart types: text, image, audio, video, and application, except structured parts such as `MIMEMessagePart` and `MIMEMultiPart`.
- `class netscape.messaging.mime.MIMEMessagePart` (implements `java.lang.Cloneable`). Represents a forwarded message.

- `class netscape.messaging.mime.MIMEMultiPart` (implements `java.lang.Cloneable`). Represents a message composed of several basic parts.
- `class netscape.messaging.mime.MIMEDataSink`. Represents the data sink for the MIME dynamic parser.
- `class netscape.messaging.mime.MIMEDynamicParser`. Represents the MIME dynamic parser.
- `class netscape.messaging.mime.MIMEHelper`. Contains MIME utility methods.
- `class netscape.messaging.mime.MIMEMessage`. (implements `java.lang.Cloneable`). Represents the data sink for the MIME message.
- `class netscape.messaging.mime.MIMEParser`. Represents the data sink for the MIME parser.
- `class netscape.messaging.mime.MIMEException`. Exception thrown when a MIME API error condition is detected by the Messaging Access SDK.
- `class netscape.messaging.mime.fileMIMEType`. Contains file type and encoding information.

Top]

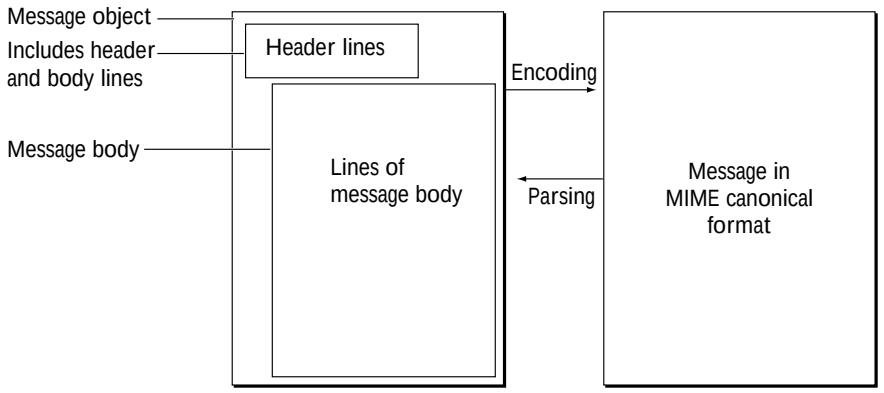
Steps in a MIME Session

The basic MIME operations focus on preparing a message to be sent and translating a received message into readable form for a mail application. Before the MIME message can be sent, the message and its attachments must be built and encoded in MIME format. When a message is received, it must be parsed and decoded.

Generally, a messaging application follows these steps when using MIME to build a message.

- Building the MIME Message.
- Encoding the Message.
- Decoding and parsing the Message. See Parsing MIME Messages.

Figure 3.2 Translating a MIME Message



[Top]

Building the MIME Message

Before sending a MIME message, you must build the message and its attachments and encode them in MIME format. This section describes the steps in building a message.

You can build MIME messages from RFC headers and simple text or attachments or both. You have the option of either building the message part by part, or using one of the convenience methods provided by the API.

There are two stages in building a message.

First, create the message object. A MIME message is an instance of the `MIMEMessage` class. The methods of this class create the message, add headers and body parts, return information about message attributes, and encode the message.

To create a `MIMEMessage` object, call one of the `MIMEMessage` class constructors.

- `MIMEMessage.MIMEMessage()`. Creates a default message.
- `MIMEMessage.MIMEMessage(Header[])`. Creates a message, given a set of RFC 822 headers.
- `MIMEMessage.MIMEMessage(InputStream, String, int)`. Creates a multipart `MIMEMessage` with the specified text and file.

Then, add two components, in either order.

- Add the message headers. Headers are `name:value` pairs that conform to the requirements of RFC 822. For more information, see Adding Message Headers.
- Add the content. Content can simply be text or it can include several parts or file attachments. For more information, see Adding Content to the Message.

You can also create a message with `ImTransport.sendDocuments`, a convenience method for building and mailing documents. This method builds a MIME message, given headers, host, recipients, and other basic message information, then connects to the SMTP transport and submits the message. For more information, see Sending Documents with the Convenience API.

One of the sample applications in the `examples` directory of the SDK illustrates building a message using this method.

The following section of code demonstrates using `MIMEMessage.MIMEMessage` to build a MIME message with text from an input stream, a data buffer, and the default encoding type.

```
/* Get an inputStream to user entered text */

bins = new ByteArrayInputStream (textMsg.getBytes());

/* Create a new Multipart MIMEMessage with the above text and the file
passed; -1 selects default encoding */

mmsg = new MIMEMessage(bins, fullfilename, -1);
```

Next, add the headers to the message. See Adding Message Headers.

[Top]

Adding Message Headers

After building a message, add the message headers. A MIME header is an instance of the `Header` class. The methods of this class create the header, and get and set header information.

To create a `Header` object, call one of the `Header` class constructors. These methods create a header entry as a `name:value` pair.

- `Header.Header(byte[])`. Creates a default header from ASCII characters.
- `Header.Header(String)`. Creates a message, given an ASCII string that contains a header in `name:value` format.
- `Header.Header(String, String)`. Creates a `header`, given a name and value.

This method creates a header entry. You supply the header name and value. This method creates a header entry as a `name:value` pair.

```
public Header(String name,
         String value) throws MIMEException
```

Alternatively, you can add a header to a message, using either the `MIMEMessage.addHeader` or the `MIMEMessage.setHeader` method. The `addHeader` method adds the specified header to the message. If a header with the specified name already exists, it appends the value to the current header value.

```
public void addHeader(String name,
         String value) throws MIMEException
```

The `MIMEMessage.setHeader` method sets any RFC 822 headers including X-headers. If a header exists, it overwrites the existing value.

```
public void setHeader(String name,
         String value) throws MIMEException
```

The following section of code creates RFC 822-compliant headers for a MIME message.

```
/* Set user-entered RFC822 headers to a message (mmsg). */

mmsg.setHeader ("From", sender);
mmsg.setHeader ("Reply-To", sender);
mmsg.setHeader ("To", To);
mmsg.setHeader ("Subject", subject);

/* Add any other desired headers. */

mmsg.setHeader ("X-MsgSdk-Header", "This is a Text Message");
```

Next, you can add other message parts or attachments as needed to the message. See Adding Content to the Message.

[Top]

Adding Content to the Message

To add content to a message, you first create a message body part that contains data, then use `MIMEMessage` methods to add this part to the message. A MIME message can include the following types of body parts:

- Basic part, a `MIMEBasicPart` object. The simplest basic part is text that you type in. Other basic parts are audio, video, image, or application files. To add data to a basic part, use the `MIMEBasicPart.setBodyData` method.
- Multipart, a `MIMEMultiPart` object. A multipart contains two or more basic parts. Use `MIMEBasicPart` methods to build a basic part. Use the `MIMEMultiPart.addBodyPart` method to add the basic part to the multipart. Then you can add the constructed multipart to the message. Use this content type if you are constructing a message that has more than one attachment.
- Message part, a `MIMEMessagePart` object. A message part is a message that becomes an attachment, for example, when it is forwarded. You can use the `MIMEMessagePart.setMessage` method to add the message to the message part.

Creating the Basic Part and Adding Data

Before you can add a basic part, you first create the basic part object. A MIME basic part is an instance of the `MIMEBasicPart` class. This is the common structure for the leaf parts, text, audio, video, image, and application. Use one of the two class constructors.

- `MIMEBasicPart.MIMEBasicPart()`. Creates a default basic part with the "Text" content type.
- `MIMEBasicPart.MIMEBasicPart(int)`. Creates a basic part, given the content type. Set the attributes of the basic part as required. For example, use `MIMEBasicPart.setContentID` to set the content ID.

After you create the basic part object and its attributes, you can add the body data. To add data to an existing basic part, use one of the `MIMEBasicPart.setBodyData` methods, depending on the source of the data.

- `MIMEBasicPart.setBodyData(InputStream is)`. Sets the body data of this basic part from an input stream.
- `MIMEBasicPart.setBodyData(byte s[])`. Sets the body data of this basic part from a buffer.

The following section of code demonstrates adding data to a body part.

```
byte() bodyData = userTxt.getBytes()
/* userTxt is a user-entered text string */

MIMEBasicPart bp1 = new MIMEBasicPart (MIMEBasicPart.Text);

bp1.setBodyData(bodyData);
```

Now you can add this part to the message. See Adding Parts to the Message.

[Top]

Adding Body Parts to a Multipart

If a message has two or more attachments, you must create a multipart that includes them before you can add them the message.

- First, create each part as a basic part. For more information, see Creating the Basic Part and Adding Data.
- Then create a multipart and add each basic part to it.
- After this, you can add this multipart to the message.

To add an existing body part to a multipart, use this `MIMEMultiPart` method:

```
public int addBodyPart(MIMEBodyPart part,
                  boolean clone) throws MIMEException
```

Supply the body part type you are adding, either a `MIMEBasicPart`, a `MIMEMultiPart`, or a `MIMEMessagePart`.

The following section of code demonstrates adding a body part to a multipart.

```
/* Create the basic part */

MIMEBasicPart bp1 = new MIMEBasicPart();

MIMEBasicPart bp2 = new MIMEBasicPart(MIMEBasicPart.AUDIO);

/* Set bodyData of bp1 and bp2; set attributes of bp1 and bp2 */
/* See Creating the Basic Part and Adding Data for details */

/* Create the multipart */

MIMEMultiPart mp = new MIMEMultiPart();

mp.addBodyPart(bp1, false);

mp.addBodyPart(bp2, false);

/* Set attributes of multipart */

mp.setContentSubType ("Mixed");
```

After creating and assembling the multipart, the next step is to add the multipart to the message. See Adding Parts to the Message.

[Top]

Creating a Message Part

When you forward a message, it becomes the content of another message. This means that the mail application must do two things:

- Make a message part from the message structure to be forwarded.
- Add it as the content (body) of the message to be sent.

To make a message part from the message structure, use the `MIMEMessagePart` constructor:

```
public MIMEMessagePart( MIMEMessage msg) throws MIMEException
```

For the `msg` parameter, supply the `MIMEMessage` object for the message to be forwarded. Alternatively, you can create a message part and then add the message to be forwarded to it, as shown here:

```
MIMEMessagePart msgPart = new MIMEMessagePart();

msgPart.setMessage(msg);

public void setMessage(MIMEMessage msg,
               boolean clone) throws MIMEException
```

Supply the message. The `clone` parameter should contain `true` if the function should clone a copy of the message or `false` if it should store a reference to the passed object.

[Top]

Adding Parts to the Message

After you create the MIME message object and the body parts you want it to include, use `MIMEMessage` methods to add the parts to the message. For information about creating the message, see Adding Content to the Message.

When a basic part, multipart, or message part is complete and includes data, you can add it to the message with the `MIMEMessage.setBody` method:

```
public void setBody(MIMEBodyPart part,
            boolean clone) throws MIMEException
```

You supply the body part type, either `MIMEBasicPart`, `MIMEMultiPart`, or `MIMEMessagePart`, and this part becomes the body of the message. For information about constructing a basic part, see Creating the Basic Part and Adding Data.

You can simplify the process of building and adding content to a message by using the `MIMEMessage` constructor that takes a new stream and file name. You can also use the `ImTransport.sendDocuments` Convenience API to build and send MIME messages from files and memory-based buffers. You supply message content and other attributes. The method creates a message, then connects to the SMTP transport and submits it. For more information, see Sending Documents with the Convenience API.

[Top]

Deleting Parts of a Message

If you want to delete the message, a body part, or parts of it, after it is built, use one of the MIME delete methods. You can delete the entire message, the body, or a message part, as needed.

- `MIMEBasicPart.deleteBodyData`. Deletes the body data for a part.
- `MIMEMultiPart.deleteBodyPart`. Deletes a body part from a multipart.
- `MIMEMessagePart.deleteMessage`. Deletes a MIME message that is the body of a message part.
- `MIMEMessage.deleteBody`. Deletes the body of a message.

[Top]

Encoding the Message

After building a message, the next step is to encode it. You can encode an entire message, with headers and message attachments, in one operation with the `MIMEMessage.putByteStream` method. Encoding generates a byte stream in MIME canonical form, so that it can be transmitted over SMTP and other transport methods.

```
public void putByteStream(OutputStream os)
               throws IOException, MIMEException
```

This method encodes the data and writes it to the specified MIME output stream.

If you need to encode only a message body part, use the `putByteStream` method for that part. `MIMEMessage.putByteStream` internally invokes the `putByteStream` method for each constituent body part as needed.

The following section of code demonstrates using the `putByteStream` method.

```
FileOutputStream fos = new fileInputStream(“<fileName>”);

mmsg.putByteStream (fos);
```

[Top]

Encoding and Decoding Utilities

The `MIMEHelper` class provides a number of utility methods for encoding and decoding. These methods are used by other MIME API methods internally, and are also made available to developers to use in their applications.

The `MIMEMessage.putByteStream` method could call either of two encoding utility methods, `MIMEHelper.encodeBase64` or `MIMEHelper.encodeQP`, based on whether the requested encoding type is Base64 (default for non-text types) or Quoted Printable. These `MIMEHelper` utility methods each provide a single form of encoding. Like `MIMEMessage.putByteStream`, these methods take an input stream and encode it. If an application requires only Base64 or QP-encoded data, you can use one of these methods in place of `MIMEMessage.putByteStream`.

- `MIMEHelper.encodeBase64`. Base64 encodes data and writes it to an output stream.
- `MIMEHelper.encodeQP`. Quoted Printable encodes the data from an input stream and writes to an output stream.

The `MIMEParser.parseEntireMessage` method, which parses and decodes encoded messages, could call either of two decoding utility methods, `MIMEHelper.decodeBase64` or `MIMEHelper.decodeQP`, based on the requested encoding type. These `MIMEHelper` utility methods each provide a single form of decoding, and could be used instead of `parseEntireMessage` if an application only needs to decode Base64 or QP-encoded data.

- `MIMEHelper.decodeBase64`. Base64 decodes the data from an input stream and writes to an output stream.
- `MIMEHelper.decodeQP`. Quoted Printable decodes the data from an input stream and writes to an output stream.

[Top]

Encoding and Decoding Headers

Two utility methods allow you to encode and decode only the headers of a message.

- `MIMEHelper.encodeHeader`. Encodes an RFC 2047-compliant header from an input stream, using Base64 or Q encoding. You can select the character set for the encoding operation. The header string can be used as the value of unstructured headers or in the comments section of structured headers.
- `MIMEHelper.decodeHeader`. Decodes an RFC 2047 header of a message.

[Top]

Sending Documents with the Convenience API

The Messaging Access SDK provides convenience APIs that combine several message-handling operations in one step. These can be helpful when you are mail-enabling an otherwise mail-ignorant application.

`ImTransport.sendDocuments` is a convenience method for mailing documents. This method builds a MIME message with the specified parameters by automatically detecting the MIME types. It then connects to the SMTP transport at the specified host, and submits the message. If the message has more than one attachment, it is sent as a MIME message of multipart/mixed type.

```
public String[] sendDocuments(String host,
                        String sender,
                        String recipients[],
                        String subject,
                        String[] msgHeaderNames,
                        String[] msgHeaderValues,
                        IMAttachment [] attachments,
                        boolean fUseTempFiles)
                        throws IMException
```

You provide the names of the host and sender, the email addresses of the recipients, the subject of the message, header information, any message attachments. If you set the `fUseTempFiles` parameter to `true`, `sendDocuments` uses temporary intermediate files for some internal processing, for better performance. If you don't want to create temporary files, set this flag to `false`.

For other purposes, or for more sophisticated email requirements, use the `ImTransport.sendMessage` method in association with the Netscape MIME API or other Messaging APIs provided by Netscape.

The Messaging Access SDK also provides an API for sending a message that is already in MIME format in a single step. For more information, see Sending the Message.

[Top]

Parsing MIME Messages

For parsing encoded messages, the Messaging Access SDK provides these options:

- Parsing the Entire Message. Use this option when the message to be parsed is available in its entirety when you begin parsing.
- Dynamic Parsing. Use the dynamic parser when the entire message is not available when you begin parsing, but becomes available block by block. This could happen when you are receiving a message from a server.

[Top]

Parsing the Entire Message

You can use `MIMEParser` class methods to parse and decode encoded messages retrieved through email protocol APIs, such as POP3 and IMAP4. First, create a `MIMEParser` object; then call `MIMEParser.parseEntireMessage`.

This method parses an entire MIME message in one operation and returns the parsed message:

```
public MIMEMessage parseEntireMessage(
            InputStream input) throws MIMEException
```

Supply the identifier of the input stream for the message.

The following section of code uses `MIMEParser.parseEntireMessage` as part of a routine that parses an entire file.

```
MIMEMessagePart msg = parseEntireMessage(inputStream);
```

[Top]

Dynamic Parsing

This section describes the steps involved in using the dynamic parser. Dynamic parsing contrasts with standard MIME parsing, as described in Parsing the Entire Message, in several ways.

- The dynamic parser can parse a message in chunks, rather than in its entirety, in a single operation. The dynamic parser decodes the message on the fly, passing the data to the user right away without waiting for the whole message.
- The dynamic parser returns parsed message data to the caller using callbacks in the data sink. For information about this, see Creating a Data Sink. The `MIMEParser.parseEntireMessage` method does not use callbacks; instead, it passes the entire parsed message to the user after parsing is complete.
- The dynamic parser does not decode the Base64/QP-encoded parts of the message. To do this, use the utility methods in the `MIMEHelper` class.

[Top]

Steps in Dynamic Parsing

Using the dynamic parser involves these operations:

- Create a `MIMEDataSink` object and callback methods for the parser. The MIME data sink contains one call for each piece of information that the parser can return. For example, for a header, it contains a header callback method. See Creating a Data Sink.
- Create a parser object, which takes the data sink as a parameter. See Creating the Dynamic Parser.
- Begin parsing. See Running the Parser.

- As long as there is more data to parse, continue to call a dynamic parsing method that matches the source of the data. Keep calling this method until there is no more data. See Running the Parser.
- When there is no more data to parse, indicate that parsing is complete. These steps are described in Running the Parser.

All dynamic parser methods are defined in the `MIMEDynamicParser` class.

[Top]

MIME Data Sink Callbacks

Callbacks operate in the same way in MIME as they do in other Messaging Access SDK protocols. However, for SMTP, IMAP4, and POP3, callbacks are tied to server responses to individual functions.

The dynamic parser data sink differs from the response sink in that, with a response sink, the mail application sends a command and gets a server response. In the data sink, when a callback takes place, the data is passed to the sink in callbacks. The callback is dependent upon the data in the input stream.

The MIME data sink contains one callback prototype for each piece of information that the parser can return. The dynamic parser makes callbacks based on the kind of data it finds in its input stream. For example, if the parser finds a header, the result is a header callback. As the parser encounters data, it returns information to the caller through callbacks in the data sink.

For general information about the data sink, response sinks, and callbacks, see SDK Sink Classes for Java.

[Top]

Creating a Data Sink

The first step in starting the MIME parser session is to create and initialize the MIME data sink. To do this, extend the `MIMEDataSink` abstract class. The `MIMEDataSink` class contains null body callbacks for all methods. For general information about the data sink, see SDK Sink Classes for Java.

After creating the data sink, the application passes it to the parser. As the parser encounters information, it sends this on to the caller through callbacks in the data sink. The MIME data sink contains a call for each piece of information that the parser can return.

The following section of code demonstrates creating a data sink.

```
public class myDataSink extends MIMEDataSink
{
   public MIMEMessage m_mimeMessage;
   /* Constructor */

   public myDataSink()
   {
      super();

   public void header( Object callbackObject,
               byte[] name, byte[] value )
   {
      show ("header name = " + new String(name) +
               "value = " + new String(value));
   }

   public void contentType( Object callbackObject,
               int nContentType )
   {
      show("contentType()=" + nContentType);
   }

   public void contentSubType( Object callbackObject,
               byte[] contentSubType )
   {
      show("contentSubType()=" + new String (contentSubType));
   }

   public void contentTypeParams( Object callbackObject,
               byte[] contentTypeParams )
   {

      show("contentTypeParams()" + new String(contentTypeParams));

   }

   public void contentID( Object callbackObject,
               byte[] contentID )
```

```
   {
      show("contentID()" + new String(contentID));
   }
/* Processing continues... */
}
```

After the data sink is created, the mail application can pass it to the parser when this object is created. As the parser encounters information, it goes through the data sink, returning information to the caller through callbacks.

You can create parsers with different data sinks, based on what you want the messaging application to do. For example, you can define data sinks that create a brief header, a normal header, or list all header lines, each of which can be invoked with a different parser invocation. To add headers, define the ones your application requires within the data sink structure.

After you create the data sink, the next step is Creating the Dynamic Parser.

[Top]

Creating the Dynamic Parser

After creating the data sink, the next step is to create a dynamic parser. You can use the `MIMEDynamicParser` class constructor to create a new parser and identify the data sink to use.

```
public MIMEDynamicParser(MIMEDataSink dataSink) throws MIMEException
```

The following section of code creates a dynamic parser.

```
/* Initialize sink first, as described in Creating a Data Sink */

MIMEDynamicParser mdp = new MIMEDynamkcParser(myDataSink);
```

After you create the dynamic parser, the next step is Running the Parser.

[Top]

Running the Parser

After the dynamic parser object has been created, as described in Creating the Dynamic Parser, parsing can begin. The parsing operation should continue until no more data is available, then signal that the parsing is complete.

To begin parsing, use the `MIMEDynamicParser.beginParse` method:

```
public void beginParse() throws MIMEException
```

This method starts a new parse cycle and resets the parser's internal data structures.

To continue parsing, use the `MIMEDynamicParser.parse` method:

```
public void parse(InputStream input) throws MIMEException
```

This method requires the input stream for the data to parse. Continue to call this method until there is no more data left.

When no more data remains to be parsed, call this method to indicate that parsing is complete. The parser ends the parse operation.

```
public void endParse() throws MIMEException
```

To initiate another parsing cycle, you can call `beginParse` again.

The following section of code creates a dynamic parser, parses data from an input stream, and ends the parse operation.

```
Class myDataSink extends MIMEDataSink;
   /* Implement the methods of myDataSink as needed */
   myDataSink dataSink = new myDataSink();
   /* Create the dynamic parser; see Creating the Dynamic Parser */
   MIMEDynamicParser mdp = new MIMEDynamicParser(dataSink);
   /* Start dynamic parsing */
   mdp.beginParse();
   /* Continue dynamic parsing until no more data remains */
   while (not done)
   {
      mdp.parse (data to parse);
   }
   /* When data is finished, stop the dynamic parser */
```

```
mdp.endParse ();
```

[Top]

Chapter

4

Receiving Mail with IMAP4

This chapter is an overview of using IMAP4 (Internet Message Access Protocol 4) to retrieve and manage messages remotely.

- The IMAP4 Protocol
- IMAP4 Callback Mapping
- Creating a Response Sink
- Creating a Client
- Connecting to a Server
- Logging In and Out
- Checking for New Messages
- Searching for Messages
- Fetching Message Data
- Closing a Mailbox

[Top]

The IMAP4 Protocol

IMAP4 (Internet Message Access Protocol, Version 4), which was developed at the University of Washington, allows clients to retrieve and manage their email messages remotely. This can be very helpful to users who access mail on several different computers.

IMAP4 also provides these capabilities:

- Filing: You can create folders, called “mailboxes,” on the server, manage mail messages on the server (search, delete, rename), and transfer messages from one folder to another on the server.
- Searching: You can find the messages that meet specified criteria on the server without downloading messages to the client

To send mail, use SMTP (Simple Mail Transport Protocol). For more information, see Chapter 2, “Sending Mail with SMTP.”

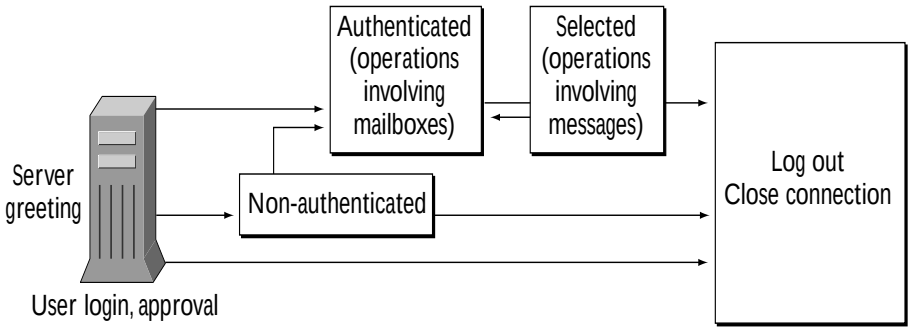
For detailed information about IMAP4, consult one of the RFCs listed, with links, in IMAP4 RFCs.

[Top]

IMAP4 Session States

An IMAP4 session progresses through several stages, or states. Within each state, only certain commands are possible.

Figure 4.1 IMAP4 Session States



Note: All states can result in logging out and closing the connection in response to the logout command, server shutdown, or a closed connection.

Table 4.1 IMAP4 Session States and Commands

Session State	Commands
All States	Commands: CAPABILITY, LOGOUT, NOOP
Non-Authenticated	Before login. User login, approval. Command: LOGIN
Authenticated	User is logged in, can perform operations involving mailboxes and mailbox management. Commands: APPEND, CREATE, DELETE, EXAMINE, LIST, LSUB, RENAME, SELECT, STATUS, SUBSCRIBE, UNSUBSCRIBE
Selected	Operations involving messages. Commands: CHECK, CLOSE, COPY, EXPUNGE, FETCH, SEARCH, STORE, UID

The client must keep track of the current session state in order to know which commands are valid.

For a table of SDK-supported IMAP4 protocol commands that lists the state in which each can be called, see Supported IMAP4 Internet Protocol Commands. For detailed information about IMAP4 and IMAP4 session states, consult one of the RFCs listed, with links, in IMAP4 RFCs.

[Top]

Steps in an IMAP4 Session

Generally, a messaging application follows these steps when using IMAP4 to receive mail and manage mailboxes and messages. These steps are listed below with links to more detailed descriptions.

Step	Section with details
Create a response sink.	Creating a Response Sink
Create a client.	Creating a Client
Connect to the server.	Connecting to a Server
Log in.	Logging In and Out
Check for new messages.	Checking for New Messages
Select a mailbox.	Searching for Messages
Fetch a new message.	Fetching Message Data
Perform other message and mailbox management tasks.	IMAP4 in the Messaging Access SDK
Close the mailbox.	Closing a Mailbox

[Top]

IMAP4 in the Messaging Access SDK

The IMAP4 class hierarchy is made up of the following classes.

- `netscape.messaging.imap4.IIMAP4Sink`. Interface for the IMAP4 response sink. See Creating a Response Sink.

- `netscape.messaging.imap4.IMAP4Client`. Represents the IMAP4 client. See Creating a Client.
- `netscape.messaging.imap4.IMAP4Sink`. Implementation of the `IIMAP4Sink` interface. See Creating a Response Sink.
- `netscape.messaging.imap4.SystemPreferences`. Sets and returns preferences for block size, default port, debug flag, and time-out length.
- `netscape.messaging.imap4.IMAP4Exception`. Exception thrown when an IMAP4 API error condition is detected by the Messaging Access SDK. Extends `IOException`.
- `netscape.messaging.imap4.IMAP4ServerException`. Exception thrown from the `error` callback on the response sink when the server sends an error. Extends `IOException` and `IMAP4Exception`.

[Top]

IMAP4 Callback Mapping

Callbacks are associated with many IMAP4 methods. For general information about the response sink and callbacks, see SDK Sink Classes for Java.

The `.IIMAP4Sink` interface contains callbacks for each client call. The client's `processResponses` method invokes the interface method that corresponds to the client call.

Methods with multi-line responses map to two or more callbacks. For example, when a method is mapped to three callbacks, the first provides a notification of the start of the operation, the second of the response, and the third that the operation is complete.

Many IMAP4 methods generate a tag (`out_ppTagID`) that you can use to help match the command and the response associated with it within the within the `IMAP4Client.taggedLine` response.

If a server error occurs, the error callback is invoked.

Table 4.2 shows which IMAP4 methods are mapped to callbacks in the `IIMAP4Sink` interface.

Table 4.2 Methods with Callbacks

`IMAP4Client` Methods	Possible Callbacks on `IIMAP4Sink`
General Commands	
`connect`	`ok, error`
`disconnect`	`bye`
`sendCommand`	`rawResponse, taggedLine, error`
Non-Authenticated State Commands	
`capability`	`capability, taggedLine, error`
`noop`	`exists, expunge, recent, fetchStart, fetchFlags, fetchEnd, taggedLine, error`
`login`	`taggedLine, error`
`logout`	`bye, taggedLine, error`
Authenticated State Commands	
`append`	`taggedLine, error`
`create`	`taggedLine, error`
`delete`	`taggedLine, error`
`examine`	`flags, exists, recent, ok, taggedLine, error`
`list`	`list, taggedLine, error`
`lsub`	`lsub, taggedLine, error`
`rename`	`taggedLine, error`
`select`	`flags, exists, recent, ok, taggedLine, error`
`status`	`statusMessages, statusRecent, statusUidnext, statusUidvalidity, statusUnseen, taggedLine, error`
`subscribe`	`taggedLine, error`
`unsubscribe`	`taggedLine, error`
Selected State Commands	
`check`	`taggedLine, error`
`close`	`taggedLine, error`

IMAP4Client Methods	Possible Callbacks on IIMAP4Sink
copy	taggedLine, error
uidCopy	taggedLine, error
expunge	expunge, taggedLine, error
fetch	fetchStart, fetchEnd, fetchSize, fetchData, fetchFlags, fetchBodyStructure, fetchEnvelope, fetchInternalDate, fetchHeader, fetchUid, taggedLine, error
uidFetch	fetchStart, fetchEnd, fetchSize, fetchData, fetchFlags, fetchBodyStructure, fetchEnvelope, fetchInternalDate, fetchHeader, fetchUid, taggedLine, error
search	searchStart, search, searchEnd, taggedLine, error
uidSearch	searchStart, search, searchEnd, taggedLine, error
store	taggedLine, error
uidStore	taggedLine, error
Extended IMAP Commands	
nameSpace	nameSpaceStart, nameSpacePersonal, nameSpaceOtherUsers, nameSpaceShared, nameSpaceEnd, taggedLine, error
setACL	taggedLine, error
deleteACL	taggedLine, error
getACL	aclStart, aclIdentifierRight, aclEnd, taggedLine, error
myRights	myRights, taggedLine, error
listRights	listRightsStart, listRightsRequiredRights, listRightsOptionalRights, listRightsEnd, taggedLine, error

[Top] [IMAP4 Callback Mapping]

Creating a Response Sink

The first step in starting an IMAP4 session is to create the IMAP4 response sink, which is defined in the `IIMAP4Sink` interface. For general information about the response sink, see SDK Sink Classes for Java.

The `IIMAP4Sink` interface contains callbacks for each client call. You must implement this interface in order to use the IMAP4 client object. The constructor for the `IMAP4Client` class takes an `IIMAP4Sink` object as a parameter.

To create a response sink class, you can implement the `IIMAP4Sink` interface. Use this syntax:

```
public class ResponseSink
   implements IIMAP4Sink {
   /* implementation of all methods declared in the sink */
   }
```

As a convenience, the Messaging SDK provides the `IMAP4Sink` class, which implements the `IIMAP4Sink` interface. `IMAP4Sink` implements all the interfaces in `IIMAP4Sink`. By default, the implementation does nothing, except provide the `error` callback, which throws an exception. You can save time by extending this class, using this syntax:

```
public class ResponseSink extends IMAP4Sink{
   }
```

The following section of code creates a response sink.

```
ResponseSink l_sink = new ResponseSink();
```

After you create the response sink, the next step is Creating a Client.

[Top]

Creating a Client

The IMAP4 client uses an `IMAP4Client` object to communicate with the server. To create the `IMAP4Client` object and set the response sink for the client's use, call the `IMAP4Client.IMAP4Client` class constructor, which takes an existing response sink. Use this syntax:

```
public IMAP4Client(IIMAP4Sink in_sink)
```

The following section of code creates a client.

```
/* After Creating a Response Sink */

IMAP4Client l_client = new IMAP4Client(l_sink);
```

After you initialize the client, the next step is Connecting to a Server.

[Top]

Connecting to a Server

Before retrieving mail, the client must connect with the server through a service port. To connect to the server, call either of the two `connect` methods in the `IMAP4Client` class, depending on whether or not you want to specify the connection port.

To connect to the server using the default port (143) for the IMAP4 protocol, use this form of `connect` and supply the identifier of the server:

```
public synchronized boolean connect(String in_IMAPHost)
                                    throws IOException
```

To specify the server port to use for the server connection, use the other form of `connect`. With this form, you can pass in the port number as well as the server identification, as follows:

```
public synchronized boolean connect(String in_IMAPHost,
                                    int in_portNumber)
                                    throws IOException
```

These methods generate tags that you can use to help match the commands and the responses associated with them.

The following section of code connects the client to the server.

```
/* After Creating a Response Sink and Creating a Client */

l_client.connect ( “HOSTNAME”, 143);

l_client.processResponses();
```

During the connect process, you can find out what extensions the server supports. For more information, see Determining Server Capabilities. You also might want to log in. For more information, see Logging In and Out.

To disconnect the client from the server and close the socket connection, use this `IMAP4Client` class method:

```
public synchronized void disconnect() throws IOException
```

You could use this method as part of a Cancel operation while retrieving a message. Remember that you do not call `processResponses` after `disconnect`.

Note For the callback mapping for these methods, see IMAP4 Callback Mapping. §

[Top]

Determining Server Capabilities

To retrieve a listing of the capabilities that are supported by the server, call the `IMAP4Client.capability` method:

```
public synchronized String capability() throws IOException
```

This method calls the CAPABILITY IMAP4 protocol command, which can be issued in any session state, but is usually issued after connecting to the server.

Note For this method's callback mapping, see IMAP4 Callback Mapping. §

The following section of code retrieves a list of server capabilities.

```
/* After Connecting to a Server */

l_client.capability();

l_client.processResponses();
```

[Top]

Logging In and Out

Once the client is connected to the server, the user can log in. Login identifies the client to the server. Logging in requires the user ID and the plain text password that authenticates this user. Use this `IMAP4Client` class method:

```
public synchronized String login(String in_user,
                  String in_password) throws IOException
```

The method sends the LOGIN IMAP4 protocol command, which can be issued during the Non-Authenticated state. Successful login moves the IMAP4 session to the Authenticated state, where the user can search for messages and manage messages on the server. This method generate a tag that you can use to help match the command and the response associated with it.

The following section of code logs the user in with user name and password.

```
l_client.login(“userid”, “password”);

l_client.processResponses();
```

After you log in, the next step is Checking for New Messages.

To log out at the end of a session, use this `IMAP4Client` class method:

```
public synchronized String logout() throws IOException
```

The following section of code logs the user out.

```
l_client.logout();

l_client.processResponses();
```

Note For the callback mapping for these methods, see IMAP4 Callback Mapping. §

[Top]

Checking for New Messages

Most IMAP4 servers check for messages whenever a command is issued. In the absence of commands, the server does not check for messages and may disconnect. To keep the server open indefinitely and check for messages periodically, the developer can call `IMAP4Client.noop` at set intervals.

The `noop` method is ideal for polling for new mail and ensuring that the server connection is still active. `noop` does nothing in itself, so it only produces the side effect of resetting the autologout timer inside the server and retrieving unsolicited server responses, which all commands do. The server responses may indicate the arrival of new messages or a change in the attributes of an existing message. Use this method:

```
public synchronized String noop() throws IOException
```

Note For this method's callback mapping, see IMAP4 Callback Mapping. §

The following section of code uses `IMAP4Client.noop` to check for messages.

```
l_client.noop();
l_client.processResponses();
```

After checking for new messages, the next step is Searching for Messages.

[Top]

Searching for Messages

The `IMAP4Client` class provides two ways to search for messages while in the Selected state. Two methods, `search` and `uidsearch`, search the currently selected mailbox and return the message numbers of messages that match a search key. These numbers can be used in turn to fetch the messages themselves.

You can supply one or more of the search keys defined in RFC 2060, section 6.4.4. Place more than one search key in a parenthesized list.

The `IMAP4Client.search` method searches the mailbox for messages that match the search criteria and returns their message numbers:

```
public synchronized String search(String in_criteria)
```

This method sends the SEARCH IMAP4 protocol command, which can be issued in the Selected session state.

The `IMAP4Client.uidSearch` method retrieves the message numbers that match the search criteria in the currently selected mailbox:

```
public synchronized String uidSearch(String in_criteria)
                                     throws IOException
```

This method uses the UID IMAP4 protocol command to specify that the SEARCH command uses unique message identifiers rather than sequence numbers.

Both methods generate tags that you can use to help match the command and the response associated with it.

Note For the callback mapping for these methods, see IMAP4 Callback Mapping. §

The following section of code searches for messages that have the SUBJECT “afternoon meeting.”

```
l_client.search(“SUBJECT \”Afternoon Meeting\””);

l_client.processResponses();
```

After locating messages, the next step is Fetching Message Data.

[Top]

Fetching Message Data

IMAP4 provides two methods that fetch messages while in the Selected state. `IMAP4Client.fetch` and `IMAP4Client.uidfetch` both search the currently selected mailbox and retrieve the data specified by the fetch criteria.

When you fetch messages, you supply the message set (mailbox) and one or more fetch criteria, placing more than one data item in a parenthesized list. The fetch criteria determine the information that is returned. You can fetch one or more of the data items defined in RFC 2060, section 6.4.5.

The `IMAP4Client.fetch` method performs a fetch:

```
public synchronized String fetch(String in_msgSet,
            String in_fetchCriteria) throws IOException
```

The `IMAP4Client.uidfetch` method performs a fetch using unique identifiers for messages:

```
public synchronized String uidFetch(String in_msgSet,
            String in_fetchCriteria) throws IOException
```

It uses the UID IMAP4 protocol command to specify that the FETCH command uses unique message identifiers rather than sequence numbers.

Both methods generate tags that you can use to help match the command and the response associated with it.

Note For the callback mapping for these methods, see IMAP4 Callback Mapping. §

The following section of code fetches the body of the message specified by the message number.

```
l_client.fetch("1:*", "(BODY[HEADER])");

l_client.processResponses();
```

[Top]

Closing a Mailbox

To close a mailbox, use the `IMAP4Client.close` method. This method sends the CLOSE IMAP4 protocol command, which closes the mailbox and removes any messages marked with the `\Deleted` flag. You can close the mailbox without logging out. In this case, the session moves to the parent mailbox. Use this syntax:

```
public synchronized String close() throws IOException
```

If you need to permanently delete messages without closing, call the `IMAP4Client.expunge` method.

This method generate a tag that you can use to help match the command and the response associated with it.

Note For this method's callback mapping, see IMAP4 Callback Mapping. §

The following section of code closes a mailbox.

```
l_client.close();
l_client.processResponses();
```

[Top]

Chapter

5

Receiving Mail with POP3

This chapter is an overview of using POP3 (Post Office Protocol 3) to download messages to a client.

- The POP3 Protocol
- POP3 Callback Mapping
- Creating a Response Sink
- Creating a Client
- Connecting to a Server
- Logging In
- Getting Message Count
- Listing Messages
- Retrieving Message Headers
- Retrieving a Message
- Ending the Session

[Top]

The POP3 Protocol

POP3 (Post Office Protocol 3) retrieves mail from mailboxes on a remote server. The server retains messages until the client requests them.

Unlike IMAP4, POP3 only receives mail. IMAP4 provides the capabilities of POP3 along with the ability to move messages back and forth between client and server, and manage mailboxes on the server. For information about IMAP4, see Chapter 4, "Receiving Mail with IMAP4."

POP3 commands use the ASCII character set. They are made up of a keyword, followed by any parameters the command has, and ending with "<CRLF>.<CRLF>." Commands are line-oriented and can return a single or multi-line response.

For detailed information about POP3, see RFC 1939: "Post Office Protocol - Version 3."

[Top]

POP3 Session States

A POP3 session progresses through three stages, or states. Within each state, only certain commands are possible.

Figure 5.1 POP3 Session States

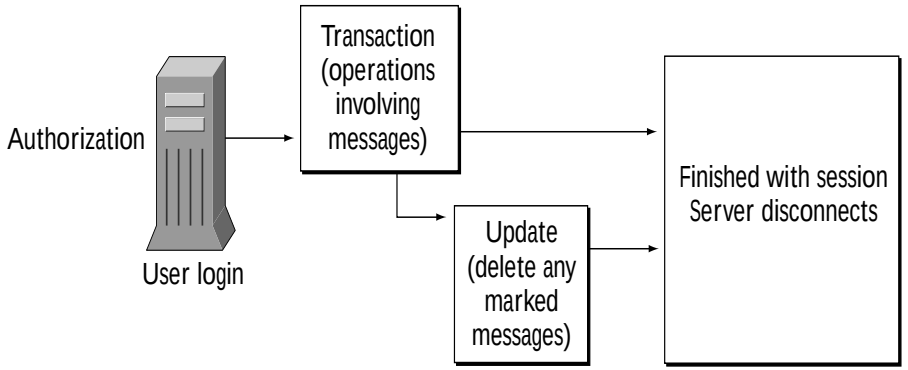


Table 5.1 POP3 Session States and Commands

Session State	Commands
Authorization	User login, password approval, quit. Commands: USER, PASS, QUIT
Transaction	Operations involving messages. Commands: STAT, LIST, RETR, DELE, NOOP, RSET, TOP, UIDL
Update	Delete any marked messages. The session enters this state after a QUIT command, deletes messages marked for deletion, then quits. Commands: QUIT

For a table of SDK-supported POP3 protocol commands that lists the state in which each can be called, see Supported POP3 Internet Protocol Commands. For detailed information about POP3 and POP3 session states, see POP3 RFCs.

[Top]

POP3 Response Codes

When a client sends a POP3 command, a text response code is returned with descriptive text. The response can either be single- or multi-line.

Table 5.2 POP3 Response Types

Response Type	Response Code and Description
Single line	• +OK (Success), followed by descriptive text, for example, "+OK message deleted" for the delete operation. Responses are mapped to appropriate callbacks. • -ERR (failure), followed by descriptive text, for example, "-ERR no such message" for the delete operation. Mapped to error callback.
Multi-line	**First line:** Like single-line response: • +OK (Success), followed by descriptive text, for example, "+OK message follows" for the retrieve operation. Responses are mapped to appropriate callbacks. • -ERR (failure), followed by descriptive text, for example, "-ERR no such message" for the retrieve operation. Mapped to error callback. **Subsequent lines:** More information about the condition. **Final line**: . (dot) and <CRLF>. (Not considered part of the response.) **Note:** If an error occurs on a multi-line response, a single line is returned.

For a list of functions and their associated callbacks, see POP3 Callback Mapping.

[Top]

Steps in a POP3 Session

Generally, a messaging application follows these steps when using POP3 to receive mail. These steps are listed below with links to more detailed descriptions.

Step	Section with details
Initialize the response sink.	Creating a Response Sink
Create a client.	Creating a Client

Step	Section with details
Connect to the server.	Connecting to a Server
Log in to the server.	Logging In
Get the message count.	Getting Message Count
List messages on the server.	Listing Messages
Retrieve the message headers.	Retrieving Message Headers
Retrieve messages themselves.	Retrieving a Message
End the POP3 session.	Ending the Session

[Top]

POP3 in the Messaging Access SDK

The POP3 class hierarchy is made up of the following classes.

- `netscape.messaging.pop3.IPOP3Sink`. Interface for the POP3 response sink. See Creating a Response Sink.
- `netscape.messaging.pop3.POP3Client`. Represents the POP3 client. See Creating a Client.
- `netscape.messaging.pop3.POP3Sink`. Implementation of the `IPOP3Sink` interface. See Creating a Response Sink.
- `netscape.messaging.pop3.POP3Exception`. Exception thrown when an POP3 API error condition is detected by the Messaging Access SDK. Extends `IOException`.
- `netscape.messaging.pop3.POP3ServerException`. Exception thrown from the `error` callback on the response sink when the server sends an error. Extends `IOException` and `POP3Exception`.

[Top]

POP3 Callback Mapping

Callbacks are associated with many POP3 methods. For general information about the response sink and callbacks, see SDK Sink Classes for Java.

The `IPOP3Sink` interface contains callbacks for each client call. The client's `processResponses` method invokes the interface method that corresponds to the client call. Methods with multi-line responses map to more than one callback. The first type of callback indicates the start of a notification. The second type passes back multi-line data. The third type of callback provides a notification that the operation is complete.

If a server error occurs, the error callback is invoked.

Table 5.3 shows which POP3 methods are mapped to callbacks in the `IPOP3Sink` interface. Table 5.4 shows methods that do not map to callbacks.

Table 5.3 Methods with Callbacks

`POP3Client` Methods	Possible Callbacks on `IPOP3Sink`
`connect`	`connect, error`
`delete`	`dele, error`
`list`	`listStart, list, listComplete, error`
`noop`	`noop, error`
`pass`	`pass, error`
`quit`	`quit, error`
`reset`	`reset, error`
`retrieve`	`retrieveStart, retrieve, retrieveComplete, error`
`sendCommand`	`sendCommandStart, sendCommand, sendCommandComplete, error`
`stat`	`stat, error`
`top`	`topStart, top, topComplete, error`
`uidList`	`uidListStart, uidList, uidListComplete, error`

POP3Client Methods	Possible Callbacks on IPOP3Sink
user	user, error
xAuthList	xAuthListStart, xAuthList, xAuthListComplete, error
xSender	xSender, error

Table 5.4 Methods without Callbacks

Methods Without Callbacks
disconnect
processResponses
setChunkSize
setResponseSink
setTimeout

[Top] [POP3 Callback Mapping]

Creating a Response Sink

The first step in starting a POP3 session is to create the POP3 response sink, which is defined in the `IPOP3Sink` interface. The response sink contains the callback methods for the POP3 client. For general information about the response sink, see SDK Sink Classes for Java.

The `IPOP3Sink` interface contains callbacks for each client call. You must implement this interface in order to use the POP3 client object. The constructor for the `POP3Client` class takes an `IPOP3Sink` object as a parameter.

To implement your own response sink class, you can extend the `IPOP3Sink` interface, using this syntax:

```
public class newIPOP3Sink extends Object
   implements IPOP3Sink
```

As a convenience, the Messaging SDK provides the `POP3Sink` class, which implements the `IPOP3Sink` interface. `POP3Sink` implements all the interfaces in `IPOP3Sink`. By default, the implementation does nothing, except for the `error` callback, which throws an exception. You can save a step by extending this class, using this syntax:

```
public class newPOP3Sink extends IPOP3Sink{
   }
```

The following section of code creates a response sink.

```
POP3Sink l_pop3Sink;

l_pop3Sink = new POP3Sink();
```

After you create the response sink, the next step is Creating a Client.

[Top]

Creating a Client

The POP3 client uses a `POP3Client` object to communicate with the server. To create the `POP3Client` object and set the response sink for the client's use, call the `POP3Client.POP3Client` class constructor, which takes an existing response sink. Use this syntax:

```
public POP3Client(IPOP3Sink in_sink)
```

The client class must implement all of the methods in the `IPOP3Sink` interface.

The following section of code creates a client.

```
/* Create sink first as described in Creating a Response Sink */

POP3Client l_client;

l_client = new POP3Client( l_pop3Sink );
```

After you initialize the client, the next step is Connecting to a Server.

[Top]

Connecting to a Server

Before receiving mail, the client must connect with the server through a service port. To connect to the server, call either of two `POP3Client connect` methods, depending on whether or not you want to specify the connection port. The SDK methods perform some error-checking.

To connect to the server using the default port for the POP3 protocol (port 110), use this form of `connect` and supply the identifier of the server:

```
public synchronized void connect(String in_server)
                                  throws IOException
```

To specify the server port to use for the server connection, use this form of `connect`:

```
public synchronized void connect(String in_server,
                     int in_port) throws IOException
```

Note For the callback mapping for these methods, see POP3 Callback Mapping. §

The following section of code connects the client to the server.

```
/* After Creating a Response Sink and Creating a Client */

l_client.connect( "pop3server.com" );

l_client.processResponses();
```

After connecting to the server, the next step is Logging In.

To disconnect the client from the server and close the socket connection, use this `POP3Client` class method:

```
public synchronized void disconnect() throws IOException
```

You could use this method as part of a Cancel operation while retrieving a message. Remember that you do not call `processResponses` after `disconnect`.

[Top]

Logging In

Once the client is connected to the server, the user can log in. Login identifies the client to the server. Logging in requires the identifier of the POP3 client, as well as the user's ID and plain text password.

First, call `POP3Client.user` and give the name of the user's maildrop or mailbox:

```
public synchronized void user(String in_user) throws IOException
```

This method sends the USER POP3 protocol command.

To submit the user password, call the `POP3Client.pass` method:

```
public synchronized void pass(String in_password)
                              throws IOException
```

This method sends the PASS POP3 protocol command.

While these commands execute, the session is in Authorization state. Successful completion moves the session to the Transaction state. For a list of states and commands that can be executed in each state, see POP3 Session States.

A successful login moves the POP3 session from the Authorization state to the Transaction state, where the user can perform a number of operations.

Note For the callback mapping for these methods, see POP3 Callback Mapping. §

The following section of code shows a login sequence for a user.

```
/* User id sequence */
l_client.user( "pop3user" );
l_client.processResponses();
/* Password sequence */
l_client.pass( "pop3password" );
l_client.processResponses();
```

After you log in, you can perform any Transaction state operations. The next step is Retrieving a Message.

[Top]

Getting Message Count

In the POP3 Transaction state, the client can find out how many messages are present by requesting the status of the mail drop or mailbox. The `POP3Client.status` method gets the mailbox status for a given client by issuing the STAT protocol command.

The `POP3Client.status` method gets mailbox status for a given mailbox:

```
public synchronized String status(String in_mailbox,
                     String in_statusData) throws IOException
```

The status returned includes the number of messages present and the octet size of the mail drop.

Note For this method's callback mapping, see POP3 Callback Mapping. §

The following section of code requests the status of the mail drop.

```
l_client.stat();

l_client.processResponses();
```

[Top]

Listing Messages

To list messages while in the POP3 Transaction state, call either of two `POP3Client.list` methods, depending on whether you want to list all of the messages in a mailbox or a specific message.

To go through all the messages in the mailbox and generate a list of messages, use the `list` method that takes no parameters:

```
public synchronized void list() throws IOException
```

This method sends the LIST POP3 protocol command.

To list only the message specified by the message number, use this `list` method:

```
public synchronized void list(int in_messageNumber)
                              throws IOException
```

Supply the message number as a parameter. This method sends the LIST [arg] POP3 protocol command.

Note For the callback mapping for these methods, see POP3 Callback Mapping. §

The following section of code retrieves the email address of the sender along with authenticated messages.

```
l_client.list();

l_client.processResponses();
```

[Top]

Retrieving Message Headers

In the Transaction state, the user can preview mailbox contents or part of a long message before deciding to download it by listing the headers plus some of the lines of the body. The user (or the mail application) determines the number of message lines to retrieve.

To identify the message, supply the number of the message and the number of body lines to retrieve. Use the `POP3Client.top` method, which issues the TOP protocol command:

```
public synchronized void top(int in_messageNumber,
                             int in_lines) throws IOException
```

To retrieve all headers for a given mailbox, combine `POP3Client.top` with a call to `POP3Client.stat`. First, call `stat` to find the number of messages in the mailbox. When you get the number, for example, 10, call `top` once, passing each message number in turn, until you get to the total (in this case, 10). For the `in_lines` parameter, use a value of 0 so that no body lines are returned.

Note For this method's callback mapping, see POP3 Callback Mapping. §

The following section of code lists the header of the message specified by its message number. It retrieves no body lines.

```
l_client.top( 1, 0 );

l_client.processResponses();
```

After you retrieve message headers, you can go on to Retrieving a Message if you like.

[Top]

Retrieving a Message

Retrieving a message is one of the most common activities that users want to perform during the POP3 Transaction state.

The `POP3Client.pop3_retrieve` method takes the identifier of the POP3 client that is retrieving the mail and the message number, and retrieves the contents of the message:

```
public synchronized void retrieve(int in_messageNumber)
                                  throws IOException
```

The message is returned in the form of data chunks, which are sent to the application through callbacks. The `pop3_retrieve` method issues a RETR command. It fails if the message with the specified number does not exist.

Note For this method's callback mapping, see POP3 Callback Mapping. §

The following section of code retrieves the contents of a message.

```
l_client.retrieve( 1 );

l_client.processResponses();
```

[Top]

Ending the Session

When it is time to end the session, the client should call `POP3Client.quit` to notify the server:

```
public synchronized void quit() throws IOException
```

This method sends the QUIT POP3 protocol command. The server closes the TCP connection and sends back a response. It is preferable to end a session with `quit` instead of just closing the connection.

If the session is in the Authentication state when this method is called, the server simply closes the connection. If the session is in the Transaction state, the server goes into the Update state and expunges any messages marked for deletion, and then quits.

Note For this method's callback mapping, see POP3 Callback Mapping. §

The following section of code notifies the server that the client is terminating the session.

```
l_client.quit();

l_client.processResponses();
```

[Top]

2

Messaging Access SDK Java

Reference

This page contains links to the reference, in JavaDocs format, to the Java versions of the Messaging Access SDK Guide Protocol APIs.

SMTP Class Hierarchy

MIME Class Hierarchy

IMAP4 Class Hierarchy

POP3 Class Hierarchy

Java Convenience API

Chapter 6 Reference to Protocols

This chapter summarizes essential information about the Internet Protocols accessed through the Messaging Access SDK.

[Top]

Chapter

6

Reference to Protocols

This chapter summarizes essential information about the Internet Protocols accessed through the Messaging Access SDK.

Messaging Access SDK Protocol APIs are based on the standard Internet messaging protocols, SMTP, IMAP4, POP3, and MIME. The SDK implementations of the protocols contain methods that call Internet Protocol commands. This chapter lists the Internet Protocol commands supported by the Messaging Access SDK, defines them, and notes the SDK methods that call them.

- Supported SMTP Internet Protocol Commands. SMTP (Simple Mail Transfer Protocol) sends messages.
- Supported IMAP4 Internet Protocol Commands. IMAP4 (Internet Message Access Protocol) retrieves and manages messages remotely.
- Supported POP3 Internet Protocol Commands. POP3 (Post Office Protocol) downloads messages to a client and allows for search and retrieval of messages.

- MIME, Multipurpose Internet Mail Extension. MIME builds code messages with attachments for sending with SMTP, and parses and decodes received messages. SDK MIME methods and functions do not map to a set of Internet Protocol commands in the same way that the other protocols do.

[Top]

Supported SMTP Internet Protocol Commands

This table lists supported protocol commands for SMTP (Simple Mail Transfer Protocol) and the SDK `SMTPClient` method that calls each command. For the RFC sources for these protocol APIs, see SMTP RFCs.

Supported Internet Protocol command	What the command does	`SMTPClient` method that calls the command
`DATA`	Informs server that the client is about to send the message. After server OK, client sends RFC 822-compliant message data line by line. On completion, client sends "`<CRLF>.<CRLF>`" line.	`data`
`DSN [NOTIFY, RET, ENVID]`	Allows SMTP client to generate delivery status notifications (DSNs) when needed, determine whether the notifications return the message contents, and get additional information with a DSN so that the sender can identify both recipient(s) for the DSN and the transaction that contained the original message. (RFC 1891)	
`EHLO`	Client starts SMTP session by sending identification to server; server responds (identifies itself) in a greeting message. EHLO replaces older HELO command for SMTP clients that support SMTP service extensions. Can be issued at the start of the session to see which extensions the server supports.	`ehlo`
`EXPN`	Expands a mailing list alias; the command retrieves the alias member list.	`expand`

Supported Internet Protocol command	What the command does	SMTPClient method that calls the command
HELP	Calls the server Help utility. Application-specific; usually lists available commands.	help
MAIL FROM	Initiates sending the message; supplies the message's reverse path (usually the sender's fully qualified domain name).	mailFrom
NOOP	Gets positive server response.	noop
QUIT	Sent by a client to a server when the client is ready to end the session. Server sends response and closes TCP connection. Best to use this command rather than just closing the connection.	quit
PIPELINING	Allows command pipelining (batching multiple commands into single TCP sends). To find out if server supports pipelining, issue the EHLO command. If it does, the server response includes code 250 and EHLO keyword PIPELINING. Pipelining allows the client to transmit batches of SMTP commands without waiting for a response to each. (RFC 2197)	
RCPT TO	Specifies the address of a message recipient. Called once for each recipient. Follows the MAIL command.	rcptTo
RSET	Cancels the current mail transfer and all current processes, discards data, and clears session states. Returns to the session state that followed the EHLO command.	reset
VRFY	Sent by a client to verify a user name with the server. The server responds with a positive or negative code.	verify

[Top]

Supported IMAP4 Internet Protocol Commands

This table lists supported protocol commands for IMAP4 (Internet Message Access Protocol) and the SDK `IMAP4Client` method that calls each command. For the RFC sources for these protocol APIs, see IMAP4 RFCs.

Supported Internet Protocol command	What the command does	Session state for command	`IMAP4Client` method that calls the command
`APPEND`	Appends message to specified mailbox, passes on any message flags.	Authenticated	`append`
`CAPABILITY`	Gets a list of server capabilities.	All states	`capability`
`CHECK`	Requests a checkpoint of the currently selected mailbox; server flushes existing mailbox states to disk.	Selected	`check`
`CLOSE`	Closes a mailbox, deletes flagged messages, moves session to Non-Authenticated state.	Selected	`close`
`COPY`	Copies a message to the specified mailbox.	Selected	`copy`
`CREATE`	Creates a mailbox.	Authenticated	`create`
`DELETE`	Marks a message for deletion.	Authenticated	`delete`
`EXAMINE`	Like `SELECT`, but read-only.	Authenticated	`examine`
`EXPUNGE`	Removes all messages flagged "`\Deleted`" in a mailbox.	Selected	`expunge`
`FETCH`	Returns information from messages. The protocol's `FETCH [XSENDER]` form is supported.	Selected	`fetch`
`LIST`	Gets list of user names.	Authenticated	`list`
`LOGIN`	Logs in to server with user name and password.	Non-Authenticated	`login`
`LOGOUT`	Ends session; server responds with "BYE."	All states	`logout`
`LSUB`	Lists members of subscription list (must be added with SUBSCRIBE).	Authenticated	`lsub`

Supported Internet Protocol command	What the command does	Session state for command	IMAP4Client method that calls the command
NAMESPACE	Retrieves the prefixes of namespaces used by a server for personal mailboxes, other user's mailboxes, and shared mailboxes.	All states	namespace
NOOP	Gets positive server response.	All states	noop
RENAME	Renames mailbox.	Authenticated	rename
SEARCH	Finds messages that meet specified criteria.	Selected	search
SELECT	Selects a mailbox on the server for operations involving messages.	Authenticated	select
STATUS	Requests one or more types of status for the specified mailbox.	Authenticated	status
STORE	Updates flags on messages; can return the new flag status.	Selected	store
SUBSCRIBE	Adds a mailbox to a server's subscribed mailbox list. List is accessed with LSUB.	Authenticated	subscribe
UID	Used with a command name to specify that it uses unique message identifiers.	Selected	uidCopy uidFetch uidSearch uidStore
UNSUBSCRIBE	Deletes a mailbox from a server's subscribed mailbox list. List is accessed with LSUB.	Authenticated	unsubscribe
SETACL	Changes the access control list on the specified mailbox and grants specified permissions.	All states	setACL
DELETEACL	Removes an <identifier, rights> pair for the specified identifier from the access control list for the specified mailbox.	All states	deleteACL
GETACL	Retrieves the access control list for the mailbox in an untagged ACL reply.	All states	getACL
LISTRIGHTS	Retrieves the access control list for mailbox in an untagged ACL reply.	All states	listRights
MYRIGHTS	Retrieves the user's rights to the specified mailbox.	All states	myRights

[Top]

Supported POP3 Internet Protocol Commands

This table lists supported protocol commands for POP3 (Post Office Protocol) and the SDK `POP3Client` method that calls each command. For the RFC sources for these protocol APIs, see POP3 RFCs.

Supported Internet Protocol command	What the command does	Session state for command	`POP3Client` method that calls the command
`DELE`	Asks server to mark specified message for deletion. Deletion actually takes place on entry into Update state.	Transaction	`delete`
`LIST`	Gets the size of one or all messages.	Transaction	`list` `listA`
`NOOP`	Gets positive server response.	Transaction	`noop`
`PASS`	Identifies a user password; on success, moves session to the Transaction state.	Authorization	`pass`
`QUIT`	Ends the session. If issued in Authentication state, server closes connections. If issued in Transaction state, server goes into Update and deletes any marked messages, then quits.	Authorization, Update	`quit`
`RSET`	Asks the server to clear all delete tags from messages.	Transaction	`reset`
`RETR`	Requests the entire specified message.	Transaction	`retrieve`
`STAT`	Gets the number of messages in and octet size of mail drop.	Transaction	`stat`
`TOP`	Asks server for first *n* lines of specified message.	Transaction	`top`

Supported Internet Protocol command	What the command does	Session state for command	POP3Client method that calls the command
UIDL	Gets the unique identifier string for specified or all messages.	Transaction	uidList uidListA
USER	Identifies the user or mail drop by name to the server; the server returns a known or unknown response.	Authorization	user
XAUTHLIST	Returns a list of authenticated users.	Transaction	xAuthList xAuthListA
XSENDER	Gets the email address of the sender of the specified message. Client uses this to query whether an individual message has been authenticated. Server returns an empty OK string if no authenticated sender is found.	Transaction	xSender

[Top]

Appendix

A

Writing Multithreaded Applications with the Messaging Access SDK

This appendix provides some important information for developers who want to take advantage of multithreading in their messaging applications.

The Java Messaging Access SDK is thread-safe. Three types of Messaging Access SDK methods can share resources. The methods of each type are synchronized with each other and with one of the other types, as shown in this table.

Type	Description	Synchronized with
1	Protocol API commands, for example, `SMTPClient.rcptTo`, `POP3Client.list`.	Synchronized among each other and with type 3.
2	The `processResponses` methods.	Synchronized among each other and with type 3.
3	Methods that set options, for example, `SMTPClient.setTimeout`, `POP3Client.setTimeout`.	Synchronized among each other and with type 2.

Using multiple threads enhances performance by allowing the client to send commands to the server for processing before completing a call to `processResponses`. For example, a thread that uses the IMAP4 module can invoke a call to `processResponses` to download a message while another thread is executing API commands.

A multithreaded application may need to provide a lock on the SDK in order to synchronize state-sensitive sequences of commands within the SMTP, POP3, and IMAP4 modules as well. For example, if two threads that are connected to the same SMTP server are used to send mail, a lock is needed to prevent both threads from sending mail at the same time. If the `SMTPClient.rcptTo` methods of the two threads are interleaved, the email may be sent to the wrong destination.

[Top]

Multithreading in the MIME API

In general, in the MIME API, concurrent access by multiple threads is not necessary. Multiple threads should not need to change the same `MIMEMessage` or other MIME objects, such as `MIMEMultiPart`, concurrently.

One situation that is applicable to MIME in a multithreaded environment is the simultaneous parsing of multiple messages by different threads. To do this, multiple threads can create their own instances of the `MIMEParser` (or `MIMEDynamicParser`) object or share the same `MIMEParser` (or `MIMEDynamicParser`) instance.

When multiple threads share the parser object, the client application must serialize access to the object. Different `MIMEDynamicParser` objects can share a single instance of the `MIMEDataSink` object, however.

[Top]

Index

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

Messaging Access SDK Guide