SUN IS WILLING TO LICENSE THIS SPECIFICATION TO YOU ONLY UPON THE CONDITION THAT YOU ACCEPT ALL OF THE TERMS CONTAINED IN THIS LICENSE AGREEMENT ("AGREEMENT"). PLEASE READ THE TERMS AND CONDITIONS OF THIS LICENSE CAREFULLY. BY DOWNLOADING THIS SPECIFICATION, YOU ACCEPT THE TERMS AND CONDITIONS OF THIS LICENSE AGREEMENT. IF YOU ARE NOT WILLING TO BE BOUND BY ITS TERMS, SELECT THE "DECLINE" BUTTON AT THE BOTTOM OF THIS PAGE AND THE DOWNLOADING PROCESS WILL NOT CONTINUE.

JavaMail (TM) Specification ("Specification")
Version: 1.2
Status: FCS
Release: December 05, 2000
Copyright 2000 Sun Microsystems, Inc.
901 San Antonio Road, Palo Alto, California 94303, U.S.A.
All rights reserved.

NOTICE
The Specification is protected by copyright and the information described therein may be protected by one or more U.S. patents, foreign patents, or pending applications. Except as provided under the following license, no part of the Specification may be reproduced in any form by any means without the prior written authorization of Sun Microsystems, Inc. ("Sun") and its licensors, if any. Any use of the Specification and the information described therein will be governed by the terms and conditions of this license and the Export Control and General Terms as set forth in Sun’s website Legal Terms. By viewing, downloading or otherwise copying the Specification, you agree that you have read, understood, and will comply with all of the terms and conditions set forth herein.

Sun hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under Sun’s intellectual property rights that are essential to practice the Specification, to internally practice the Specification solely for the purpose of creating a clean room implementation of the Specification that: (i) includes a complete implementation of the current version of the Specification, without subsetting or supersetting; (ii) implements all of the interfaces and functionality of the Specification, as defined by Sun, without subsetting or supersetting; (iii) includes a complete implementation of any optional components (as defined by Sun in the Specification) which you choose to implement, without subsetting or supersetting; (iv) implements all of the interfaces and functionality of such optional components, without subsetting or supersetting; (v) does not add any additional packages, classes or interfaces to the "java." or "javax." packages or subpackages (or other packages defined by Sun); (vi) satisfies all testing requirements available from Sun relating to the most recently published version of the Specification six (6) months prior to any release of the clean room implementation or upgrade thereto; (vii) does not derive from any Sun source code or binary code materials; and (viii) does not include any Sun source code or binary code materials without an appropriate and separate license from Sun. The Specification contains the proprietary information of Sun and may only be used in accordance with the license terms set forth herein. This license will terminate immediately without notice from Sun if you fail to comply with any provision of this license. Upon termination or expiration of this license, you must cease use of or destroy the Specification.

TRADEMARKS
No right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun’s licensors is granted hereunder. Sun, Sun Microsystems, the Sun logo, Java, JavaMail, and the Java Coffee Cup logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

DISCLAIMER OF WARRANTIES
THE SPECIFICATION IS PROVIDED "AS IS". SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. SUN MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

LIMITATION OF LIABILITY
TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. You will indemnify, hold harmless, and defend Sun and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your Java application, applet and/or clean room implementation; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND
U.S. Government: If this Specification is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government’s rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7201-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.210 and 12.212 (for non-DoD acquisitions).

REPORT
You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your use of the Specification ("Feedback"). To the extent that you provide Sun with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

(Form last modified 11-03-2000.)

September 2000
## Contents

### Chapter 1: Introduction  1
- Target Audience  1
- Acknowledgments  1

### Chapter 2: Goals and Design Principles  3

### Chapter 3: Architectural Overview  5
- JavaMail Layered Architecture  5
- JavaMail Class Hierarchy  7
- The JavaMail Framework  8
- Major JavaMail API Components  10
  - The `Message` Class  10
  - Message Storage and Retrieval  10
  - Message Composition and Transport  11
- The `Session` Class  11
- The JavaMail Event Model  11
- Using the JavaMail API  12

### Chapter 4: The Message Class  13
- The Part Interface  16
  - Message Attributes  16
  - The `ContentType` Attribute  17
- The Address Class  18
- The BodyPart Class  18
- The Multipart Class  19
- The Flags Class  22
- Message Creation And Transmission  23

### Chapter 5: The Mail Session  25
- The Provider Registry  26
  - Resource Files  26
- Provider  28
- Protocol Selection and Defaults  28
The MimeUtility Class | 58
Content Encoding and Decoding | 59
Header Encoding and Decoding | 59
The ContentType Class | 60

Appendix A: Environment Properties | 61

Appendix B: Examples Using the JavaMail API | 63
Example: Showing a Message | 63
Example: Listing Folders | 69
Example: Search a Folder for a Message | 72
Example: Monitoring a Mailbox | 77
Example: Sending a Message | 78

Appendix C: Message Security | 81
Overview | 81
Displaying an Encrypted/Signed Message | 81
MultiPartEncrypted/Signed Classes | 81
Reading the Contents | 82
Verifying Signatures | 82
Creating a Message | 83

Appendix D: Part and Multipart Class Diagram | 85

Appendix E: MimeMessage Object Hierarchy | 87

Appendix F: Features Added in JavaMail 1.1 | 89
The MessageContext Class and MessageAware Interface | 89
The getMessageID method | 89
Additions to the InternetAddress Class | 90
Additions to the MimeUtility Class | 90
New SearchTerms | 90
Additions to the Folder Class | 91
New Service Class | 91

Appendix G: Features Added in JavaMail 1.2 | 93
Additions to the MimeMessage Class | 93
Additions to the MimeMultipart Class | 94
The getRawInputStream method | 94
Additions to the InternetAddress Class | 94
The MailDateFormat Class | 95
Additions to Exceptions and Events 95
Additions to the Session Class 96
Additions to the MimeUtility Class 96
Additions for serializable javax.mail.search terms 96
Additions to the Store Class 97
New ContentDisposition Class 97
New performance improvements 98
Additions to the ParameterList class 98
Chapter 1: Introduction

In the few years since its first release, the Java™ programming language has matured to become a platform. The Java platform has added functionality, including distributed computing with RMI and the CORBA bridge, and a component architecture (JavaBeans™). Java applications have also matured, and many now need an addition to the Java platform: a mail and messaging framework. The JavaMail™ API described in this specification satisfies that need.

The JavaMail API provides a set of abstract classes defining objects that comprise a mail system. The API defines classes like Message, Store and Transport. The API can be extended and can be subclassed to provide new protocols and to add functionality when necessary.

In addition, the API provides concrete subclasses of the abstract classes. These subclasses, including MimeMessage and MimeBodyPart, implement widely used Internet mail protocols and conform to specifications RFC822 and RFC2045. They are ready to be used in application development.

Target Audience

The JavaMail API is designed to serve several audiences:

- Client, server, or middleware developers interested in building mail and messaging applications using the Java programming language.
- Application developers who need to “mail-enable” their applications.
- Service Providers who need to implement specific access and transfer protocols. For example; a telecommunications company can use the JavaMail API to implement a PAGER Transport protocol that sends mail messages to alphanumeric pagers.

Acknowledgments

The authors of this specification are John Mani, Bill Shannon, Max Spivak, Kapono Carter and Chris Cotton.

We would like to acknowledge the following people for their comments and feedback on the initial drafts of this document:
Acknowledgments

Terry Cline, John Russo, Bill Yeager and Monica Gaines: Sun Microsystems.

Arn Perkins and John Ragan: Novell, Inc.

Nick Shelness: Lotus Development Corporation.

Juerg von Kaenel: IBM Corporation.

Prasad Yendluri, Jamie Zawinski, Terry Weissman and Gena Cunanan: Netscape Communications Corporation.
Chapter 2: Goals and Design Principles

The JavaMail API is designed to make adding electronic mail capability to simple applications easy, while also supporting the creation of sophisticated user interfaces. It includes appropriate convenience classes which encapsulate common mail functions and protocols. It fits with other packages for the Java platform in order to facilitate its use with other Java APIs, and it uses familiar programming models.

The JavaMail API is therefore designed to satisfy the following development and runtime requirements:

- Simple, straightforward class design is easy for a developer to learn and implement.
- Use of familiar concepts and programming models support code development that interfaces well with other Java APIs.
  - Uses familiar exception-handling and JDK 1.1 event-handling programming models.
  - Uses features from the JavaBeans Activation Framework (JAF) to handle access to data based on data-type and to facilitate the addition of data types and commands on those data types. The JavaMail API provides convenience functions to simplify these coding tasks.
- Lightweight classes and interfaces make it easy to add basic mail-handling tasks to any application.
- Supports the development of robust mail-enabled applications, that can handle a variety of complex mail message formats, data types, and access and transport protocols.

The JavaMail API draws heavily from IMAP, MAPI, CMC, c-client and other messaging system APIs: many of the concepts present in these other systems are also present in the JavaMail API. It is simpler to use because it uses features of the Java programming language not available to these other APIs, and because it uses the Java programming language’s object model to shelter applications from implementation complexity.

The JavaMail API design is driven by the needs of the applications it supports—but it is also important to consider the needs of API implementors. It is critically important to enable the implementation of messaging systems written using the Java programming language that interoperate with existing messaging systems—especially
Internet mail. It is also important to anticipate the development of new messaging systems. The JavaMail API conforms to current standards while not being so constrained by current standards that it stifles future innovation.

The JavaMail API supports many different messaging system implementations—different message stores, different message formats, and different message transports. The JavaMail API provides a set of base classes and interfaces that define the API for client applications. Many simple applications will only need to interact with the messaging system through these base classes and interfaces.

JavaMail subclasses can expose additional messaging system features. For instance, the MimeMessage subclass exposes and implements common characteristics of an Internet mail message, as defined by RFC822 and MIME standards. Developers can subclass JavaMail classes to provide the implementations of particular messaging systems, such as IMAP4, POP3, and SMTP.

The base JavaMail classes include many convenience APIs that simplify use of the API, but don’t add any functionality. The implementation subclasses are not required to implement those convenience methods. The implementation subclasses must implement only the core classes and methods that provide functionality required for the implementation.

Alternately, a messaging system can choose to implement all of the JavaMail API directly, allowing it to take advantage of performance optimizations, perhaps through use of batched protocol requests. The IMAP4 protocol implementation takes advantage of this approach.

The JavaMail API uses the Java programming language to good effect to strike a balance between simplicity and sophistication. Simple tasks are easy, and sophisticated functionality is possible.
Chapter 3: Architectural Overview

This section describes the JavaMail architecture, defines major classes and interfaces comprising that architecture, and lists major functions that the architecture implements.

JavaMail provides elements that are used to construct an interface to a messaging system, including system components and interfaces. While this specification does not define any specific implementation, JavaMail does include several classes that implement RFC822 and MIME Internet messaging standards. These classes are delivered as part of the JavaMail class package.

JavaMail Layered Architecture

The JavaMail architectural components are layered as shown below:

- The Abstract Layer declares classes, interfaces and abstract methods intended to support mail handling functions that all mail systems support. API elements comprising the Abstract Layer are intended to be subclassed and extended as necessary in order to support standard data types, and to interface with message access and message transport protocols as necessary.

- The internet implementation layer implements part of the abstract layer using internet standards - RFC822 and MIME.

- JavaMail uses the JavaBeans Activation Framework (JAF) in order to encapsulate message data, and to handle commands intended to interact with that data. Interaction with message data should take place via JAF-aware JavaBeans, which are not provided by the JavaMail API.
JavaMail clients use the JavaMail API and Service Providers implement the JavaMail API. The layered design architecture allows clients to use the same JavaMail API calls to send, receive and store a variety of messages using different data-types from different message stores and using different message transport protocols.

FIGURE 3-1
JavaMail Class Hierarchy

The figure below shows major classes and interfaces comprising the JavaMail API. See “Major JavaMail API Components” on page 10 for brief descriptions of all components shown on this diagram.
The JavaMail Framework

The JavaMail API is intended to perform the following functions, which comprise the standard mail handling process for a typical client application:

- Create a mail message consisting of a collection of header attributes and a block of data of some known data type as specified in the Content-Type header field. JavaMail uses the Part interface and the Message class to define a mail message. It uses the JAF-defined DataHandler object to contain data placed in the message.
- Create a Session object, which authenticates the user, and controls access to the message store and transport.
- Send the message to its recipient list.
- Retrieve a message from a message store.
- Execute a high-level command on a retrieved message. High-level commands like view and print are intended to be implemented via JAF-Aware JavaBeans.

Note – The JavaMail framework does not define mechanisms that support message delivery, security, disconnected operation, directory services or filter functionality. Security, disconnected operation and filtering support will be added in future releases.
This figure illustrates the JavaMail message-handling process.

FIGURE 3-3

MESSAGE

Send a Message

TRANSPORT

Submit a Message

Network Infrastructure

Receive a Message

MESSAGE

Contains Messages

FOLDERS

STORE

Contains Messages
Major JavaMail API Components

This section reviews major components comprising the JavaMail architecture.

The Message Class

The Message class is an abstract class that defines a set of attributes and a content for a mail message. Attributes of the Message class specify addressing information and define the structure of the content, including the content type. The content is represented as a DataHandler object that wraps around the actual data.

The Message class implements the Part interface. The Part interface defines attributes that are required to define and format data content carried by a Message object, and to interface successfully to a mail system. The Message class adds From, To, Subject, Reply-To, and other attributes necessary for message routing via a message transport system. When contained in a folder, a Message object has a set of flags associated with it. JavaMail provides Message subclasses that support specific messaging implementations.

The content of a message is a collection of bytes, or a reference to a collection of bytes, encapsulated within a Message object. JavaMail has no knowledge of the data type or format of the message content. A Message object interacts with its content through an intermediate layer—the JavaBeans Activation Framework (JAF). This separation allows a Message object to handle any arbitrary content and to transmit it using any appropriate transmission protocol by using calls to the same API methods. The message recipient usually knows the content data type and format and knows how to handle that content.

The JavaMail API also supports multipart Message objects, where each Bodypart defines its own set of attributes and content.

Message Storage and Retrieval

Messages are stored in Folder objects. A Folder object can contain subfolders as well as messages, thus providing a tree-like folder hierarchy. The Folder class declares methods that fetch, append, copy and delete messages. A Folder object can also send events to components registered as event listeners.

The Store class defines a database that holds a folder hierarchy together with its messages. The Store class also specifies the access protocol that accesses folders and retrieves messages stored in folders. The Store class also provides methods to establish a connection to the database, to fetch folders and to close a connection. Service providers implementing Message Access protocols (IMAP4, POP3 etc.) start off by subclassing the Store class. A user typically starts a session with the mail system by connecting to a particular Store implementation.
Message Composition and Transport

A client creates a new message by instantiating an appropriate Message subclass. It sets attributes like the recipient addresses and the subject, and inserts the content into the Message object. Finally, it sends the Message by invoking the Transport.send method.

The Transport class models the transport agent that routes a message to its destination addresses. This class provides methods that send a message to a list of recipients. Invoking the Transport.send method with a Message object identifies the appropriate transport based on its destination addresses.

The Session Class

The Session class defines global and per-user mail-related properties that define the interface between a mail-enabled client and the network. JavaMail system components use the Session object to set and get specific properties. The Session class also provides a default authenticated session object that desktop applications can share. The Session class is a final concrete class. It cannot be subclassed.

The Session class also acts as a factory for Store and Transport objects that implement specific access and transport protocols. By calling the appropriate factory method on a Session object, the client can obtain Store and Transport objects that support specific protocols.

The JavaMail Event Model

The JavaMail event model conforms to the JDK 1.1 event-model specification, as described in the JavaBeans Specification. The JavaMail API follows the design patterns defined in the JavaBeans Specification for naming events, event methods and event listener registration.

All events are subclassed from the MailEvent class. Clients listen for specific events by registering themselves as listeners for those events. Events notify listeners of state changes as a session progresses. During a session, a JavaMail component generates a specific event-type to notify objects registered as listeners for that event-type. The JavaMail Store, Folder, and Transport classes are event sources. This specification describes each specific event in the section that describes the class that generates that event.
Using the JavaMail API

This section defines the syntax and lists the order in which a client application calls some JavaMail methods in order to access and open a message located in a folder:

1. A JavaMail client typically begins a mail handling task by obtaining the default JavaMail Session object.
   
   ```java
   Session session = Session.getDefaultInstance(props, authenticator);
   ```

2. The client uses the Session object's getStore method to connect to the default store. The getStore method returns a Store object subclass that supports the access protocol defined in the user properties object, which will typically contain per-user preferences.
   
   ```java
   Store store = session.getStore();
   store.connect();
   ```

3. If the connection is successful, the client can list available folders in the Store, and then fetch and view specific Message objects.
   
   ```java
   // get the INBOX folder
   Folder inbox = store.getFolder("INBOX");
   
   // open the INBOX folder
   inbox.open(Folder.READ_WRITE);
   
   Message m = inbox.getMessage(1);       // get Message # 1
   String subject = m.getSubject();       // get Subject
   Object content = m.getContent();       // get content
   ```

4. Finally, the client closes all open folders, and then closes the store.
   
   ```java
   inbox.close();              // Close the INBOX
   store.close();                   // Close the Store
   ```

See “Examples Using the JavaMail API” on page 63 for a more complete example.
Chapter 4: The Message Class

The Message class defines a set of attributes and a content for a mail message. Message attributes specify message addressing information and define the structure of the content, including the content type. The content is represented by a DataHandler object that wraps around the actual data. The Message class is an abstract class that implements the Part interface.

Subclasses of the Message classes can implement several standard message formats. For example, the JavaMail API provides the MimeMessage class, that extends the Message class to implement the RFC822 and MIME standards. Implementations can typically construct themselves from byte streams and generate byte streams for transmission.

A Message subclass instantiates an object that holds message content, together with attributes that specify addresses for the sender and recipients, structural information about the message, and the content type of the message body. Messages placed into a folder also have a set of flags that describe the state of the message within the folder.
The figure below illustrates the structure of the `Message` class.

**FIGURE 4-1**

The `Message` object has no direct knowledge of the nature or semantics of its content. This separation of structure from content allows the message object to contain any arbitrary content.

`Message` objects are either retrieved from a `Folder` object or constructed by instantiating a new `Message` object of the appropriate subclass. Messages stored within a `Folder` object are sequentially numbered, starting at one. An assigned message number can change when the folder is expunged, since the expunge operation removes deleted messages from the folder and also renumbers the remaining messages.
A Message object can contain multiple parts, where each part contains its own set of attributes and content. The content of a multipart message is a Multipart object that contains BodyPart objects representing each individual part. The Part interface defines the structural and semantic similarity between the Message class and the BodyPart class.

The figure below illustrates a Message instance hierarchy, where the message contains attributes, a set of flags, and content. See “MimeMessage Object Hierarchy” on page 81 for an illustration of the MimeMessage object hierarchy.

The Message class provides methods to perform the following tasks:
- Get, set and create its attributes and content:
  
  ```java
  public String getSubject() throws MessagingException;
  
  public void setSubject(String subject) throws MessagingException;
  
  public String[] getHeader(String name) throws MessagingException;
  ```
The Part Interface

The Part interface defines a set of standard headers common to most mail systems, specifies the data-type assigned to data comprising a content block, and defines set and get methods for each of these members. It is the basic data component in the JavaMail API and provides a common interface for both the Message and BodyPart classes. See the JavaMail API (Javadoc) documentation for details.

Note – A Message object can not be contained directly in a Multipart object, but must be embedded in a BodyPart first.

Message Attributes

The Message class adds its own set of standard attributes to those it inherits from the Part interface. These attributes include the sender and recipient addresses, the subject, flags, and sent and received dates. The Message class also supports non-standard attributes in the form of headers. See the JavaMail API (Javadoc) Documentation for the list of standard attributes defined in the Message class. Not all messaging systems will support arbitrary headers, and the availability and meaning of particular header names is specific to the messaging system implementation.
**The ContentType Attribute**

The `contentType` attribute specifies the data type of the content, following the MIME typing specification (RFC 2045). A MIME type is composed of a primary type that declares the general type of the content, and a subtype that specifies a specific format for the content. A MIME type also includes an optional set of type-specific parameters.

JavaMail API components can access content via these mechanisms:

<table>
<thead>
<tr>
<th>As an input stream</th>
<th>The <code>Part</code> interface declares the <code>getInputStream</code> method that returns an input stream to the content. Note that <code>Part</code> implementations must decode any mail-specific transfer encoding before providing the input stream.</th>
</tr>
</thead>
<tbody>
<tr>
<td>As a DataHandler object</td>
<td>The <code>Part</code> interface declares the <code>getDataHandler</code> method that returns a <code>javax.activation.DataHandler</code> object that wraps around the content. The <code>DataHandler</code> object allows clients to discover the operations available to perform on the content, and to instantiate the appropriate component to perform those operations. See “The JavaBeans Activation Framework” on page 41 for details describing the data typing framework.</td>
</tr>
<tr>
<td>As an object in the Java programming language</td>
<td>The <code>Part</code> interface declares the <code>getContent</code> method that returns the content as an object in the Java programming language. The type of the returned object is dependent on the content’s data type. If the content is of type multipart, the <code>getContent</code> method returns a <code>Multipart</code> object, or a <code>Multipart</code> subclass object. The <code>getContent</code> method returns an input stream for unknown content-types. Note that the <code>getContent</code> method uses the <code>DataHandler</code> internally to obtain the native form.</td>
</tr>
</tbody>
</table>

The `setDataHandler(DataHandler)` method specifies content for a new `Part` object, as a step toward the construction of a new message. The `Part` also provides some convenience methods to set up most common content types.

Part provides the `writeTo` method that writes its byte stream in mail-safe form suitable for transmission. This byte stream is typically an aggregation of the `Part` attributes and the byte stream for its content.
The Address Class

The Address class represents email addresses. The Address class is an abstract class. Subclasses provide implementation-specific semantics.

The BodyPart Class

The BodyPart class is an abstract class that implements the Part interface in order to define the attribute and content body definitions that Part declares. It does not declare attributes that set From, To, Subject, ReplyTo, or other address header fields, as a Message object does.

A BodyPart object is intended to be inserted into a Multipart container, later accessed via a multipart message.
The Multipart Class

The Multipart class implements multipart messages. A multipart message is a Message object where the content-type specifier has been set to multipart. The Multipart class is a container class that contains objects of type Bodypart. A Bodypart object is an instantiation of the Part interface—it contains either a new Multipart container object, or a DataHandler object.

The figure below illustrates the structure and content of a multipart message:

A multipart message is a simple message object where the Content-Type is set to 'multipart,' and the Content Body carries a reference to a Multipart object.

A Multipart object is a container of Bodypart objects, where each Bodypart can contain either a DataHandler object, or another Multipart object.
Note that Multipart objects can be nested to any reasonable depth within a multipart message, in order to build an appropriate structure for data carried in DataHandler objects. Therefore, it is important to check the ContentType header for each BodyPart element stored within a Multipart container. The figure below illustrates a typical nested Multipart message.

Typically, the client calls the getContentType method to get the content type of a message. If getContentType returns a MIME-type whose primary type is multipart, then the client calls getContent to get the Multipart container object.

The Multipart object supports several methods that get, create, and remove individual BodyPart objects.

```java
public int getCount() throws MessagingException;

public Body getBodyPart(int index)
    throws MessagingException;
```
The Multipart class implements the javax.beans.DataSource interface. It can act as the DataSource object for javax.beans.DataHandler and javax.beans.DataContentHandler objects. This allows message-aware content handlers to handle multipart data sources more efficiently, since the data has already been parsed into individual parts.

This diagram illustrates the structure of a multipart message, and shows calls from the associated Message and Multipart objects, for a typical call sequence returning a BodyPart containing text/plain content.

In this figure, the ContentType attribute of a Message object indicates that it holds a multipart content. Use the getContent method to obtain the Multipart object.
This code sample below shows the retrieval of a Multipart object. See “Examples Using the JavaMail API” on page 63 for examples that traverse a multipart message and examples that create new multipart messages.

```java
Multipart mp = (Multipart)message.getContent();
int count = mp.getCount();
BodyPart body_part;
for (int i = 0; i < count; i++)
    body_part = mp.getBodyPart(i);
```

**The Flags Class**

Flags objects carry flag settings that describe the state of a Message object within its containing folder. The Message.getFlags method returns a Flags object that holds all the flags currently set for that message.

The setFlags(Flags f, boolean set) method sets the specified flags for that message. The add(Flags.Flag f) method on a Flags object sets the specified flag; the contains(Flags.Flag f) method returns whether the specified flag is set.

<table>
<thead>
<tr>
<th>Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSWERED</td>
<td>Clients set this flag to indicate that this message has been answered.</td>
</tr>
<tr>
<td>DRAFT</td>
<td>Indicates that this message is a draft.</td>
</tr>
<tr>
<td>FLAGGED</td>
<td>No defined semantics. Clients can use this flag to mark a message in some user-defined manner.</td>
</tr>
<tr>
<td>RECENT</td>
<td>This message is newly arrived in this folder. This flag is set when the message is first delivered into the folder and cleared when the containing folder is closed. Clients cannot set this flag.</td>
</tr>
<tr>
<td>SEEN</td>
<td>Marks a message that has been opened. A client sets this flag implicitly when the message contents are retrieved.</td>
</tr>
<tr>
<td>DELETED</td>
<td>Allows undoable message deletion. Setting this flag for a message marks it deleted but does not physically remove the message from its folder. The client calls the expunge method on a folder to remove all deleted messages in that folder.</td>
</tr>
</tbody>
</table>

Note that a folder is not guaranteed to support either standard system flags or arbitrary user flags. The getPermanentFlags method in a folder returns a Flags object that contains all the system flags supported by that Folder implementation. The presence of the special USER flag indicates that the client can set arbitrary user-definable flags on any message belonging to this folder.
Message Creation And Transmission

The `Message` class is abstract, so an appropriate subclass must be instantiated to create a new `Message` object. A client creates a message by instantiating an appropriate `Message` subclass.

For example, the `MimeMessage` subclass handles Internet email messages. Typically, the client application creates an email message by instantiating a `MimeMessage` object, and passing required attribute values to that object. In an email message, the client defines `Subject`, `From`, and `To` attributes. The client then passes message content into the `MimeMessage` object by using a suitably configured `DataHandler` object. See “Message Composition” on page 45 for details.

After the `Message` object is constructed, the client calls the `Transport.send` method to route it to its specified recipients. See “Transport Protocols and Mechanisms” on page 51 for a discussion of the transport process.
Chapter 5: The Mail Session

A mail `Session` object manages the configuration options and user authentication information used to interact with messaging systems.

The JavaMail API supports simultaneous multiple sessions. Each session can access multiple message stores and transports. Any desktop application that needs to access the current primary message store can share the default session. Typically the mail-enabled application establishes the default session, which initializes the authentication information necessary to access the user’s Inbox folder. Other desktop applications then use the default session when sending or accessing mail on behalf of the user. When sharing the session object, all applications share authentication information, properties, and the rest of the state of the object.

For example,

- To create a `Session` using a static factory method:
  ```java
  Session session = Session.getInstance(props, authenticator);
  ```
- To create an unshared session, or to access the default session.
  ```java
  Session defaultSession = Session.getDefaultInstance(props, authenticator);
  ```

The `Properties` object that initializes the `Session` contains default values and other configuration information. It is expected that clients using the APIs set the values for the listed properties, especially `mail.store.protocol`, `mail.transport.protocol`, `mail.host`, `mail.user`, and `mail.from`, since the defaults are unlikely to work in all cases. See “Environment Properties” on page 61 for a list of properties used by the JavaMail APIs and their defaults.

Some messaging system implementations can use additional properties. Typically the properties object contains user-defined customizations in addition to system-wide defaults. Mail-enabled application logic determines the appropriate set of properties. Lacking a specific requirement, the application can use the system properties object retrieved from the `System.getProperties` method.

The `Authenticator` object controls security aspects for the `Session` object. The messaging system uses it as a callback mechanism to interact with the user when a password is required to login to a messaging system. It indirectly controls access to the default session, as described below.

Clients using JavaMail can register `PasswordAuthentication` objects with the `Session` object for use later in the session or for use by other users of the same session. Because `PasswordAuthentication` objects contain passwords, access to
this information must be carefully controlled. Applications that create Session objects must restrict access to those objects appropriately. In addition, the Session class shares some responsibility for controlling access to the default session object.

The first call to the getDefaultInstance method creates a new Session object and associates it with the Authenticator object. Subsequent calls to the getDefaultInstance method compare the Authenticator object passed in with the Authenticator object saved in the default session. Access to the default session is allowed if both objects have been loaded by the same class loader. Typically, this is the case when both the default session creator and the program requesting default session access are in the same “security domain.” Also, if both objects are null, access is allowed. Using null to gain access is discouraged, because this allows access to the default session from any security domain.

A mail-enabled client uses the Session object to retrieve a Store or Transport object in order to read or send mail. Typically, the client retrieves the default Store or Transport object based on properties loaded for that session:

```java
Store store = session.getStore();
```

The client can override the session defaults and access a Store or Transport object that implements a particular protocol.

```java
Store store = session.getStore("imap");
```

See “The Provider Registry” on page 26 for details.

Implementations of Store and Transport objects will be told the session to which they have been assigned. They can then make the Session object available to other objects contained within this Store or Transport objects using application-dependent logic.

The Provider Registry

The Provider Registry allows providers to register their protocol implementations to be used by JavaMail APIs. It provides a mechanism for discovering available protocol, for registering new protocols, and for specifying default implementations.

Resource Files

The providers for JavaMail APIs are configured using the following files:

- javamail.providers and javamail.default.providers
- javamail.address.map and javamail.default.address.map

Each javamail.X resource file is searched in the following order:

1. java.home/lib/javamail.X
2. META-INF/javamail.X
3. META-INF/javamail.default.X
The first method allows the user to include their own version of the resource file by placing it in the \texttt{lib} directory where the \texttt{java.home} property points. The second method allows an application that uses the JavaMail APIs to include their own resource files in their application’s or jar file’s \texttt{META-INF} directory. The \texttt{javamail.default.X} default files are part of the JavaMail mail.jar file.

File location depends upon how the \texttt{ClassLoader.getResource} method is implemented. Usually, the \texttt{getResource} method searches through \texttt{CLASSPATH} until it finds the requested file and then stops. JDK 1.1 has a limitation that the number of files of each type that will be found in the \texttt{CLASSPATH} is limited to one. However, this only affects method two, above; method one is loaded from a specific location (if allowed by the \texttt{SecurityManager}) and method three uses a different name to ensure that the default resource file is always loaded successfully.

The ordering of entries in the resource files matters. If multiple entries exist, the first entries take precedence over the latter entries as the initial defaults. For example, the first IMAP provider found will be set as the default IMAP implementation until explicitly changed by the application.

The user- or system-supplied resource files augment, they do not override, the default files included with the JavaMail APIs. This means that all entries in all files loaded will be available.

\texttt{javamail.providers} and \texttt{javamail.default.providers}

These resource files specify the stores and transports that are available on the system, allowing an application to “discover” what store and transport implementations are available. The protocol implementations are listed one per line. The file format defines four attributes that describe a protocol implementation. Each attribute is an ”=”-separated name-value pair with the name in lowercase. Each name-value pair is semi-colon (”;”) separated.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protocol</td>
<td>Name assigned to protocol. For example, ’smtp’ for Transport.</td>
</tr>
<tr>
<td>type</td>
<td>Valid entries are “store” and “transport”.</td>
</tr>
<tr>
<td>class</td>
<td>Class name that implements this protocol.</td>
</tr>
<tr>
<td>vendor</td>
<td>Optional string identifying the vendor.</td>
</tr>
<tr>
<td>version</td>
<td>Optional string identifying the version.</td>
</tr>
</tbody>
</table>

\textit{Table 5-1} Protocol Attributes
Here’s an example of META-INF/javamail.default.providers file contents:

```
protocol=imap; type=store;
class=com.sun.mail.imap.IMAPStore;
vendor=SunMicrosystems,Inc;
protocol=smtp; type=transport;
class=com.sun.mail.smtp.SMTPTransport;
```

```
javamail.address.map and javamail.default.address.map
```

These resource files map transport address types to the transport protocol. The javax.mail.Address.getType() method returns the address type. The javamail.address.map file maps the transport type to the protocol. The file format is a series of name-value pairs. Each key name should correspond to an address type that is currently installed on the system; there should also be an entry for each javax.mail.Address implementation that is present if it is to be used. For example, javax.mail.internet.InternetAddress.getType() returns rfc822. Each referenced protocol should be installed on the system. For the case of news, below, the client should install a Transport provider supporting the nntp protocol.

Here are the typical contents of a javamail.address.map file.

```
rfc822=smtp
news=nntp
```

**Provider**

Provider is a class that describes a protocol implementation. The values come from the javamail.providers and javamail.default.providers resource files.

**Protocol Selection and Defaults**

The constructor for the Session object initializes the appropriate variables from the resource files. The order of the protocols in the resource files determines the initial defaults for protocol implementations. The methods, getProviders(), (getProvider() and setProvider() allow the client to discover the available (installed) protocol implementations, and to set the protocols to be used by default.

At runtime, an application may set the default implementation for a particular protocol. It can set the mail.protocol.class property when it creates the Session object. This property specifies the class to use for a particular protocol. The getProvider() method consults this property first.

The code can also call setProviders() passing in a Provider that was returned by the discovery methods. A Provider object cannot be explicitly created; it must be retrieved using the getProviders() method.
In either case, the provider specified must be one of the ones configured in the resource files. Note that the methods described here allow the client to choose from among preconfigured implementations, but doesn’t allow it to configure a new implementation.

**Example Scenarios**

**Scenario 1:** The client application invokes the default protocols:

```java
class Application1 {
    init() {
        // application properties include the JavaMail
        // required properties: mail.store.protocol,
        // mail.transport.protocol, mail.host, mail.user
        Properties props = loadApplicationProps();
        Session session = Session.getInstance(props, null);

        // get the store implementation of the protocol
        // defined in mail.store.protocol; the implementation
        // returned will be defined by the order of entries in
        // javamail.providers & javamail.default.providers
        try {
            Store store = session.getStore();
            store.connect();
        } catch (MessagingException mex) {} ...
    }
}
```

**Scenario 2:** The client application presents available implementations to the user and then sets the user’s choice as the default implementation:

```java
class Application2 {
    init() {
        // application properties include the JavaMail
        // properties: mail.store.protocol,
        // mail.transport.protocol, mail.host, mail.user
        Properties props = loadApplicationProps();
        Session session = Session.getInstance(props, null);

        // find out which implementations are available
        Provider[] providers = session.getProviders();

        // ask the user which implementations to use
        // user’s response may include a number of choices,
        // i.e. imap & nntp store providers & smtp transport
        Provider[] userChosenProviders = askUserWhichProvidersToUse(providers);

        // set the defaults based on users response
        for (int i = 0; i < userChosenProviders.length; i++)
            session.setProvider(userChosenProviders[i]);
    }
}
Managing Security

The `Session` class allows messaging system implementations to use the `Authenticator` object that was registered when the session was created. The `Authenticator` object is created by the application and allows interaction with the user to obtain a user name and password. The user name and password is returned in a `PasswordAuthentication` object. The messaging system implementation can ask the session to associate a user name and password with a particular message store using the `setPasswordAuthentication` method. This information is retrieved using the `getPasswordAuthentication` method. This avoids the need to ask the
user for a password when reconnecting to a Store that has disconnected, or when a second application sharing the same session needs to create its own connection to the same Store.

Messaging system implementations can register PasswordAuthentication objects with the Session object for use later in the session or for use by other users of the same session. Because PasswordAuthentication objects contain passwords, access to this information must be carefully controlled. Applications that create Session objects must restrict access to those objects appropriately. In addition, the Session class shares some responsibility for controlling access to the default Session object.

The first call to getDefaultInstance creates a new Session object and associates the Authenticator object with the Session object. Later calls to getDefaultInstance compare the Authenticator object passed in, to the Authenticator object saved in the default session. If both objects have been loaded by the same class loader, then getDefaultInstance will allow access to the default session. Typically, this is the case when both the creator of the default session and the code requesting access to the default session are in the same "security domain." Also, if both objects are null, access is allowed. This last case is discouraged because setting objects to null allows access to the default session from any security domain.

In the future, JDK 1.2 security Permissions could control access to the default session. Note that the Authenticator and PasswordAuthentication classes and their use in JavaMail is similar to the classes with the same names provided in the java.net package in JDK 1.2. As new authentication mechanisms are added to the system, new methods can be added to the Authenticator class to request the needed information. The default implementations of these new methods will fail, but new clients that understand these new authentication mechanisms can provide implementations of these methods. New classes other than PasswordAuthentication could be needed to contain the new authentication information, and new methods could be needed in the Session class to store such information. JavaMail design evolution will be patterned after the corresponding JDK classes.

---

**Store and Folder URLs**

To simplify message folder naming and to minimize the need to manage Store and Transport objects, folders can be named using URLNames. URLNames are similar to URLs except they only include the parsing of the URL string. The Session class provides methods to retrieve a Folder object given a URLName:

```java
Folder f = session.getFolder(URLName);
```

or

```java
Store s = session.getStore(URLName);
```
Chapter 5: The Mail Session
Store and Folder URLs
Chapter 6:
Message Storage And Retrieval

This section describes JavaMail message storage facilities supported by the Store and Folder classes.

Messages are contained in Folders. New messages are usually delivered to folders by a transport protocol or a delivery agent. Clients retrieve messages from folders using an access protocol.

The Store Class

The Store class defines a database that holds a Folder hierarchy and the messages within. The Store also defines the access protocol used to access folders and retrieve messages from folders. Store is an abstract class. Subclasses implement specific message databases and access protocols.

Clients gain access to a Message Store by obtaining a Store object that implements the database access protocol. Most message stores require the user to be authenticated before they allow access. The connect method performs that authentication.

For many message stores, a host name, user name, and password are sufficient to authenticate a user. The JavaMail API provides a connect method that takes this information as input parameters. Store also provides a default connect method. In either case, the client can obtain missing information from the Session object's properties, or by interacting with the user by accessing the Session's Authenticator object.

The default implementation of the connect method in the Store class uses these techniques to retrieve all needed information and then calls the protocolConnect method. The messaging system must provide an appropriate implementation of this method. The messaging system can also choose to directly override the connect method.

By default, Store queries the following properties for the user name and host name:

- mail.user property, or user.name system property (if mail.user is not set)
- mail.host

These global defaults can be overridden on a per-protocol basis by the properties:

- mail.protocol.user
- mail.protocol.host
Note that passwords can not be specified using properties.

The Store presents a default namespace to clients. Store implementations can also present other namespaces. The getDefaultFolder method on Store returns the root folder for the default namespace.

Clients terminate a session by calling the close method on the Store object. Once a Store is closed (either explicitly using the close method; or externally, if the Mail server fails), all Messaging components belonging to that Store become invalid. Typically, clients will try to recover from an unexpected termination by calling connect to reconnect to the Store object, and then fetching new Folder objects and new Message objects.

**Store Events**

Store sends the following events to interested listeners:

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectionEvent</td>
<td>Generated when a connection is successfully made to the Store, or when an existing connection is terminated or disconnected.</td>
</tr>
<tr>
<td>StoreEvent</td>
<td>Communicates alerts and notification messages from the Store to the end user. The getMessageType method returns the event type, which can be one of: ALERT or NOTICE. The client must display ALERT events in some fashion that calls the user’s attention to the message.</td>
</tr>
<tr>
<td>FolderEvent</td>
<td>Communicates changes to any folder contained within the Store. These changes include creation of a new Folder, deletion of an existing Folder, and renaming of an existing Folder.</td>
</tr>
</tbody>
</table>

**The Folder Class**

The Folder class represents a folder containing messages. Folders can contain subfolders as well as messages, thus providing a hierarchical structure. The getType method returns whether a Folder can hold subfolders, messages, or both. Folder is an abstract class. Subclasses implement protocol-specific Message Folders.

The getDefaultFolder method for the corresponding Store object returns the root folder of a user’s default folder hierarchy. The list method for a Folder returns all the subfolders under that folder. The getFolder(String name) method for a Folder object returns the named subfolder. Note that this subfolder need not exist physically in the store. The exists method in a folder indicates whether this folder exists. A folder is created in the store by invoking its create method.
A closed Folder object allows certain operations, including deleting the folder, renaming the folder, listing subfolders, creating subfolders and monitoring for new messages. The open method opens a Folder object. All Folder methods except open, delete, and renameTo are valid on an open Folder object. Note that the open method is applicable only on Folder objects that can contain messages.

The messages within a Folder are sequentially numbered, from 1 through the total number of messages. This ordering is referred to as the “mailbox order” and is usually based on the arrival time of the messages in the folder. As each new message arrives into a folder, it is assigned a sequence number that is one higher than the previous number of messages in that folder. The getMessageNumber method on a Message object returns its sequence number.

The sequence number assigned to a Message object is valid within a session, but only as long as it retains its relative position within the Folder. Any change in message ordering can change the Message object’s sequence number. Currently this occurs when the client calls expunge to remove deleted messages and renumber messages remaining in the folder.

A client can reference a message stored within a Folder either by its sequence number, or by the corresponding Message object itself. Since a sequence number can change within a session, it is preferable to use Message objects rather than sequence numbers as cached references to messages. Clients extending JavaMail are expected to provide light-weight Message objects that get filled ‘on-demand’, so that calling the getMessages method on a Folder object is an inexpensive operation, both in terms of CPU cycles and memory. For instance, an IMAP implementation could return Message objects that contain only the corresponding IMAP UIDs.

**The FetchProfile Method**

The Message objects returned by a Folder object are expected to be light-weight objects. Invoking getXXX methods on a Message cause the corresponding data items to be loaded into the object on demand. Certain Store implementations support batch fetching of data items for a range of Messages. Clients can use such optimizations, for example, when filling the header-list window for a range of messages. The FetchProfile method allows a client to list the items it will fetch in a batch for a certain message range.

The following code illustrates the use of FetchProfile when fetching Messages from a Folder. The client fills its header-list window with the Subject, From, and X-mailer headers for all messages in the folder.

```java
Message[] msgs = folder.getMessages();
FetchProfile fp = new FetchProfile();
fp.add(FetchProfile.Item.ENVELOPE);
fp.add("X-mailer");
folder.fetch(msgs, fp);
for (int i = 0; i < folder.getMessageCount(); i++) {
    display(msgs[i].getFrom());
    display(msgs[i].getSubject());
    display(msgs[i].getHeader("X-mailer"));
}
```
Folder Events

Folders generate events to notify listeners of any change in either the folder or in its Messages list. The client can register listeners to a closed Folder, but generates a notification event only after that folder is opened.

Folder supports the following events:

- **ConnectionEvent**: This event is generated when a Folder is opened or closed. When a Folder closes (either because the client has called `close` or from some external cause), all Messaging components belonging to that Folder become invalid. Typically, clients will attempt to recover by reopening that Folder, and then fetching Message objects.

- **FolderEvent**: This event is generated when the client creates, deletes or renames this folder. Note that the Store object containing this folder can also generate this event.

- **MessageCountEvent**: This event notifies listeners that the message count has changed. The following actions can cause this change:
  - **Addition** of new Messages into the Folder, either by a delivery agent or because of an append operation. The new Message objects are included in the event.
  - **Removal** of existing messages from this Folder. Removed messages are referred to as expunged messages. The `isExpunged` method returns true for removed Messages and the `getMessageNumber` method returns the original sequence number assigned to that message. All other Message methods throw a `MessageRemovedException`. See “The Folder Class” on page 34 for a discussion of removing deleted messages in shared folders. The expunged Message objects are included in the event. An expunged message is invalid and should be pruned from the client's view as early as possible. See “The Expunge Process” on page 37 for details on the `expunge` method.
The Expunge Process

Deleting messages from a Folder is a two-phase operation. Setting the DELETED flag on messages marks them as deleted, but it does not remove them from the Folder. The deleted messages are removed only when the client invokes the expunge method on that Folder object. The Folder object then notifies listeners by firing an appropriate MessageEvent. The MessageEvent object contains the expunged Message objects. Note that the expunge method also returns the expunged Message objects. The Folder object also rennumbers the messages falling after the expunged messages in the message list. Thus, when the expunge method returns, the sequence number of those Message objects will change. Note, however, that the expunged messages still retain their original sequence numbers.

Since expunging a folder can remove some messages from the folder and renumber others, it is important that the client synchronize itself with the expunged folder as early as possible. The next sections describe a set of recommendations for clients wanting to expunge a Folder:

- Expunge the folder; close it; and then reopen and refetch messages from that Folder. This ensures that the client was notified of the updated folder state. In fact, the client can just issue the close method with the expunge parameter set to true to force an expunge of the Folder during the close operation, thus even avoiding the explicit call to expunge.

- The previous solution might prove to be too simple or too drastic in some circumstances. This paragraph describes the scenario of a more complex client expunging a single access folder; for example, a folder that allows only one read-write connection at a time. The recommended steps for such a client after it issues the expunge command on the folder are:

  - Update its message count, either by decrementing it by the number of expunged messages, or by invoking the getMessageCount method on the Folder.

  - If the client uses sequence numbers to reference messages, it must account for the renumbering of Message objects subsequent to the expunged messages. Thus if a folder has 5 messages as shown below, (sequence numbers are within parenthesis), and if the client is notified that messages A and C are removed, it should account for the renumbering of the remaining messages as shown in the second figure.
The client should prune expunged messages from its internal storage as early as possible.

The expunge process becomes complex when dealing with a shared folder that can be edited. Consider the case where two clients are operating on the same folder. Each client possesses its own Folder object, but each Folder object actually represents the same physical folder.

If one client expunges the shared folder, any deleted messages are physically removed from the folder. The primary client can probably deal with this appropriately since it initiated this process and is ready to handle the consequences. However, secondary clients are not guaranteed to be in a state where they can handle an unexpected Message removed event—especially if the client is heavily multithreaded or if it uses sequence numbers.

To allow clients to handle such situations gracefully, the JavaMail API applies the following restrictions to Folder implementations:

- A Folder can remove and renumber its Messages only when it is explicitly expunged using the expunge method. When the folder is implicitly expunged, it marks any expunged messages as expunged, but it still maintains access to those Message objects. This means that the following state is maintained when the Folder is implicitly expunged:
  - getMessages returns expunged Message objects together with valid message objects. However, an expunged message can throw the MessageExpungedException if direct access is attempted.
  - The messages in the Folder should not be renumbered.
  - The implicit expunge operation can not change the total Folder message count.

A Folder can notify listeners of “implicit” expunges by generating appropriate MessageEvents. However, the removed field in the event must be set to false to indicate that the message is still in the folder. When this Folder is explicitly expunged, then the Folder must remove all expunged messages, renumber it's internal Message cache, and generate MessageEvents for all the expunged messages, with each removed flag set to true.
The recommended set of actions for a client under the above situation is as follows:

- Multithreaded clients that expect to handle shared folders are advised not to use sequence numbers.

- If a client receives a MessageEvent indicating message removal, it should check the removed flag. If the flag is false, this indicates that another client has removed the message from this folder. This client might want to issue an expunge request on the folder object to synchronize it with the physical folder (but note the caveats in the previous section about using a shared folder). Alternatively, this client might want to close the Folder object (without expunging) and reopen it to synchronize with the physical folder (but note that all message objects would need to be refreshed in this case). The client may also mark the expunged messages in order to notify the end user.

- If the removed flag was set to true, the client should follow earlier recommendations on dealing with explicit expunges.

---

The Search Process

Search criteria are expressed as a tree of search-terms, forming a parse tree for the search expression. The SearchTerm class represents search terms. This is an abstract class with a single method:

```java
public boolean match(Message msg);
```

Subclasses implement specific matching algorithms by implementing the `match` method. Thus new search terms and algorithms can be easily introduced into the search framework by writing the required code using the Java programming language.

The search package provides a set of standard search terms that implement specific match criteria on Message objects. For example, SubjectTerm pattern-matches the given String with the subject header of the given message.

```java
public final class SubjectTerm extends StringTerm {
    public SubjectTerm(String pattern);
    public boolean match(Message m);
}
```
The search package also provides a set of standard logical operator terms that can be used to compose complex search terms. These include AndTerm, OrTerm and NotTerm.

```java
final class AndTerm extends SearchTerm {
    public AndTerm(SearchTerm t1, SearchTerm t2);
    public boolean match(Message msg) {
        // The AND operator
        for (int i=0; i < terms.length; i++)
            if (!terms[i].match(msg))
                return false;
        return true;
    }
}
```

The Folder class supports searches on messages through these search method versions:

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

These methods return the Message objects matching the specified search term. The default implementation applies the search term on each Message object in the specified range. Other implementations may optimize this; for example, the IMAP Folder implementation maps the search term into an IMAP SEARCH command that the server executes.
Chapter 7:
The JavaBeans Activation Framework

JavaMail relies heavily on the JavaBeans Activation Framework (JAF) to determine the MIME data type, to determine the commands available on that data, and to provide a software component corresponding to a particular behavior. The JAF specification is part of the "Glasgow" JavaBeans specification. More details can be obtained from http://java.sun.com/beans/glasgow/jaf.html

This section explains how the JavaMail and JAF APIs work together to manage message content. It describes how clients using JavaMail can access and operate on the content of Messages and BodyParts. This discussion assumes you are familiar with the JAF specification posted at http://java.sun.com.

Accessing the Content

For a client using JavaMail, arbitrary data is introduced to the system in the form of mail messages. The javax.mail.Part interface allows the client to access the content. Part consists of a set of attributes and a "content". The Part interface is the common base interface for Messages and BodyParts. A typical mail message has one or more body parts, each of a particular MIME type.

Anything that deals with the content of a Part will use the Part’s DataHandler. The content is available through the DataHandlers either as an InputStream or as an object in the Java programming language. The Part also defines convenience methods that call through to the DataHandler. For example: the Part.getContent method is the same as calling Part.getDataHandler().getContent() and the Part.getInputStream method is the same as Part.getDataHandler().getInputStream().

The content returned (either via an InputStream or an object in the Java programming language) depends on the MIME type. For example: a Part that contains textual content returns the following:

- The Part.getInputStream method returns text/plain
- The Part.getInputStream method returns an InputStream containing the bytes of the text
- The Part.getContent method returns a java.lang.String object
Content is returned either as an input stream, or as an object in the Java programming language.

- When an InputStream is returned, any mail-specific encodings are decoded before the stream is returned.

- When an object in the Java programming language is returned using the getContent method, the type of the returned object depends upon the content itself. In the JavaMail API, any Part with a main content type set to "multipart/" (any kind of multipart) should return a javax.mail.Multipart object from the getContent method. A Part with a content type of message/rfc822 returns a javax.mail.Message object from the getContent method.

**Example: Message Output**

This example shows how you can traverse Parts and display the data contained in a message.

```java
public void printParts(Part p) {
    Object o = p.getContent();
    if (o instanceof String) {
        System.out.println("This is a String");
        System.out.println((String)o);
    } else if (o instanceof Multipart) {
        System.out.println("This is a Multipart");
        Multipart mp = (Multipart)o;
        int count = mp.getCount();
        for (int i = 0; i < count; i++) {
            printParts(mp.getBodyPart(i));
        }
    } else if (o instanceof InputStream) {
        System.out.println("This is just an input stream");
        InputStream is = (InputStream)o;
        int c;
        while ((c = is.read()) != -1)
            System.out.write(c);
    }
}
```
Operating on the Content

The DataHandler allows clients to discover the operations available on the content of a Message, and to instantiate the appropriate JavaBeans to perform those operations. The most common operations on Message content are view, edit and print.

Example: Viewing a Message

Consider a Message “Viewer” Bean that presents a user interface that displays a mail message. This example shows how a viewer bean can be used to display the content of a message (that usually is text/plain, text/html, or multipart/mixed).

Note – Perform error checking to ensure that a valid Component was created.

```java
// message passed in as parameter
void setMessage(Message msg) {  
    DataHandler dh = msg.getDataHandler();  
    CommandInfo cinfo = dh.getCommand("view");  
    Component comp = (Component) dh.getBean(cinfo);  
    this.setMainViewer(comp);  
}
```

Example: Showing Attachments

In this example, the user has selected an attachment and wishes to display it in a separate dialog. The client locates the correct viewer object as follows.

```java
// Retrieve the BodyPart from the current attachment
BodyPart bp = getSelectedAttachment();

DataHandler dh = bp.getDataHandler();
CommandInfo cinfo = dh.getCommand("view");
Component comp = (Component) dh.getBean(cinfo);

// Add viewer to dialog Panel
MyDialog myDialog = new MyDialog();
myDialog.add(comp);

// display dialog on screen
myDialog.show();
```

See “Setting Message Content” on page 47 for examples that construct a message for a send operation.
Adding Support for Content Types

Support for commands acting on message data is an implementation task left to the client. JavaMail and JAF APIs intend for this support to be provided by a JAF-Aware JavaBean. Almost all data will require edit and view support.

Currently, the JavaMail API does not provide viewer JavaBeans. The JAF does provide two very simple JAF-aware viewer beans: A Text Viewer and Image Viewer. These beans handle data where content-type has been set to text/plain or image/gif.

Developers writing a JavaMail client need to write additional viewers that support some of the basic content types—specifically message/rfc822, multipart/mixed, and text/plain. These are the usual content-types encountered when displaying a Message, and they provide the look and feel of the application.

Content developers providing additional data types should refer to the JAF specification, that discusses how to create DataContentHandlers and Beans that operate on those contents.
Chapter 8:
Message Composition

This section describes the process used to instantiate a message object, add content to that message, and send it to its intended list of recipients.

The JavaMail API allows a client program to create a message of arbitrary complexity. Messages are instantiated from the `Message` subclass. The client program can manipulate any message as if it had been retrieved from a `Store`.

Building a Message Object

To create a message, a client program instantiates a `Message` object, sets appropriate attributes, and then inserts the content.

- The attributes specify the message address and other values necessary to send, route, receive, decode and store the message. Attributes also specify the message structure and data content type.

- Message content is carried in a `DataHandler` object, that carries either data or a `Multipart` object. A `DataHandler` carries the content body and provides methods the client uses to handle the content. A `Multipart` object is a container that contains one or more `Bodypart` objects, each of which can in turn contain `DataHandler` objects.

Message Creation

`javax.mail.Message` is an abstract class that implements the `Part` interface. Therefore, to create a message object, select a message subclass that implements the appropriate message type.

For example, to create a Mime message, a JavaMail client instantiates an empty `javax.mail.internet.MimeMessage` object passing the current `Session` object to it:

```java
Message msg = new MimeMessage(session);
```
Setting Message Attributes

The Message class provides a set of methods that specify standard attributes common to all messages. The MimeMessage class provides additional methods that set MIME-specific attributes. The client program can also set non-standard attributes (custom headers) as name-value pairs.

The methods for setting standard attributes are listed below:

```java
public class Message {
    public void setFrom(Address addr);
    public void setFrom(); // retrieves from system
    public void setRecipients(RecipientType type, Address[] addrs);
    public void setReplyTo(Address[] addrs);
    public void setSentDate(Date date);
    public void setSubject(String subject);
    ...
}
```

The Part interface specifies the following method, that sets custom headers:

```java
public void setHeader(String name, String value)
```

The setRecipients method takes a RecipientType as its first parameter, which specifies which recipient field to use. Currently, Message.RecipientType.TO, Message.RecipientType.CC, and Message.RecipientType.BCC are defined. Additional RecipientTypes may be defined as necessary.

The Message class provides two versions of the of the setFrom method:

- `setFrom(Address addr)` specifies the sender explicitly from an Address object parameter.
- `setFrom()` retrieves the sender’s username from the local system.

The code sample below sets attributes for the MimeMessage just created. First, it instantiates Address objects to be used as To and From addresses. Then, it calls set methods, which equate those addresses to appropriate message attributes:

```java
toAddrs[] = new InternetAddress[1];
toAddrs[0] = new InternetAddress("luke@rebellion.gov");
Address fromAddr =
    new InternetAddress("han.solo@smuggler.com");

msg.setFrom(fromAddr);
msg.setRecipients(Message.RecipientType.TO, toAddrs);
msg.setSubject("Takeoff time.");
msg.setSentDate(new Date());
```
**Setting Message Content**

The `Message` object carries content data within a `DataHandler` object. To add content to a `Message`, a client creates content, instantiates a `DataHandler` object, places content into that `DataHandler` object, and places that object into a `Message` object that has had its attributes defined.

The JavaMail API provides two techniques that set message content. The first technique uses the `setDataHandler` method. The second technique uses the `setContent` method.

Typically, clients add content to a `DataHandler` object by calling `setDataHandler(DataHandler)` on a `Message` object. The `DataHandler` is an object that encapsulates data. The data is passed to the `DataHandler`'s constructor as either a `DataSource` (a stream connected to the data) or as an object in the Java programming language. The `InputStream` object creates the `DataSource`. See “The JavaBeans Activation Framework” on page 41 for additional information.

```java
public class DataHandler {
    DataHandler(DataSource dataSource);
    DataHandler(Object data, String mimeType);
}
```

The code sample below shows how to place text content into an `InternetMessage`. First, create the text as a string object. Then, pass the string into a `DataHandler` object, together with its MIME type. Finally, add the `DataHandler` object to the message object:

```java
// create brief message text
String content = "Leave at 300."

// instantiate the DataHandler object
DataHandler data = new DataHandler(content, "text/plain");

// Use setDataHandler() to insert data into the
// new Message object
msg.setDataHandler(data);
```

Alternately, `setContent` implements a simpler technique that takes the data object and its MIME type. `setContent` creates the `DataHandler` object automatically:

```java
// create the message text
String content = "Leave at 300."

// call setContent to pass content and content type
// together into the message object
msg.setContent(content, "text/plain");
```
When the client calls `Transport.send()` to send this message, the recipient will receive the message below, using either technique:

```plaintext
Date: Wed, 23 Apr 1997 22:38:07 -0700 (PDT)
From: han.solo@smuggler.com
Subject: Takeoff time
To: luke@rebellion.gov

Leave at 300.
```

---

**Building a MIME Multipart Message**

Follow these steps to create a MIME Multipart Message:

1. Instantiate a new `MimeMultipart` object, or a subclass.
2. Create `MimeBodyParts` for the specific message parts. Use the `setContent` method or the `setDataHandler` method to create the content for each `Bodypart`, as described in the previous section.

---

**Note** – The default subtype for a `MimeMultipart` object is *mixed*. It can be set to other subtypes as required. `MimeMultipart` subclasses might already have their subtype set appropriately.
3. Insert the **Multipart** object into the **Message** object by calling `setContent(Multipart)` within a newly-constructed **Message** object.

The example below creates a **Multipart** object and then adds two message parts to it. The first message part is a text string, “Spaceport Map,” and the second contains a document of type “application/postscript.” Finally, this multipart object is added to a **MimeMessage** object of the type described above.

```java
// Instantiate a Multipart object
MimeMultipart mp = new MimeMultipart();

// create the first bodypart object
MimeBodyPart b1 = new MimeBodyPart();

// create textual content
// and add it to the bodypart object
b1.setContent("Spaceport Map","text/plain");
mp.addBodyPart(b1);

// Multipart messages usually have more than
// one body part. Create a second body part
// object, add new text to it, and place it
// into the multipart message as well. This
// second object holds postscript data.

MimeBodyPart b2 = new MimeBodyPart(); b2.setContent(map,"application/postscript");
mp.addBodyPart(b2);

// Create a new message object as described above,
// and set its attributes. Add the multipart
// object to this message and call saveChanges()
// to write other message headers automatically.

Message msg = new MimeMessage(session);

// Set message attributes as in a singlepart
// message.

msg.setContent(mp);              // add Multipart
msg.saveChanges();               // save changes
```

After all message parts are created and inserted, call the `saveChanges` method to ensure that the client writes appropriate message headers. This is identical to the process followed with a single part message. Note that the JavaMail API calls the `saveChanges` method implicitly during the `send` process, so invoking it is unnecessary and expensive if the message is to be sent immediately.
Chapter 9:
Transport Protocols and Mechanisms

The Transport abstract class defines the message submission and transport protocol. Subclasses of the Transport class implement SMTP and other transport protocols.

Obtaining the Transport Object

The Transport object is seldom explicitly created. The getTransport method obtains a Transport object from the Session factory. The JavaMail API provides three versions of the getTransport method:

```java
public class Session {
    public Transport getTransport(Address address);
    public Transport getTransport(String protocol);
    public Transport getTransport();
}
```

- `getTransport(Address address)` returns the implementation of the transport class based on the address type. A user-extensible map defines which transport type to use for a particular address. For example, if the address is an `InternetAddress`, and `InternetAddress` is mapped to a protocol that supports SMTP then `SMTPTransport` can be returned.

- The client can also call `getTransport("SMTP")` to request SMTP, or another transport implementation protocol.

- `getTransport()` returns the transport specified in the `mail.transport.protocol` property.

See “The Mail Session” on page 25 for details.

Transport Methods

The Transport class provides the `connect` and `protocolConnect` methods, which operate similarly to those on the Store class. See “The Store Class” on page 33 for details.

A Transport object generates a ConnectionEvent to notify its listeners of a successful or a failed connection. A Transport object can throw an IOException if the connection fails.
Transport implementations should ensure that the message specified is of a known type. If the type is known, then the Transport object sends the message to its specified destinations. If the type is not known, then the Transport object can attempt to reformat the Message object into a suitable version using gatewaying techniques, or it can throw a MessagingException, indicating failure. For example, the SMTP transport implementation recognizes MimeMessages. It invokes the writeTo method on a MimeMessage object to generate a RFC822 format byte stream that is sent to the SMTP host.

The message is sent using the Transport.send static method or the sendMessage instance method. The Transport.send method is a convenience method that instantiates the transports necessary to send the message, depending on the recipients' addresses, and then passes the message to each transport's sendMessage method. Alternatively, the client can get the transport that implements a particular protocol itself and send the message using the sendMessage method. This adds the benefit of being able to register as event listeners on the individual transports.

Note that the Address[] argument passed to the send and sendMessage methods do not need to match the addresses provided in the message headers. Although these arguments usually will match, the end-user determines where the messages are actually sent. This is useful for implementing the Bcc: header, and other similar functions.

---

**Transport Events**

Clients can register as listeners for events generated by transport implementations. (Note that the abstract Transport class doesn't fire any events, only particular protocol implementations generate events). There are two events generated: ConnectionEvent and TransportEvent.

**ConnectionEvent**

If the transport connects successfully, it will fire the ConnectionEvent with the type set to OPENED. If the connection times out or is closed, ConnectionEvent with type CLOSED is generated.
TransportEvent

The sendMessage method generates a TransportEvent to its listeners. That event contains information about the method’s success or failure. There are three types of TransportEvent: MESSAGE_DELIVERED, MESSAGE_NOT_DELIVERED, MESSAGE_PARTIALLY_DELIVERED. The event contains three arrays of addresses: validSent[], validUnsent[], and invalid[] that list the valid and invalid addresses for this message and protocol.

<table>
<thead>
<tr>
<th>Transport Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MESSAGE_DELIVERED</td>
<td>When the message has been successfully sent to all recipients by this transport. validSent[] contains all the addresses. validUnsent[] and invalid[] are null.</td>
</tr>
<tr>
<td>MESSAGE_NOT_DELIVERED</td>
<td>When ValidSent[] is null, the message was not successfully sent to any recipients. validUnsent[] may have addresses that are valid. invalidSent[] may contain invalid addresses.</td>
</tr>
<tr>
<td>MESSAGE_PARTIALLY_DELIVERED</td>
<td>Message was successfully sent to some recipients but not to all. ValidSent[] holds addresses of recipients to whom the message was sent. validUnsent[] holds valid addresses but the message wasn't sent to them. invalid[] holds invalid addresses.</td>
</tr>
</tbody>
</table>
Using The Transport Class

The code segment below sends a MimeMessage using a Transport class implementing the SMTP protocol. The client creates two InternetAddress objects that specify the recipients and retrieves a Transport object from the default Session that supports sending messages to Internet addresses. Then the Session object uses a Transport object to send the message.

```java
// Get a session
Session session = Session.getInstance(props, null);

// Create an empty MimeMessage and its part
Message msg = new MimeMessage(session);
... add headers and message parts as before

// create two destination addresses
Address[] addrs = {new InternetAddress("mickey@disney.com"),
                  new InternetAddress("goofy@disney.com")};

// get a transport that can handle sending message to
// InternetAddresses. This will probably map to a transport
// that supports SMTP.
Transport trans = session.getTransport(addrs[0]);

// add ourselves as ConnectionEvent and TransportEvent listeners
trans.addConnectionListener(this);
trans.addTransportListener(this);

// connect method determines what host to use from the
// session properties
trans.connect();

// send the message to the addresses we specified above
trans.sendMessage(msg, addrs);
```
Chapter 10: 

Internet Mail

The JavaMail specification does not define any implementation. However, the API does include a set of classes that implement Internet Mail standards. Although not part of the specification, these classes can be considered part of the JavaMail package. They show how to adapt an existing messaging architecture to the JavaMail framework.

These classes implement the Internet Mail Standards defined by the RFCs listed below:

- RFC822 (Standard for the Format of Internet Text Messages)
- RFC2045, RFC2046, RFC2047 (MIME)

RFC822 describes the structure of messages exchanged across the Internet. Messages are viewed as having a header and contents. The header is composed of a set of standard and optional header fields. The header is separated from the content by a blank line. The RFC specifies the syntax for all header fields and the semantics of the standard header fields. It does not however, impose any structure on the message contents.

The MIME RFCs 2045, 2046 and 2047 define message content structure by defining structured body parts, a typing mechanism for identifying different media types, and a set of encoding schemes to encode data into mail-safe characters.

The Internet Mail package allows clients to create, use and send messages conforming to the standards listed above. It gives service providers a set of base classes and utilities they can use to implement Stores and Transports that use the Internet mail protocols. See “MimeMessage Object Hierarchy” on page 81 for a Mime class and interface hierarchy diagram.

The JavaMail MimePart interface models an entity as defined in RFC2045, Section 2.4. MimePart extends the JavaMail Part interface to add MIME-specific methods and semantics. The MimeMessage and MimeBodyPart classes implement the MimePart interface. The following figure shows the class hierarchy of these classes.
FIGURE 10-1

The MimeMessage Class

The MimeMessage class extends Message and implements MimePart. This class implements an email message that conforms to the RFC822 and MIME standards.

The MimeMessage class provides a default constructor that creates an empty MimeMessage object. The client can fill in the message later by invoking the parse method on an RFC822 input stream. Note that the parse method is protected, so that only this class and its subclasses can use this method. Service providers implementing ‘light-weight’ Message objects that are filled in on demand can generate the appropriate byte stream and invoke the parse method when a component is requested from a message. Service providers that can provide a separate byte stream for the message body (distinct from the message header) can override the getContentStream method.

The client can also use the default constructor to create new MimeMessage objects for sending. The client sets appropriate attributes and headers, inserts content into the message object, and finally calls the send method for that MimeMessage object.
This code sample creates a new MimeMessage object for sending. See “Message Composition” on page 45 and “Transport Protocols and Mechanisms” on page 51 for details.

MimeMessage m = new MimeMessage(session);
// Set FROM:
m.setFrom(new InternetAddress("jmk@Sun.COM"));
// Set TO:
InternetAddress a[] = new InternetAddress[1];
a[0] = new InternetAddress("javamail@Sun.COM");
m.setRecipients(Message.RecipientType.TO, a);
// Set content
m.setContent(data, "text/plain");
// Send message
m.send();

The MimeMessage class also provides a constructor that uses an input stream to instantiate itself. The constructor internally invokes the parse method to fill in the message. The InputStream object is left positioned at the end of the message body.

InputStream in = getMailSource(); // a stream of mail messages
MimeMessage m = null;
for (; ;) {
  try {
    m = new MimeMessage(session, in);
  } catch (MessagingException ex) {
    // reached end of message stream
    break;
  }
}

MimeMessage implements the writeTo method by writing an RFC822-formatted byte stream of its headers and body. This is accomplished in two steps: First, the MimeMessage object writes out its headers; then it delegates the rest to the DataHandler object representing the content.

The MimeBodyPart Class

The MimeBodyPart class extends BodyPart and implements the MimePart interface. This class represents a Part inside a Multipart. MimeBodyPart implements a Body Part as defined by RFC2045, Section 2.5.

The getBodyPart(int index) returns the MimeBodyPart object at the given index. MimeMultipart also allows the client to fetch MimeBodyPart objects based on their Content-IDs.

The addBodyPart method adds a new MimeBodyPart object to a MimeMultipart as a step towards constructing a new multipart MimeMessage.
The MimeMultipart Class

The MimeMultipart class extends Multipart and models a MIME multipart content within a message or a body part.

A MimeMultipart is obtained from a MimePart containing a ContentType attribute set to multipart, by invoking that part's getContent method.

The client creates a new MimeMultipart object by invoking its default constructor. To create a new multipart MimeMessage, create a MimeMultipart object (or its subclass); use set methods to fill in the appropriate MimeBodyParts; and finally, use setContent(Multipart) to insert it into the MimeMessage.

MimeMultipart also provides a constructor that takes an input stream positioned at the beginning of a MIME multipart stream. This class parses the input stream and creates the child body parts.

The getSubType method returns the multipart message MIME subtype. The subtype defines the relationship among the individual body parts of a multipart message. More semantically complex multipart subtypes are implemented as subclasses of MimeMultipart, providing additional methods that expose specific functionality.

Note that a multipart content object is treated like any other content. When parsing a MIME Multipart stream, the JavaMail implementation uses the JAF framework to locate a suitable DataContentHandler for the specific subtype and uses that handler to create the appropriate Multipart instance. Similarly, when generating the output stream for a Multipart object, the appropriate DataContentHandler is used to generate the stream.

The MimeUtility Class

MimeUtility is a utility class that provides MIME-related functions. All methods in this class are static methods. These methods currently perform the functions listed below:
Content Encoding and Decoding

Data sent over RFC 821/822-based mail systems are restricted to 7-bit US-ASCII bytes. Therefore, any non-US-ASCII content needs to be encoded into the 7-bit US-ASCII (mail-safe) format. MIME (RFC 2045) specifies the “base64” and “quoted-printable” encoding schemes to perform this encoding. The following methods support content encoding:

- The `getEncoding` method takes a `DataSource` object and returns the Content-Transfer-Encoding that should be applied to the data in that `DataSource` object to make it mail-safe.
- The `encode` method wraps an encoder around the given output stream based on the specified Content-Transfer-Encoding. The `decode` method decodes the given input stream, based on the specified Content-Transfer-Encoding.

Header Encoding and Decoding

RFC 822 restricts the data in message headers to 7bit US-ASCII characters. MIME (RFC 2047) specifies a mechanism to encode non 7bit US-ASCII characters so that they are suitable for inclusion in message headers. This section describes the methods that enable this functionality.

The header-related methods (getHeader, setHeader) in Part and Message operate on Strings. String objects contain (16 bit) Unicode characters. Since RFC 822 prohibits non US-ASCII characters in headers, clients invoking the `setHeader()` methods must ensure that the header values are appropriately encoded if they contain non US-ASCII characters.

The encoding process (based on RFC 2047) consists of two steps:

1. Convert the Unicode String into an array of bytes in another charset. This step is required because Unicode is not yet a widely used charset. Therefore, a client must convert the Unicode characters into a charset that is more palatable to the recipient.

2. Apply a suitable encoding format that ensures that the bytes obtained in the previous step are mail-safe.

The `encodeText` method combines the two steps listed above to create an encoded header. Note that as RFC 2047 specifies, only “unstructured” headers and user-defined extension headers can be encoded. It is prudent coding practice to run such header values through the encoder to be safe. Also note that the `encodeText` method encodes header values only if they contain non US-ASCII characters.

The reverse of this process (decoding) needs to be performed when handling header values obtained from a MimeMessage or MimeBodyPart using the `getHeader` set of methods, since those headers might be encoded as per RFC 2047. The `decodeText` method takes a header value, applies RFC 2047 decoding standards, and returns the
decoded value as a Unicode String. Note that this method should be invoked only on “unstructured” or user-defined headers. Also note that decodeText attempts decoding only if the header value was encoded in RFC 2047 style. It is advised that you always run header values through the decoder to be safe.

The **ContentType Class**

The **ContentType** class is a utility class that parses and generates MIME content-type headers.

To parse a MIME content-Type value, create a **ContentType** object and invoke the `toString` method.

The **ContentType** class also provides methods that match Content-Type values.

The following code fragment illustrates the use of this class to extract a MIME parameter.

```java
String type = part.getContentType();
ContentType cType = new ContentType(type);

if (cType.match("application/x-foobar"))
    iString color = cType.getParameter("color");
```

This code sample uses this class to construct a MIME Content-Type value:

```java
ContentType cType = new ContentType();
cType.setPrimaryType("application");
cType.setSubType("x-foobar");
cType.setParameter("color", "red");

String contentType = cType.toString();
```
Appendix A:

Environment Properties

This section lists the environment properties that are used by the JavaMail APIs.

Note that Applets can not determine some defaults listed in this Appendix. When writing an applet, you must specify the properties you require.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>mail.store.protocol</td>
<td>Specifies the default Message Access Protocol. The Session.getStore() method returns a Store object that implements this protocol. The client can override this property and explicitly specify the protocol with the Session.getStore(String protocol) method.</td>
<td>The first appropriate protocol in the config files</td>
</tr>
<tr>
<td>mail.transport.protocol</td>
<td>Specifies the default Transport Protocol. The Session.getTransport() method returns a Transport object that implements this protocol. The client can override this property and explicitly specify the protocol by using Session.getTransport(String protocol) method.</td>
<td>The first appropriate protocol in the config files</td>
</tr>
<tr>
<td>mail.host</td>
<td>Specifies the default Mail server. The Store and Transport object's connect methods use this property, if the protocol-specific host property is absent, to locate the target host.</td>
<td>The local machine</td>
</tr>
<tr>
<td>mail.user</td>
<td>Specifies the username to provide when connecting to a Mail server. The Store and Transport object's connect methods use this property, if the protocol-specific username property is absent, to obtain the username.</td>
<td>user.name</td>
</tr>
</tbody>
</table>
### Appendix A: Environment Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mail.protocol.host</code></td>
<td>Specifies the protocol-specific default Mail server. This overrides the <code>mail.host</code> property.</td>
<td><code>mail.host</code></td>
</tr>
<tr>
<td><code>mail.protocol.user</code></td>
<td>Specifies the protocol-specific default username for connecting to the Mail server. This overrides the <code>mail.user</code> property.</td>
<td><code>mail.user</code></td>
</tr>
<tr>
<td><code>mail.from</code></td>
<td>Specifies the return address of the current user. Used by the <code>InternetAddress.getLocalAddress</code> method to specify the current user’s email address.</td>
<td><code>username@host</code></td>
</tr>
<tr>
<td><code>mail.debug</code></td>
<td>Specifies the initial debug mode. Setting this property to <code>true</code> will turn on debug mode, while setting it to <code>false</code> turns debug mode off. Note that the <code>Session.setDebug</code> method also controls the debug mode.</td>
<td><code>false</code></td>
</tr>
</tbody>
</table>
Example: Showing a Message

```java
import java.util.*;
import java.io.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

/**
 * Demo app that exercises the Message interfaces.
 * Show information about and contents of messages.
 *
 * msgshow protocol host user password mailbox msgnum
 */

public class msgshow {
    static String protocol;
    static String host = null;
    static String user = null;
    static String password = null;
    static String mbox = "INBOX";
    static String url = null;
    static int port = -1;
    static boolean verbose = false;
    static boolean debug = false;
    static boolean showStructure = false;

    public static void main(String argv[]) {
        int msgnum = -1;
        int optind;

        for (optind = 0; optind < argv.length; optind++) {
            if (argv[optind].equals("-T")) {
                protocol = argv[++optind];
            } else if (argv[optind].equals("-H")) {
                host = argv[++optind];
            } else if (argv[optind].equals("-U")) {
                user = argv[++optind];
            }
        }
    }
}
```
Example: Showing a Message

```java
try {
    if (optind < argv.length)
        msgnum = Integer.parseInt(argv[optind]);

    // Get a Properties object
    Properties props = System.getProperties();

    // Get a Session object
    Session session = Session.getDefaultInstance(props, null);
    session.setDebug(debug);

    // Get a Store object
    Store store = null;
    if (url != null) {
        URLName urln = new URLName(url);
        store = session.getStore(urln);
        store.connect();
    } else {
        if (protocol != null)
            store = session.getStore(protocol);
        else
            store = session.getDefaultStore();

        // Connect
        if (host != null || user != null || password != null)
        }
    }

    try {
        if (optind < argv.length)
            msgnum = Integer.parseInt(argv[optind]);

        // Get a Properties object
        Properties props = System.getProperties();

        // Get a Session object
        Session session = Session.getDefaultInstance(props, null);
        session.setDebug(debug);

        // Get a Store object
        Store store = null;
        if (url != null) {
            URLName urln = new URLName(url);
            store = session.getStore(urln);
            store.connect();
        } else {
            if (protocol != null)
                store = session.getStore(protocol);
            else
                store = session.getDefaultStore();

            // Connect
            if (host != null || user != null || password != null)
```
store.connect(host, port, user, password);
else
store.connect();
}

// Open the Folder
Folder folder = store.getDefaultFolder();
if (folder == null) {
    System.out.println("No default folder");
    System.exit(1);
}

folder = folder.getFolder(mbox);
if (folder == null) {
    System.out.println("Invalid folder");
    System.exit(1);
}

// try to open read/write and if that fails try read-only
try {
    folder.open(Folder.READ_WRITE);
} catch (MessagingException ex) {
    folder.open(Folder.READ_ONLY);
}
int totalMessages = folder.getMessageCount();

if (totalMessages == 0) {
    System.out.println("Empty folder");
    folder.close(false);
    store.close();
    System.exit(1);
}

if (verbose) {
    int newMessages = folder.getNewMessageCount();
    System.out.println("Total messages = " + totalMessages);
    System.out.println("New messages = " + newMessages);
    System.out.println("-------------------------------");
}

if (msgnum == -1) {
    // Attributes & Flags for all messages ..
    Message[] msgs = folder.getMessages();

    // Use a suitable FetchProfile
    FetchProfile fp = new FetchProfile();
    fp.add(FetchProfile.Item.ENVELOPE);
    fp.add(FetchProfile.Item.FLAGS);
    fp.add("X-Mailer");
    folder.fetch(msgs, fp);
    for (int i = 0; i < msgs.length; i++) {

System.out.println("--------------------------");
System.out.println("MESSAGE #" + (i + 1) + ":");
dumpEnvelope(msgs[i]);
}
} else {
    System.out.println("Getting message number: " + msgnum);
    Message m = null;
    try {
        m = folder.getMessage(msgnum);
        dumpPart(m);
    } catch (IndexOutOfBoundsException iex) {
        System.out.println("Message number out of range");
    }
}

folder.close(false);
store.close();
catch (Exception ex) {
    ex.printStackTrace();
}
System.exit(1);

public static void dumpPart(Part p) throws Exception {
    if (p instanceof Message)
        dumpEnvelope((Message)p);

    /** Dump input stream ..
    InputStream is = p.getInputStream();
    // If "is" is not already buffered, wrap a BufferedInputStream
    // around it.
    if (!is instanceof BufferedInputStream)
        is = new BufferedInputStream(is);
    int c;
    while ((c = is.read()) != -1)
        System.out.write(c);
    **/
    pr("CONTENT-TYPE: " + p.getContentType());

    /*
    * Using isMimeType to determine the content type avoids
    * fetching the actual content data until we need it.
    */
    if (p.isMimeType("text/plain")) {
        pr("This is plain text");
        pr("--------------------------");
        if (!showStructure)
            System.out.println((String)p.getContent());
    } else if (p.isMimeType("multipart/*")) {
Appendix B: Examples Using the JavaMail API

Example: Showing a Message

```java
pr("This is a Multipart");
pr("---------------------------");
Multipart mp = (Multipart)p.getContent();
level++;
int count = mp.getCount();
for (int i = 0; i < count; i++)
    dumpPart(mp.getBodyPart(i));
level--;
}
else if (p.isMimeType("message/rfc822")) {
    pr("This is a Nested Message");
    pr("---------------------------");
    level++;
    dumpPart((Part)p.getContent());
    level--;
} else if (!showStructure) {
    /*
     * If we actually want to see the data, and it's not a
     * MIME type we know, fetch it and check its Java type.
     */
    Object o = p.getContent();
    if (o instanceof String) {
        pr("This is a string");
        pr("---------------------------");
        System.out.println((String)o);
    } else if (o instanceof InputStream) {
        pr("This is just an input stream");
        pr("---------------------------");
        InputStream is = (InputStream)o;
        int c;
        while ((c = is.read()) != -1)
            System.out.write(c);
    } else {
        pr("This is an unknown type");
        pr("---------------------------");
        pr(o.toString());
    }
}
else {
    pr("This is an unknown type");
    pr("---------------------------");
}
}

public static void dumpEnvelope(Message m) throws Exception {
    pr("This is the message envelope");
    pr("---------------------------");
    Address[] a;
    // FROM
    if ((a = m.getFrom()) != null) {
        for (int j = 0; j < a.length; j++)
            pr("FROM: " + a[j].toString());
    }
    // TO
```
if ((a = m.getRecipients(Message.RecipientType.TO)) != null) {
    for (int j = 0; j < a.length; j++)
        pr("TO: " + a[j].toString());
}

// SUBJECT
pr("SUBJECT: " + m.getSubject());

// DATE
Date d = m.getSentDate();
pr("SendDate: " +
    (d != null ? d.toString() : "UNKNOWN"));

// FLAGS
Flags flags = m.getFlags();
StringBuffer sb = new StringBuffer();
Flags.Flag[] sf = flags.getSystemFlags(); // get the system flags

boolean first = true;
for (int i = 0; i < sf.length; i++) {
    String s;
    Flags.Flag f = sf[i];
    if (f == Flags.Flag.ANSWERED)
        s = "\Answered";
    else if (f == Flags.Flag.DELETED)
        s = "\Deleted";
    else if (f == Flags.Flag.DRAFT)
        s = "\Draft";
    else if (f == Flags.Flag.FLAGGED)
        s = "\Flagged";
    else if (f == Flags.Flag.RECENT)
        s = "\Recent";
    else if (f == Flags.Flag.SEEN)
        s = "\Seen";
    else
        continue; // skip it
    if (first)
        first = false;
    else
        sb.append(' ');
    sb.append(s);
}

String[] uf = flags.getUserFlags(); // get user-flag strings
for (int i = 0; i < uf.length; i++) {
    if (first)
        first = false;
    else
        sb.append(' ');
    sb.append(uf[i]);
}
pr("FLAGS: " + sb.toString());
Example: Listing Folders

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

/**
 * Demo app that exercises the Message interfaces.
 * List information about folders.
 */

public class folderlist {
    static String protocol = null;
    static String host = null;
    static String user = null;
    static String password = null;
    static String url = null;
    static String root = null;
    static String pattern = "%";
    static boolean recursive = false;
    static boolean verbose = false;
    static boolean debug = false;

    public static void main(String argv[]) throws Exception {
        int optind;
        for (optind = 0; optind < argv.length; optind++) {
            if (argv[optind].equals("-T")) {
                protocol = argv[++optind];
            } else if (argv[optind].equals("-H")) {
                host = argv[++optind];
            }
        }
        ...
Example: Listing Folders

```java
} else if (argv[optind].equals("-U")) {
    user = argv[++optind];
} else if (argv[optind].equals("-P")) {
    password = argv[++optind];
} else if (argv[optind].equals("-L")) {
    url = argv[++optind];
} else if (argv[optind].equals("-R")) {
    root = argv[++optind];
} else if (argv[optind].equals("-r")) {
    recursive = true;
} else if (argv[optind].equals("-v")) {
    verbose = true;
} else if (argv[optind].equals("-D")) {
    debug = true;
} else if (argv[optind].equals("--")) {
    optind++;
    break;
} else if (argv[optind].startsWith("-")) {
    System.out.println(
    System.out.println(" [-R root] [-r] [pattern] /*");
    System.exit(1);
} else {
    break;
}

} else if (optind < argv.length)
    pattern = argv[optind];

// Get a Properties object
Properties props = System.getProperties();

// Get a Session object
Session session = Session.getDefaultInstance(props, null);
session.setDebug(debug);

// Get a Store object
Store store = null;
Folder rf = null;
if (url != null) {
    URLName urln = new URLName(url);
    store = session.getStore(urln);
    store.connect();
} else {
    if (protocol != null)
        store = session.getStore(protocol);
    else
        store = session.getStore();

    // Connect
```
if (host != null || user != null || password != null)
    store.connect(host, user, password);
else
    store.connect();
}

// List namespace
if (root != null)
    rf = store.getFolder(root);
else
    rf = store.getDefaultFolder();
dumpFolder(rf, false, ":");
if (((rf.getType() & Folder.HOLDS_FOLDERS) != 0) { 
    Folder[] f = rf.list(pattern);
    for (int i = 0; i < f.length; i++)
        dumpFolder(f[i], recursive, "    ");
}
store.close();
}

static void dumpFolder(Folder folder, boolean recurse, String tab) throws Exception {
    System.out.println(tab + "Name:      " + folder.getName());
    System.out.println(tab + "Full Name: " + folder.getFullName());
    System.out.println(tab + "URL:       " + folder.getURLName());
    if (verbose) {
        if (!folder.isSubscribed())
            System.out.println(tab + "Not Subscribed");

        if (((folder.getType() & Folder.HOLDS_MESSAGES) != 0) {
            if (folder.hasNewMessages())
                System.out.println(tab + "Has New Messages");
            System.out.println(tab + "Total Messages: " +
                folder.getMessageCount());
            System.out.println(tab + "New Messages: " +
                folder.getNewMessageCount());
            System.out.println(tab + "Unread Messages: " +
                folder.getUnreadMessageCount());
        }
        if ((folder.getType() & Folder.HOLDS_FOLDERS) != 0)
            System.out.println(tab + "Is Directory");
    }
    System.out.println();
}

if (((folder.getType() & Folder.HOLDS_FOLDERS) != 0) { 
    if (recurse) {
        Folder[] f = folder.list();
        for (int i = 0; i < f.length; i++)
            dumpFolder(f[i], recurse, tab + ":");
Example: Search a Folder for a Message

import java.util.*;
import java.io.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.mail.search.*;
import javax.activation.*;

/**
 * Search the given folder for messages matching the given criteria.
 */

public class search {
    static String protocol = "imap";
    static String host = null;
    static String user = null;
    static String password = null;
    static String mbox = "INBOX";
    static String url = null;
    static boolean debug = false;

    public static void main(String argv[]) {
        int optind;

        String subject = null;
        String from = null;
        boolean or = false;
        boolean today = false;

        for (optind = 0; optind < argv.length; optind++) {
            if (argv[optind].equals("-T")) {
                protocol = argv[++optind];
            } else if (argv[optind].equals("-H")) {
                host = argv[++optind];
            } else if (argv[optind].equals("-U")) {
                user = argv[++optind];
            } else if (argv[optind].equals("-P")) {
                password = argv[++optind];
            } else if (argv[optind].equals("-or")) {
                or = true;
            } else if (argv[optind].equals("-D")) {
                debug = true;
            } else if (argv[optind].equals("-f")) {

            }
        }
    }
}
mbox = argv[++optind];
} else if (argv[optind].equals("-L")) {
    url = argv[++optind];
} else if (argv[optind].equals("-subject")) {
    subject = argv[++optind];
} else if (argv[optind].equals("-from")) {
    from = argv[++optind];
} else if (argv[optind].equals("-today")) {
    today = true;
} else if (argv[optind].equals("--")) {
    optind++;
    break;
} else if (argv[optind].startsWith("-")) {
    System.out.println(  
        "[-U user] [-P password] [-f mailbox] " +  
        "[-subject subject] [-from from] [-or] [-today]";
    System.exit(1);
} else {
    break;
}
}

try {

    if ((subject == null) && (from == null) && !today) {
        System.out.println("Specify either -subject, -from or " +  
            "-today");
        System.exit(1);
    }

    // Get a Properties object  
    Properties props = System.getProperties();

    // Get a Session object  
    Session session = Session.getDefaultInstance(props, null);  
    session.setDebug(debug);

    // Get a Store object  
    Store store = null;  
    if (url != null) {
        URLName urln = new URLName(url);  
        store = session.getStore(urln);  
        store.connect();
    } else {
        if (protocol != null)  
            store = session.getStore(protocol);
        else
            store = session.getStore();

        // Connect  
        if (host != null || user != null || password != null)  
            store.connect(host, user, password);
else
    store.connect();
}

// Open the Folder
Folder folder = store.getDefaultFolder();
if (folder == null) {
    System.out.println("No default folder");
    System.exit(1);
}

folder = folder.getFolder(mbox);
if (folder == null) {
    System.out.println("Invalid folder");
    System.exit(1);
}

folder.open(Folder.READ_ONLY);
SearchTerm term = null;

if (subject != null)
    term = new SubjectTerm(subject);
if (from != null) {
    FromStringTerm fromTerm = new FromStringTerm(from);
    if (term != null) {
        if (or)
            term = new OrTerm(term, fromTerm);
        else
            term = new AndTerm(term, fromTerm);
    }
    else
        term = fromTerm;
}
if (today) {
    ReceivedDateTerm dateTerm =
        new ReceivedDateTerm(ComparisonTerm.EQ, new Date());
    if (term != null) {
        if (or)
            term = new OrTerm(term, dateTerm);
        else
            term = new AndTerm(term, dateTerm);
    }
    else
        term = dateTerm;
}

Message[] msgs = folder.search(term);
System.out.println("FOUND " + msgs.length + " MESSAGES");
if (msgs.length == 0) // no match
    System.exit(1);

// Use a suitable FetchProfile
Appendix B: Examples Using the JavaMail API

Example: Search a Folder for a Message

```java
FetchProfile fp = new FetchProfile();
fp.add(FetchProfile.Item.ENVELOPE);
folder.fetch(msgs, fp);

for (int i = 0; i < msgs.length; i++) {
    System.out.println("--------------------------");
    System.out.println("MESSAGE #" + (i + 1) + ":");
    dumpPart(msgs[i]);
}

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

```java
System.exit(1);
```

```java
public static void dumpPart(Part p) throws Exception {
    if (p instanceof Message) {
        Message m = (Message)p;
        Address[] a;
        // FROM
        if ((a = m.getFrom()) != null) {
            for (int j = 0; j < a.length; j++)
                System.out.println("FROM: " + a[j].toString());
        }

        // TO
        if ((a = m.getRecipients(Message.RecipientType.TO)) != null) {
            for (int j = 0; j < a.length; j++)
                System.out.println("TO: " + a[j].toString());
        }

        // SUBJECT
        System.out.println("SUBJECT: " + m.getSubject());

        // DATE
        Date d = m.getSentDate();
        System.out.println("SendDate: " +
            (d != null ? d.toLocaleString() : "UNKNOWN"));

        // FLAGS:
        Flags flags = m.getFlags();
        StringBuffer sb = new StringBuffer();
        Flags.Flag[] sf = flags.getSystemFlags(); // get the system flags
        boolean first = true;
        for (int i = 0; i < sf.length; i++) {
            String s;
            Flags.Flag f = sf[i];
            if (f == Flags.Flag.ANSWERED)
                System.out.println("FLAG: ANSWERED");
        }
    }
}
```
s = "\Answered";
else if (f == Flags.Flag.DELETED)
    s = "\Deleted";
else if (f == Flags.Flag.DRAFT)
    s = "\Draft";
else if (f == Flags.Flag.FLAGGED)
    s = "\Flagged";
else if (f == Flags.Flag.RECENT)
    s = "\Recent";
else if (f == Flags.Flag.SEEN)
    s = "\Seen";
else
    continue; // skip it
if (first)
    first = false;
else
    sb.append(' ');
    sb.append(s);
}

String[] uf = flags.getUserFlags(); // get the user flag strings
for (int i = 0; i < uf.length; i++) {
    if (first)
        first = false;
    else
        sb.append(' ');
        sb.append(uf[i]);
}  
System.out.println("FLAGS = " + sb.toString());

System.out.println("CONTENT-TYPE: " + p.getContentType());
/* Dump input stream
InputStream is = ((MimeMessage)m).getInputStream();
int c;
while ((c = is.read()) != -1)
    System.out.write(c);
*/

Object o = p.getContent();
if (o instanceof String) {
    System.out.println("This is a String");
    System.out.println((String)o);
} else if (o instanceof Multipart) {
    System.out.println("This is a Multipart");
    Multipart mp = (Multipart)o;
    int count = mp.getCount();
    for (int i = 0; i < count; i++)
        dumpPart(mp.getBodyPart(i));
} else if (o instanceof InputStream) {
    System.out.println("This is just an input stream");
    InputStream is = (InputStream)o;
Example: Monitoring a Mailbox

```java
import java.util.*;
import java.io.*;
import javax.mail.*;
import javax.mail.event.*;
import javax.activation.*;

/* Monitors given mailbox for new mail */
public class monitor {
    public static void main(String argv[]) {
        if (argv.length != 5) {
            System.out.println("Usage: monitor <host> <user> <password> <mbox> <freq>\n");
            System.exit(1);
        }
        System.out.println("\nTesting monitor\n");
        try {
            Properties props = System.getProperties();
            // Get a Session object
            Session session = Session.getDefaultInstance(props, null);
            // session.setDebug(true);
            // Get a Store object
            Store store = session.getStore("imap");
            // Connect
            store.connect(argv[0], argv[1], argv[2]);
            // Open a Folder
            Folder folder = store.getFolder(argv[3]);
            if (folder == null || !folder.exists()) {
                System.out.println("Invalid folder");
                System.exit(1);
            }
            folder.open(Folder.READ_WRITE);
```

int c;
while ((c = is.read()) != -1)
    System.out.write(c);
```
Example: Sending a Message

```java
import java.util.*;
import java.io.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;
/**
 * msgmultisendsample creates a simple multipart/mixed message and sends
 */
```
Example: Sending a Message

Migration to new JavaMail API 1.4

Example: Sending a Message

public class msgmultisendsample {
    static String msgText1 = "This is a message body.\nHere’s line two."
    static String msgText2 = "This is the text in the message attachment."

    public static void main(String[] args) {
        if (args.length != 4) {
            System.out.println("usage: java msgmultisend <to> <from> <smtp> true|false");
            return;
        }

        String to = args[0];
        String from = args[1];
        String host = args[2];
        boolean debug = Boolean.valueOf(args[3]).booleanValue();

        // create some properties and get the default Session
        Properties props = new Properties();
        props.put("mail.smtp.host", host);
        Session session = Session.getDefaultInstance(props, null);
        session.setDebug(debug);

        try {
            // create a message
            MimeMessage msg = new MimeMessage(session);
            msg.setFrom(new InternetAddress(from));
            InternetAddress[] address = {new InternetAddress(to)};
            msg.setRecipients(Message.RecipientType.TO, address);
            msg.setSubject("JavaMail APIs Multipart Test");
            msg.setSentDate(new Date());

            // create and fill the first message part
            MimeBodyPart mbp1 = new MimeBodyPart();
            mbp1.setText(msgText1);

            // create and fill the second message part
            MimeBodyPart mbp2 = new MimeBodyPart();
            mbp2.setText(msgText2, "us-ascii");

            // create the Multipart and its parts to it
            Multipart mp = new MimeMultipart();

            mp.addBodyPart(mbp1);
            mp.addBodyPart(mbp2);

            msg.setContent(mp);

            Transport transport = Session.getDefaultInstance(props, null).getTransport("smtp");
            transport.connect(host, from, null);
            transport.sendMessage(msg, msg.getAllRecipients());
            transport.close();
        }
    }
}
mp.addBodyPart(mbpl);
mp.addBodyPart(mbpl);

// add the Multipart to the message
msg.setContentType(mp);

// send the message
Transport.send(msg);

} catch (MessagingException mex) {
mex.printStackTrace();
Exception ex = null;
if ((ex = mex.getNextException()) != null) {
ex.printStackTrace();
}
}
}
Appendix C: Message Security

Overview

This is not a full specification of how Message Security will be integrated into the JavaMail system. This is a description of implementation strategy. The purpose of this section is to declare that it is possible to integrate message security, not to define how it will be integrated. The final design for Message Security will change based on feedback and finalization of the S/MIME IETF specification.

This section discusses encrypting/decrypting messages, and signing/verifying signatures. It will not discuss how Security Restrictions on untrusted or signed applets will work, nor will it discuss a general authentication model for Stores (For example; a GSS API in the Java platform.)

Displaying an Encrypted/Signed Message

Displaying an encrypted or signed message is the same as displaying any other message. The client uses the DataHandler for that encrypted message together with the "view" command. This returns a bean that displays the data. There will be both a multipart/signed and multipart/encrypted viewer bean (can be the same bean). The beans will need to be aware of the MultiPartSigned/MultiPartEncrypted classes.

MultiPartEncrypted/Signed Classes

The JavaMail API will probably add two new content classes: MultiPartEncrypted and MultiPartSigned. They subclass the MultiPart class and handle the MIME types multipart/encrypted and multipart/signed. There are many possible "protocols" that specify how a message has been encrypted and/or signed. The MPE/MPS classes will find all the installed protocols. The ContentType’s protocol parameter determines which protocol class to use. There needs to be a standard registration of protocol objects or a way to search for valid packages and instantiate a particular class. The MultiPart classes will hand off the control information, other parameters, and the data to be manipulated (either the signed or encrypted block) through some defined Protocol interface.
Reading the Contents

There will be times when an applet/application needs to retrieve the content of the message without displaying it. The code sample below shows one possible technique with a message containing encrypted content:

```java
Message msg = // message gotten from some folder.
if (msg.isMimeType("multipart/encrypted")) {
    Object o = msg.getContent();
    if (o instanceof MultiPartEncrypted) {
        MultiPartEncrypted mpe = (MultiPartEncrypted) o;
        mpe.decrypt();
        // use the default keys/certs from the user.
        // We should also be able to determine
        // whether or not to interact with the user
        // should then be able to use the multipart methods to
        // get any contained blocks }
    }
}
```

The `getContent` method returns a `MultiPartEncrypted` object. There will be methods on this class to decrypt the content. The decryption could either determine which keys needed to be used, use the defaults (maybe the current user’s keys) or explicitly pass which keys/certificates to use.

Verifying Signatures

Applications/applets will need to verify the validity of a signature. The code sample below shows how this might be done:

```java
Message msg = // message gotten from some folder
if (msg.isMimeType("multipart/signed")) {
    Object o = msg.getContent();
    if (o instanceof MultiPartSigned) {
        MultiPartSigned mps = (MultiPartSigned) o;
        boolean validsig = mps.verifySignature();
        // could already get the other blocks
        // even if it wasn't a valid signature
    }
}
```

If the signature is invalid, the application can still access the data. There will be methods in `MultiPartSigned` that allow the setting of which keys or certificates to use when verifying the signature.
Creating a Message

There are two methods for creating an Encrypted/Signed message. Users will probably see an editor bean for the content types multipart/signed and multipart/encrypted. These beans would handle the UI components to allow the user to select how they want to encrypt/sign the message. The beans could be integrated into an application’s Message Composition window.

Encrypted/Signed

The non-GUI method of creating the messages involves using the MultiPartEncrypted/Signed classes. The classes can be created and used as the content for a message. The following code shows how might work:

```java
MultiPartEncrypted mpe = new MultiPartEncrypted();
// Can setup parameters for how you want to encrypt the
// message; otherwise, it will use the user's preferences.
// Set the content you wish to encrypt (to encrypt multiple
// contents a multipart/mixed block should be used)
String ourContent = "Please encrypt me!";
mpe.setContentType(ourContent);

MimeMessage m = new MimeMessage(session);
m.setContent(mpe);
```

The message will be encrypted when the message is sent. There will be other methods that allow the setting of which encryption scheme shall be used, and the keys involved.

Creating a Multipart Signed message is very similar to creating a Multipart Encrypted message, except that a Multipart Signed object is created instead.
Appendix C: Message Security
Overview
Appendix D:
Part and Multipart Class Diagram

This appendix illustrates relationships between Part interfaces and Message classes.

FIGURE D-1
Appendix E: 
MimeMessage Object Hierarchy

This appendix illustrates the object hierarchy.

FIGURE E-1

MimeMessage Object
NestedMultipart Message

Legend:

method() method call and
object returned

Message

getFrom()<address of message sender>

getSubject()<message subject>

getContentType()"multipart/mixed"

getContentPane()

Multipart

BodyPart

getContentType()

(String)getContent()<message text>

getContentType()"image/gif"

DataHandler

use the DataHandler
methods to access
the image

getContentPane()

BodyPart

getContentType()"multipart/mixed"

(Multipart)getContent()

Multipart

...repeat with Multipart
as before...

<Text>

.getBodyPart(0)

getContentType()"text/plain"

<Image>

.getBodyPart(1)

getContentType()"image/png"

Multipart

.getBodyPart(2)

getContentType()"image/jpeg"
Appendix E: MimeMessage Object Hierarchy
Appendix F:
Features Added in JavaMail 1.1

This appendix summarizes the features added to JavaMail 1.1. For more information about each item, refer to the appropriate Javadoc documentation.

The MessageContext Class and MessageAware Interface

In some cases it is desirable for the object representing the content of a BodyPart object to know something about the context in which it is operating. For example, the content-object might need to know what other data is contained in the same Multipart object, who sent the message containing the data, and so forth. This allows for more interesting content types that know more about the message containing them and the mail system in general.

Some uses of the multipart/related object might require these capabilities. For instance, the handler for a text/html body part contained in a multipart/related object might need to know about the containing object in order to find the related image data needed to display the HTML document. (Note that JavaMail provides no direct support for multipart/related messages.)

To deal with these issues, the MessageContext class and MessageAware interface have been added in JavaMail 1.1.

The MessageContext class provides the basic information about the context in which a content object is operating. Given a MessageContext object, it is possible to navigate through a message’s body structure. The MessageAware interface is an optional interface, implemented by DataSources that have the capability of providing a suitable MessageContext object. The MimePartDataSource implements the MessageAware interface, making this capability available to all MIME messages.

The getMessageID method

The getMessageID method has been added to the MimeMessage class. This method returns the value of RFC822 Message-ID field.


**Additions to the `InternetAddress` Class**

The `encodedPersonal` protected field has been added to the `javax.mail.internet.InternetAddress` class.

The `toString(Address[], int)` method has also been added to this class.

**Additions to the `MimeUtility` Class**

Two static methods have been added to the `javax.mail.internet.MimeUtility` class:

- `String mimeCharset(String charset)`
- `String getDefaultJavaCharset()`

The `mimeCharset` method returns the MIME name of the given JDK™ charset.

The `getDefaultJavaCharset` method returns the default JDK charset for the platform’s locale.

**New SearchTerms**

The current address related search terms: `AddressTerm`, `FromTerm` and `RecipientTerm`, are limited in that they operate on `Address` objects, not `Strings`. These terms use the `equals` method to compare the addresses, which is not useful for the common case of substring comparisons.

Hence three new `SearchTerms` have been introduced:

- `AddressStringTerm`
- `FromStringTerm`
- `RecipientStringTerm`

These terms operate on `Address` `Strings`, rather than `Address` objects.

These new terms correspond to the capabilities provided by the IMAP protocol. The older terms were not supported by IMAP and thus resulted in client-side searches.
Additions to the Folder Class

Two methods have been added to the javax.mail.Folder class:

- int getMode()
- URLName getURLName()

The getMode method returns the mode in which the Folder object was opened.
The getURLName method returns the URLName value of the folder.

New Service Class

To emphasize the commonality in behavior between the Store and Transport classes, and to simplify maintenance of these classes, a new superclass, javax.mail.Service, has been introduced for the Store and Transport classes.
Appendix F: Features Added in JavaMail 1.1

New Service Class
Appendix G:
Features Added in JavaMail 1.2

This appendix summarizes the features that were added in JavaMail 1.2. Refer to the appropriate Javadoc documentation for additional information about each item.

Additions to the MimeMessage Class

The following have been added to the MimeMessage class:

- To simplify the creation of MimeMessage subclasses:
  - The modified field and the parse(InputStream is) method that were previously private are now protected.
  - The createInternetHeaders(InputStream is) method has also been added to this class.

- When forwarding or saving a message retrieved from a Store, it is sometimes desirable to be able to modify the message first. Since most Stores do not allow their Message objects to be modified, the message must first be copied. To simplify copying a MimeMessage, we introduce a copy constructor, MimeMessage(MimeMessage source), that allows a new MimeMessage to be created and initialized with a copy of another MimeMessage.

- The following convenience methods were added to MimeMessage.
  - setRecipients(Message.RecipientType type, String addresses)
  - addRecipients(Message.RecipientType type, String addresses)
  
  Note that these methods take a String for setting/adding a recipient (instead of javax.mail.Address objects).

- One of the most common errors encountered when constructing new messages is forgetting to call the saveChanges() method before writing out the message or calling the Transport.sendMessage() method. To solve this problem, a saved flag was added to MimeMessage and the writeTo() method was changed accordingly.
Additions to the MimeMultipart Class

To simplify the creation of MimeMultipart subclasses, the following have been added to the MimeMultipart class:

- The parse(InputStream is) method that was previously private is now protected.
- The createInternetHeaders(InputStream is) and createMimeBodyPart(InternetHeaders headers, byte[] content) methods have been added to this class as protected methods.

The getRawInputStream method

In some cases, it is desirable to get the data for a body part before JavaMail attempts to decode it. This is particularly important if the Content-Transfer-Encoding header is incorrect. (For example, some mail software is known to use "7-bit" instead of the MIME-defined "7-bit"). Access to this data is currently provided through the protected getContentStream method. Since simply making this method public has the potential to cause a source incompatibility for any subclasses that declare this method as protected, we instead add a new public method, getRawInputStream(), that calls this protected method to the MimeMessage and MimeBodyPart classes.

Additions to the InternetAddress Class

The following were added to the InternetAddress class:

- To simplify copying of InternetAddress objects, the InternetAddress class now implements the Cloneable interface and will provide a public clone() method.
- AddressStringTerm.match does not return the correct results in some situations because it wants to do the match against the formatted address string in Unicode, not the ASCII version that might include charset encoding information. To do this, it attempts to format the address itself, but its logic does not handle all the rules about formatting an address (such as, when to quote the personal name) so it does this formatting differently than InternetAddress.toString does. When the address contains only ASCII characters, the formatting should be identical. This problem has been remedied by adding a new method, toUnicodeString(), to the InternetAddress class, which returns a properly formatted address (RFC 822 syntax) of Unicode characters.
The InternetAddress class now implements the Serializable interface to support saving javax.mail.search terms (described in “Additions for Serializable javax.mail.search terms”).

---

**The MailDateFormat Class**

The MailDateFormat class is now part of the javax.mail.internet package. It was previously contained in the com.sun.mail.util package. This is a utility class used in formatting and parsing dates in MIME headers. The methods it provides are:

- `StringBuffer format(Date date, StringBuffer dateStrBuf, FieldPosition fieldPosition)`
- `Date parse(String text, ParsePosition pos)`

---

**Additions to Exceptions and Events**

The following exceptions and events have been added in JavaMail 1.2:

- Previously, if a client attempted to open a read-only folder in read-write mode, a MessagingException was thrown. This exception type does not indicate that the anomaly was caused by the lack of write-permissions. A new ReadOnlyFolderException was added to indicate that the problem was caused by a read-only folder.

- When authentication with a server fails, the server often supplies some information in its protocol message that indicates the reason for the failure. To allow a service provider to return this information to the user, we now allow the Service.protocolConnect() method to throw an AuthenticationFailedException in this case. The exception may contain a string message that includes the additional information from the server.

- The FolderNotFoundException constructors were not consistent with other exceptions defined in the API. Two new constructors were added to eliminate these inconsistencies:
  - `FolderNotFoundException(Folder folder)`
  - `FolderNotFoundException(Folder folder, String s)`

- If an error occurs when sending a message, the TransportEvent class saved the message that caused the error, but provided no getMessage method for the listener to retrieve the Message object. The getMessage() method was added to TransportEvent class.
Additions to the Session Class

Two static convenience methods were added to the Session class for retrieving the default Session or a new Session object, which do not require an Authenticator parameter (assumed to be null):

- `Session Session.getDefaultInstance(Properties props)`
- `Session Session.getInstance(Properties props)`

Additions to the MimeUtility Class

The following were added to the MimeUtility class to provide additional support for encoding:

- The UUEncode encoder requires the filename to be inserted into the encoded stream. The public access point to the encoder is through the `MimeUtility.encode()` method, which does not have any parameter that can provide the filename. Hence the uuencoded stream always has "encode.buf" as filename. A new method, that allows the setting of the filename has been added:
  ```java
  encode(OutputStream os, String encoding, String filename)
  ```
- The `getEncoding()` method which was previously added to improve the performance of JavaMail was changed from package private to public.

Additions for serializable `javax.mail.search` terms

The `javax.mail.search` package allows you to programmatically construct a search term. As a convenience, these terms can now be saved in persistent storage and restored in a later session. The simplest way to store these expressions is to use serialization.

Many of the search terms reference other objects that must also be serializable. The most problematic such objects are of the class `Message.RecipientType`. This class uses the java "type-safe enum" idiom, which involves a number of static final instances of the class. Applications are allowed to test for equivalence with these "constants" by using the "==" equality operator. Thus, it’s critical that only a single instance of each constant exist in the Java virtual machine. To ensure that this constraint is met when deserializing an object of this class, we must take advantage of the J2SE 1.2 `readReplace()` method. Since this method is not available on JDK 1.1, objects of this class, and thus search terms that reference them, can not be correctly deserialized on JDK 1.1. This is a limitation of this new capability.
To provide this support, the following classes and all their subclasses now implement the \texttt{Serializable} interface:

- \texttt{javax.mail.search.SearchTerm}
- \texttt{javax.mail.Address}
- \texttt{javax.mail.Flags}
- \texttt{javax.mail.Message.RecipientType}

In addition, to allow comparison between search terms, the \texttt{equals} and \texttt{hashCode} methods on \texttt{SearchTerm} (and all subclasses) now implement "value" equivalence rather than identity equivalence.

### Additions to the \texttt{Store} Class

The following methods have been added to \texttt{javax.mail.Store} to provide namespace information:

- \texttt{Folder[]} \texttt{getPersonalNamespaces()}
  
  A personal namespace is a set of names that is considered within the personal scope of the authenticated user. Typically, only the authenticated user has access to mail folders in their personal namespace. If an \texttt{INBOX} exists for a user, it must appear within the user’s personal namespace. In the typical case, there should be only one personal namespace for each user in each \texttt{Store}.

- \texttt{Folder[]} \texttt{getUserNamespaces(String user)}
  
  The namespaces returned represent the personal namespaces for the user. To access mail folders in the other user’s namespace, the currently authenticated user must be explicitly granted access rights. For example, it is common for a manager to grant to their secretary access rights to their mail folders.

- \texttt{Folder[]} \texttt{getSharedNamespaces()}
  
  A shared namespace is a namespace that consists of mail folders that are intended to be shared amongst users and do not exist within a user’s personal namespace.

### New \texttt{ContentDisposition} Class

The \texttt{ContentDisposition} class contained in \texttt{javax.mail.internet} package has been changed from package private to public.
New performance improvements

To allow us to improve the performance of the MimeMessage and MimeMultipart classes when parsing data from an InputStream, we introduce a new SharedInputStream interface that allows the data in the InputStream to be shared instead of copied, and we use this new interface in key parts of the implementation. The methods defined by the SharedInputStream interface are:

- long getPosition()
- InputStream newStream(long start, long end)

A new protected InputStream (which implements the SharedInputStream interface) data member, contentStream, has been added to the MimeMessage and MimeBodyPart classes.

Additions to the ParameterList class

The ParameterList.toString() method returns its results "unfolded". It would be useful to have the results "folded" in certain situations. A new method, ParameterList.toString(int used), will be added which will return "folded" results. Folding is defined by RFC 822 as the process of splitting a header field into multiple lines. "The general rule is that wherever there may be linear-white-space (NOT simply LWSP-chars), a CRLF immediately followed by AT LEAST one LWSP-char may instead be inserted." Unfolding is the process of returning to a single line representation. "Unfolding is accomplished by regarding CRLF immediately followed by a LWSP-char as equivalent to the LWSP-char."