Wireless Messaging API 2.0 Specification ("Specification")

Status: Final

Specification Lead: Siemens AG ("Specification Lead")

Release: May 17, 2004

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 the Specification Lead 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 Guidelines as set forth in the Terms of Use on the Sun's website. 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.

The Specification Lead hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under the Specification

Lead's intellectual property rights that are essential to practice the Specification, to internally practice the Specification for the purpose of designing and developing your Java applets and applications intended to run on the Java platform or 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 without subsetting or supersetting; (iii) includes a complete implementation of any optional components (as defined by 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 the Specification; (vi) satisfies all testing requirements available from the Specification Lead 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 of the Specification Lead's source code or binary code materials; and (viii) does not include any of the Specification Lead's source code or binary code materials without an appropriate and separate license from the Specification Lead. The Specification contains the proprietary information of the Specification Lead and may only be used in accordance with the license terms set forth herein. This license will terminate immediately without notice from the Specification Lead 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, the Specification Lead or the Specification Lead's licensors is granted hereunder. Sun, Sun Microsystems, the Sun logo, Java, 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" AND IS EXPERIMENTAL AND MAY CONTAIN DEFECTS OR DEFICIENCIES WHICH CANNOT OR WILL NOT BE CORRECTED BY THE SPECIFICATION LEAD. THE SPECIFICATION LEAD 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. THE SPECIFICATION LEAD 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 THE SPECIFICATION LEAD 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 THE SPECIFICATION LEAD AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend the Specification Lead 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

If this Software 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 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 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 evaluation of the Specification ("Feedback"). To the extent that you provide the Specification Lead with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant the Specification Lead 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.

(LFI#111199/Form ID#011801)

Contents

Preface

This book provides information on the messaging API which is included in the JSR 205 Wireless Messaging API 2.0(WMA 2.0) specification.

Who Should Use This Book

This book is intended primarily for those individuals and companies who want to implement WMA.

Before You Read This Book

This book assumes that you have experience programming in the C and Java™ languages. It also assumes that you are familiar with the Mobile Information Device Profile (MIDP), the Connected, Limited Device Configuration (CLDC), and the Connected Device Configuration (CDC).

Familiarity with multimedia processing recommended, but not required.

References

GSM 03.40 v7.4.0 Digital cellular telecommunications system (Phase 2+); Technical realization of the Short Message Service (SMS). ETSI 2000

TS 100 900 v7.2.0 (GSM 03.38) Digital cellular telecommunications system (Phase 2+); Alphabets and language-specific information. ETSI 1999

Mobile Information Device Profile (MIDP) Specification, Version 1.0 , Sun Microsystems, 2000

GSM 03.41, ETSI Digital Cellular Telecommunication Systems (phase 2+); Technical realization of Short Message Service Cell Broadcast (SMSCB) (GSM 03.41)

Wireless Datagram Protocol , Version 14-Jun-2001, *Wireless Application Protocol WAP-259-WDP-20010614-aWAP (WDP)*

TIA/EIA-637-A: Short Message Service for Spread Spectrum Systems (IS637)

Connected Device Configuration (CDC) and the Foundation Profile, a white paper, (Sun Microsystems, Inc., 2002)

J2ME™ CDC Specification, v1.0, (Sun Microsystems, Inc., 2002)

Porting Guide for the Connected Device Configuration, Version 1.0, and the Foundation Profile, Version 1.0; (Sun Microsystems, Inc., 2001)

WAP-209-MMSEncapsulation Version 1 01-June-2001

MMS Conformance Document Version 2.0.0 06-February-2002

RFC #822 Standard for the format of APRA Internet Text Messages August-13-1992

RFC #2045 Multipurpose Internet Mail Extensions (MIME) November-1996

RFC #2387 The MIME Multipart/Related Content-type 1998

*RFC #1738 Uniform Resource Locators (URL)*December-1994

Related Documentation

The Java™ Language Specification by James Gosling, Bill Joy, and Guy L. Steele (Addison-Wesley, 1996), ISBN 0-201-63451-1

The Java™ Virtual Machine Specification (Java Series), Second Edition by Tim Lindholm and Frank Yellin (Addison-Wesley, 1999), ISBN 0-201-43294-3

Terms, Acronyms, and Abbreviations Used in this Book

SMS - Short Message Service

MMS - Multimedia Message Service

URL - Uniform Resource Locator

Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`% su` `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized Command-line variable; replace with a real name or value	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this. To delete a file, type `rm` *filename*.

CHAPTER 1

Overview

Description

The messaging API is based on the Generic Connection Framework (GCF), which is defined in the Connected Limited Device Configuration (CLDC) 1.0 specification. The package `javax.microedition.io` defines the framework and supports input/output and networking functionality in J2ME profiles. It provides a coherent way to access and organize data in a resource-constrained environment.

The design of the messaging functionality is similar to the datagram functionality that is used for UDP in the Generic Connection Framework. Like the datagram functionality, messaging provides the notion of opening a connection based on a string address and that the connection can be opened in either client or server mode. However, there are differences between messages and datagrams, so messaging interfaces do not inherit from datagram. It might also be confusing to use the same interfaces for messages and datagrams.

The interfaces for the messaging API have been defined in the `javax.wireless.messaging` package.

Representation of a message

A message can be thought of as having an address part and a data part. A message is represented by a class that implements the interface defined for messages in the API. This interface provides methods that are common for all messages. In the `javax.wireless.messaging` package, the base interface that is implemented by all messages is named `Message`. It provides methods for addresses and timestamps.

For the data part of the message, the API is designed to handle text, binary and multipart messages. These are represented by three subinterfaces of `Message`: `TextMessage`, `BinaryMessage` and `MultipartMessage`. These subinterfaces provide ways to manipulate the payload of the message as Strings, byte arrays and message parts, respectively.

Other subinterfaces of `Message` can be defined for message payloads which are neither pure text nor pure binary. It is also possible to create further subinterfaces of `TextMessage`, `BinaryMessage` and `MultipartMessage` for possible protocol-specific features.

Sending and receiving messages

As defined by the Generic Connection Framework, the message sending and receiving functionality is implemented by a `Connection` interface, in this case, `MessageConnection`. To make a connection, the application obtains an object implementing the `MessageConnection` from the `Connector` class by providing a URL connection string that identifies the address.

If the application specifies a full destination address that defines a recipient to the `Connector`, it gets a `MessageConnection` that works in a “client” mode. This kind of `Connection` can only be used for sending messages to the address specified when creating it.

The application can create a “server” mode `MessageConnection` by providing a URL connection string that includes only an identifier that specifies the messages intended to be received by this application. Then it can use this `MessageConnection` object for receiving and sending messages.

The format of the URL connection string that identifies the address is specific to the messaging protocol used.

For sending messages, the `MessageConnection` object provides factory methods for creating `Message` objects. For receiving messages, the `MessageConnection` supports an event listener-based receiving

mechanism, in addition to a synchronous blocking `receive()` method. The methods for sending and receiving messages can throw a `SecurityException` if the application does not have the permission to perform these operations.

The generic connection framework includes convenience methods for getting `InputStream` and `OutputStream` handles for connections which are `StreamConnections`. The `MessageConnection` does not support stream based operations. If an application calls the `Connector.open*Stream` methods, it will receive an `IllegalArgumentException`.

Bearer-specific Adapter

The basic `MessageConnection` and `Message` framework provides a general mechanism with establishing a messaging application. The appendices describe the specific adapter requirements for URL connection string formatting and bearer-specific message handling requirements.

- JavaDoc API Documentation
- Appendix A - GSM SMS Adapter
- Appendix B - GSM CBS Adapter
- Appendix C - CDMA IS-637 SMS Adapter
- Appendix D - MMS Adapter

The appendices of this specification include the definition of SMS, CBS and MMS URL connection strings. These connection schemes MAY be reused in other adapter specifications, as long as the specified syntax is not modified and the usage does not overlap with these specified adapters (that is, no platform can be expected to implement two protocols for which the URI scheme would be the same, making it impossible for the platform to distinguish which is desired by the application). Other adapter specifications MAY define new connection schemes, as long as these do not conflict with any other connection scheme in use with the Generic Connection Framework.

The appendices describe how the SMS, CBS and MMS adpaters MUST be implemented to conform to the requirements of their specific wireless network environments and how these adapters supply the functionality defined in the *javax.wireless.messaging package*.

When a GSM SMS message connection is established, the platform MUST use the rules in Appendix A for the syntax of the URL connection string and for treatment of the message contents.

When a GSM CBS message connection is established, the platform MUST use the rules in Appendix B for the syntax of the URL connection string and for treatment of the message contents.

When a CDMA SMS message connection is established, the platform MUST use the rules in Appendix C for the syntax of the URL connection string and for treatment of the message contents.

When a MMS message connection is established, the platform MUST use the rules in Appendix D for the syntax of the URL connection string and for treatment of the message contents.

Security

To send and receive messages using this API, applications MUST be granted a permission to perform the requested operation. The mechanisms for granting a permission are implementation dependent.

The permissions for sending and receiving MAY depend on the type of messages and addresses being used. An implementation MAY restrict an application’s ability to send some types of messages and/or sending messages to certain recipient addresses. These addresses can include device addresses and/or identifiers, such as port numbers, within a device.

An implementation MAY restrict certain types of messages or connection addresses, such that the permission would never be available to an application on that device.

The applications MUST NOT assume that successfully sending one message implies that they have the permission to send all kinds of messages to all addresses.

An application should handle `SecurityException`s when a connection handle is provided from `Connector.open(url)` and for any message `receive()` or `send()` operation that potentially engages with the network or the privileged message storage on the device.

Permissions for MIDP 1.0 Platform

When the JSR 205 interfaces are deployed on a MIDP 1.0 device, there is no formal mechanism to identify how a permission to use a specific feature can be granted to a running application. On some systems, the decision to permit a particular operation is left in the hands of the end user. If the user decides to deny the required permission, then a `SecurityException` can be thrown from the `Connector.open()`, the `MessageConnection.send()`, or the `MessageConnection.receive()` method.

Permissions for MIDP 2.0 Platform

When the JSR 205 interfaces are deployed on a MIDP 2.0 device, permissions must be granted to open a connection and to send and receive messages. Separate permissions are provided for the SMS and CBS protocols.

To open a connection, a MIDlet suite must have the appropriate permission to access the `MessageConnection` implementation. If the permission is not granted, then `Connector.open` must throw a `SecurityException`. To send and receive messages, the MIDlet suite can restrict certain types of messages or connection addresses. If the application attempts to send or receive either a restricted type of message or a message with a restricted connection address, then a `SecurityException` must be thrown.

For more information on the permissions that are provided by WMA 2.0, see Appendix E "*Deploying JSR 205 Interfaces on a MIDP 2.0 Platform*".

How to Use the Messaging API

This section provides some examples of how the messaging API can be used.

Sending a text message to an end user

The following sample code sends the string "Hello World!" to an end user as a normal SMS message.

```
try {
  String addr = "sms://+358401234567";
  MessageConnection conn = (MessageConnection) Connector.open(addr);
  TextMessage msg =
    (TextMessage)conn.newMessage(MessageConnection.TEXT_MESSAGE);
  msg.setPayloadText("Hello World!");
  conn.send(msg);
} catch (Exception e) {
...
}
```

A server that responds to received messages

The following sample code illustrates a server application that waits for messages sent to port `5432` and responds to them.

```
try {
  String addr = “sms://:5432”;
  MessageConnection conn = (MessageConnection) Connector.open(addr);
  Message msg = null;

  while (someExitCondition) {
  // wait for incoming messages

    msg = conn.receive();
    // received a message
    if (msg instanceof TextMessage) {
      TextMessage tmsg = (TextMessage)msg;

      String receivedText = tmsg.getPayloadText();
      // respond with the same text with “Received:”
      // inserted in the beginning
      tmsg.setPayloadText(“Received:” + receivedText);
      // Note that the recipient address in the message is
      // already correct as we are reusing the same object

      conn.send(tmsg);
    } else {
      // Received message was not a text message, but e.g. binary

      ...
    }
  }
} catch (Exception e) {
      ...
}
```

Package Summary

Messaging Interfaces

javax.wireless.messaging	This package defines an API which allows applications to send and receive wireless messages.

Networking Package

javax.microedition.io	This pacakge includes the platform networking interfaces which have been modified for use on platforms that support message connections.

CHAPTER 2

Package javax.microedition.io

Description

This pacakge includes the platform networking interfaces which have been modified for use on platforms that support message connections.

This package includes the `Connector` class from MIDP 2.0. This class includes `SecurityException` as an expected return from calls to `open()` which may require explicit authorization to connect.

When the message connection is implemented on a MIDP 1.0 platform, the `SecurityException` can be provided by a platform-dependent authorization mechanism. For example, the user *might* be prompted to ask if the application can send a message and the user's denial interpretted as a `SecurityException`.

Since: MIDP2.0

Class Summary	
Interfaces	
Classes	
`Connector`	Factory class for creating new Connection objects.
Exceptions	

javax.microedition.io
Connector

Declaration

```
public class Connector
```

```
java.lang.Object
  |
  +--javax.microedition.io.Connector
```

Description

Factory class for creating new Connection objects.

The creation of Connections is performed dynamically by looking up a protocol implementation class whose name is formed from the platform name (read from a system property) and the protocol name of the requested connection (extracted from the parameter string supplied by the application programmer.) The parameter string that describes the target should conform to the URL format as described in RFC 2396. This takes the general form:

`{scheme}:[{target}][{parms}]`

where `{scheme}` is the name of a protocol such as *http*.

The `{target}` is normally some kind of network address.

Any `{parms}` are formed as a series of equates of the form ";x=y". Example: ";type=a".

An optional second parameter may be specified to the open function. This is a mode flag that indicates to the protocol handler the intentions of the calling code. The options here specify if the connection is going to be read (READ), written (WRITE), or both (READ_WRITE). The validity of these flag settings is protocol dependent. For instance, a connection for a printer would not allow read access, and would throw an IllegalArgumentException. If the mode parameter is not specified, READ_WRITE is used by default.

An optional third parameter is a boolean flag that indicates if the calling code can handle timeout exceptions. If this flag is set, the protocol implementation may throw an InterruptedIOException when it detects a timeout condition. This flag is only a hint to the protocol handler, and it does not guarantee that such exceptions will actually be thrown. If this parameter is not set, no timeout exceptions will be thrown.

Because connections are frequently opened just to gain access to a specific input or output stream, four convenience functions are provided for this purpose. See also: `DatagramConnection` for information relating to datagram addressing

Since: CLDC 1.0

Member Summary

Fields

```
            static int  READ
            static int  READ_WRITE
            static int  WRITE
```

Methods

```
      static Connection  open(java.lang.String name)
```

Member Summary	
static Connection	open(java.lang.String name, int mode)
static Connection	open(java.lang.String name, int mode, boolean timeouts)
static java.io.DataInputStream	openDataInputStream(java.lang.String name)
static java.io.DataOutputStream	openDataOutputStream(java.lang.String name)
static java.io.InputStream	openInputStream(java.lang.String name)
static java.io.OutputStream	openOutputStream(java.lang.String name)

Inherited Member Summary
Methods inherited from class `Object`
`equals(Object), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(), wait()`

Fields

READ

Declaration:
`public static final int READ`

Description:
Access mode READ.

The value `1` is assigned to `READ`.

READ_WRITE

Declaration:
`public static final int READ_WRITE`

Description:
Access mode READ_WRITE.

The value `3` is assigned to `READ_WRITE`.

WRITE

Declaration:
`public static final int WRITE`

Description:
Access mode WRITE.

The value `2` is assigned to `WRITE`.

Methods

open(String)

Declaration:
```
public static javax.microedition.io.Connection open(java.lang.String name)
            throws IOException
```

Description:
Create and open a Connection.

Parameters:
`name` - The URL for the connection.

Returns: A new Connection object.

Throws:
`java.lang.IllegalArgumentException` - If a parameter is invalid.

`ConnectionNotFoundException` - If the requested connection cannot be made, or the protocol type does not exist.

`java.io.IOException` - If some other kind of I/O error occurs.

SecurityException - If a requested protocol handler is not permitted.

open(String, int)

Declaration:
```
public static javax.microedition.io.Connection open(java.lang.String name, int mode)
            throws IOException
```

Description:
Create and open a Connection.

Parameters:
`name` - The URL for the connection.

`mode` - The access mode.

Returns: A new Connection object.

Throws:
`java.lang.IllegalArgumentException` - If a parameter is invalid.

`ConnectionNotFoundException` - If the requested connection cannot be made, or the protocol type does not exist.

`java.io.IOException` - If some other kind of I/O error occurs.

SecurityException - If a requested protocol handler is not permitted.

open(String, int, boolean)

Declaration:
```
public static javax.microedition.io.Connection open(java.lang.String name, int mode,
            boolean timeouts)
            throws IOException
```

Description:
Create and open a Connection.

Parameters:

`name` - The URL for the connection

`mode` - The access mode

`timeouts` - A flag to indicate that the caller wants timeout exceptions

Returns: A new Connection object

Throws:

`java.lang.IllegalArgumentException` - If a parameter is invalid.

`ConnectionNotFoundException` - if the requested connection cannot be made, or the protocol type does not exist.

`java.io.IOException` - If some other kind of I/O error occurs.

SecurityException - If a requested protocol handler is not permitted.

openDataInputStream(String)

Declaration:

```
public static java.io.DataInputStream openDataInputStream(java.lang.String name)
            throws IOException
```

Description:

Create and open a connection input stream.

Parameters:

`name` - The URL for the connection.

Returns: A DataInputStream.

Throws:

`java.lang.IllegalArgumentException` - If a parameter is invalid.

`ConnectionNotFoundException` - If the connection cannot be found.

`java.io.IOException` - If some other kind of I/O error occurs.

SecurityException - If access to the requested stream is not permitted.

openDataOutputStream(String)

Declaration:

```
public static java.io.DataOutputStream openDataOutputStream(java.lang.String name)
            throws IOException
```

Description:

Create and open a connection output stream.

Parameters:

`name` - The URL for the connection.

Returns: A DataOutputStream.

Throws:

`java.lang.IllegalArgumentException` - If a parameter is invalid.

`ConnectionNotFoundException` - If the connection cannot be found.

`java.io.IOException` - If some other kind of I/O error occurs.

SecurityException - If access to the requested stream is not permitted.

openInputStream(String)

Declaration:

```
public static java.io.InputStream openInputStream(java.lang.String name)
            throws IOException
```

Description:
Create and open a connection input stream.

Parameters:
`name` - The URL for the connection.

Returns: An InputStream.

Throws:
`java.lang.IllegalArgumentException` - If a parameter is invalid.

`ConnectionNotFoundException` - If the connection cannot be found.

`java.io.IOException` - If some other kind of I/O error occurs.

SecurityException - If access to the requested stream is not permitted.

openOutputStream(String)

Declaration:

```
public static java.io.OutputStream openOutputStream(java.lang.String name)
            throws IOException
```

Description:
Create and open a connection output stream.

Parameters:
`name` - The URL for the connection.

Returns: An OutputStream.

Throws:
`java.lang.IllegalArgumentException` - If a parameter is invalid.

`ConnectionNotFoundException` - If the connection cannot be found.

`java.io.IOException` - If some other kind of I/O error occurs.

SecurityException - If access to the requested stream is not permitted.

CHAPTER 3

Package javax.wireless.messaging

Description

This package defines an API which allows applications to send and receive wireless messages. The API is generic and independent of the underlying messaging protocol. The underlying protocol can be, for example, GSM Short Message Service, CDMA SMS, MMS, and so on.

Overview

This package is designed to work with `Message` objects that may contain different elements depending on the underlying messaging protocol. This is different from `Datagrams` that are assumed always to be blocks of binary data.

An adapter specification for a given messaging protocol may define further interfaces derived from the `Message` interfaces included in this generic specification.

Unlike network layer datagrams, the wireless messaging protocols that are accessed by using this API are typically of store-and-forward nature. Messages will usually reach the recipient, even if the recipient is not connected at the time of sending. This may happen significantly later if the recipient is disconnected for a long period of time. Sending and possibly also receiving these wireless messages typically involves a financial cost to the end user that cannot be neglected. Therefore, applications should not send unnecessary messages.

The MessageConnection and Message Interfaces

The `MessageConnection` interface represents a `Connection` that can be used for sending and receiving messages. The application opens a `MessageConnection` with the Generic Connection Framework by providing a URL connection string.

The `MessageConnection` can be opened either in “server” or in “client” mode. A “server” mode connection is opened by providing a URL that specifies an identifier for an application on the local device for incoming messages. A port number is an example of an identifier. Messages received with this identifier will then be delivered to the application by using this connection. A “server” mode connection can be used both for sending and for receiving messages.

A “client” mode connection is opened by providing a URL that points to another device. A “client” mode connection can only be used for sending messages.

The messages are represented by the `Message` interface and interfaces derived from it. The `Message` interface has the very basic functions that are common to all messages. Derived interfaces represent messages of different types and provide methods for accessing type-specific features. The kinds of derived interfaces that are supported depends on the underlying messaging protocol. If necessary, interfaces derived from `Message` can be defined in the adapter definitions for mapping the API to an underlying protocol.

The mechanism to derive new interfaces from the `Message` is intended as an extensibility mechanism allowing new protocols to be supported in platforms. Applications are not expected to create their own classes that implement the `Message` interface. The only correct way for applications to create object instances implementing the `Message` interface is to use the `MessageConnection.newMessage` factory method.

Since: WMA 1.0

Class Summary

Interfaces

`BinaryMessage`	An interface representing a binary message.
`Message`	This is the base interface for derived interfaces that represent various types of messages.
`MessageConnection`	The `MessageConnection` interface defines the basic functionality for sending and receiving messages.
`MessageListener`	The `MessageListener` interface provides a mechanism for the application to be notified of incoming messages.
`MultipartMessage`	An interface representing a multipart message.
`TextMessage`	An interface representing a text message.

Classes

`MessagePart`	Instances of the `MessagePart` class can be added to a `MultipartMessage`.

Exceptions

`SizeExceededException`	Inidicates, that an operation is not executable due to insufficient system resources.

javax.wireless.messaging
BinaryMessage

Declaration
```
public interface BinaryMessage extends Message
```

All Superinterfaces: Message

Description
An interface representing a binary message. This is a subinterface of `Message` which contains methods to get and set the binary data payload. The `setPayloadData()` method sets the value of the payload in the data container without any checking whether the value is valid in any way. Methods for manipulating the address portion of the message are inherited from `Message`.

Object instances implementing this interface are just containers for the data that is passed in.

Member Summary	
Methods	
byte[]	getPayloadData()
void	setPayloadData(byte[] data)

Inherited Member Summary
Methods inherited from interface Message
getAddress(), getTimestamp(), setAddress(String)

Methods

getPayloadData()

Declaration:
```
public byte[] getPayloadData()
```

Description:
Returns the message payload data as an array of bytes.

Returns `null`, if the payload for the message is not set.

The returned byte array is a reference to the byte array of this message and the same reference is returned for all calls to this method made before the next call to `setPayloadData`.

Returns: the payload data of this message or `null` if the data has not been set

See Also: setPayloadData(byte[])

setPayloadData(byte[])

Declaration:
```
public void setPayloadData(byte[] data)
```

Description:
Sets the payload data of this message. The payload may be set to `null`.

Setting the payload using this method only sets the reference to the byte array. Changes made to the contents of the byte array subsequently affect the contents of this `BinaryMessage` object. Therefore, applications should not reuse this byte array before the message is sent and the `MessageConnection.send` method returns.

Parameters:
`data` - payload data as a byte array

See Also: `getPayloadData()`

javax.wireless.messaging
Message

Declaration
`public interface` **`Message`**

All Known Subinterfaces: `BinaryMessage`, `MultipartMessage`, `TextMessage`

Description
This is the base interface for derived interfaces that represent various types of messages. This package is designed to work with `Message` objects that may contain different elements depending on the underlying messaging protocol. This is different from `Datagram`s that are assumed always to be just blocks of binary data. An adapter specification for a given messaging protocol may define further interfaces derived from the `Message` interfaces included in this generic specification.

The wireless messaging protocols that are accessed via this API are typically of store-and-forward nature, unlike network layer datagrams. Thus, the messages will usually reach the recipient, even if the recipient is not connected at the time of sending the message. This may happen significantly later if the recipient is disconnected for a long time. Sending, and possibly also receiving, these wireless messages typically involves a financial cost to the end user that cannot be neglected. Therefore, applications should not send many messages unnecessarily.

This interface contains the functionality common to all messages. Concrete object instances representing a message will typically implement other (sub)interfaces providing access to the content and other information in the message which is dependent on the type of the message.

Object instances implementing this interface are just containers for the data that is passed in. The `setAddress()` method just sets the value of the address in the data container without any checking whether the value is valid in any way.

Member Summary	
Methods	
`java.lang.String`	`getAddress()`
`java.util.Date`	`getTimestamp()`
`void`	`setAddress(java.lang.String addr)`

Methods

getAddress()

Declaration:
`public java.lang.String` **`getAddress`**`()`

Description:
Returns the address associated with this message.

If this is a message to be sent, then this address is the recipient's address.

If this is a message that has been received, then this address is the sender's address.

Returns `null`, if the address for the message is not set.

Note: This design allows responses to be sent to a received message by reusing the same `Message` object and just replacing the payload. The address field can normally be kept untouched (unless the messaging protocol requires some special handling of the address).

The returned address uses the same URL string syntax that `Connector.open()` uses to obtain this `MessageConnection`.

Returns: the address of this message, or `null` if the address is not set

See Also: `setAddress(String)`

getTimestamp()

Declaration:
```
public java.util.Date getTimestamp()
```

Description:
Returns the timestamp indicating when this message has been sent.

Returns: `Date` indicating the timestamp in the message or `null` if the timestamp is not set or if the time information is not available in the underlying protocol message

setAddress(String)

Declaration:
```
public void setAddress(java.lang.String addr)
```

Description:
Sets the address associated with this message, that is, the address returned by the `getAddress` method. The address may be set to `null`.

The address MUST use the same URL string syntax that `Connector.open()` uses to obtain this `MessageConnection`.

Parameters:
`addr` - address for the message

See Also: `getAddress()`

javax.wireless.messaging
MessageConnection

Declaration
```
public interface MessageConnection extends javax.microedition.io.Connection
```

All Superinterfaces: `javax.microedition.io.Connection`

Description
The `MessageConnection` interface defines the basic functionality for sending and receiving messages. It contains methods for sending and receiving messages, factory methods to create a new `Message` object, and a method that calculates the number of segments of the underlying protocol that are needed to send a specified `Message` object.

This class is instantiated by a call to `Connector.open()`. An application SHOULD call `close()` when it is finished with the connection. An `IOException` is thrown when any method (except `close`), which is declared to throw an `IOException`, is called on the `MessageConnection` after the connection has been closed.

Messages are sent on a connection. A connection can be defined as *server* mode or *client* mode.

In a *client* mode connection, messages can only be sent. A client mode connection is created by passing a string identifying a destination address to the `Connector.open()` method. This method returns a `MessageConnection` object.

In a *server* mode connection, messages can be sent or received. A server mode connection is created by passing a string that identifies an end point (protocol dependent identifier, for example, a port number) on the local host to the `Connector.open()` method. If the requested end point identifier is already reserved, either by some system application or by another Java application, `Connector.open()` throws an `IOException`. Java applications can open `MessageConnection`s for any unreserved end point identifier, although security permissions might not allow it to send or receive messages using that end point identifier.

The *scheme* that identifies which protocol is used is specific to the given protocol. This interface does not assume any specific protocol and is intended for all wireless messaging protocols.

An application can have several `MessageConnection` instances open simultaneously; these connections can be both client and server mode.

The application can create a class that implements the `MessageListener` interface and register an instance of that class with the `MessageConnection`(s) to be notified of incoming messages. With this technique, a thread does not have to be blocked, waiting to receive messages.

Member Summary

Fields

static java.lang.String	BINARY_MESSAGE
static java.lang.String	MULTIPART_MESSAGE
static java.lang.String	TEXT_MESSAGE

Member Summary	
Methods	
`Message`	`newMessage(java.lang.String type)`
`Message`	`newMessage(java.lang.String type, java.lang.String address)`
`int`	`numberOfSegments(Message msg)`
`Message`	`receive()`
`void`	`send(Message msg)`
`void`	`setMessageListener(MessageListener l)`

Inherited Member Summary
Methods inherited from interface `Connection`
`close()`

Fields

BINARY_MESSAGE

Declaration:
```
public static final java.lang.String BINARY_MESSAGE
```

Description:
Constant for a message type for **binary** messages (value = "binary"). If this constant is used for the `type` parameter in the `newMessage()` methods, then the newly created `Message` will be an instance implementing the `BinaryMessage` interface.

MULTIPART_MESSAGE

Declaration:
```
public static final java.lang.String MULTIPART_MESSAGE
```

Description:
Constant for a message type for **multipart MIME** messages (value = "multipart"). Using this constant as the type parameter in the `newMessage()` methods will cause the newly created `Message` to be an instance implementing the `MultipartMessage` interface.

Since: WMA 2.0

TEXT_MESSAGE

Declaration:
```
public static final java.lang.String TEXT_MESSAGE
```

Description:
Constant for a message type for **text** messages (value = "text"). If this constant is used for the `type` parameter in the `newMessage()` methods, then the newly created `Message` will be an instance implementing the `TextMessage` interface.

Methods

newMessage(String)

Declaration:
```
public javax.wireless.messaging.Message newMessage(java.lang.String type)
```

Description:
Constructs a new message object of a given type. When the type `TEXT_MESSAGE` is passed in, the created object implements the `TextMessage` interface. When type `BINARY_MESSAGE` constant is passed in, the created object implements the `BinaryMessage` interface. When type `MULTIPART_MESSAGE` is passed in, the created object implements the `MultipartMessage` interface. Adapter definitions for messaging protocols can define new constants and new subinterfaces for the `Message`s. The type strings are case-sensitive. The parameter is compared with the `String.equals()` method and does not need to be instance equivalent with the constants specified in this class.

For adapter definitions that are not defined within the JCP process, the strings used MUST begin with an inverted domain name controlled by the defining organization, as is used for Java package names. Strings that do not contain a full stop character "." are reserved for specifications done within the JCP process and MUST NOT be used by other organizations defining adapter specification.

When this method is called from a *client* mode connection, the newly created `Message` has the destination address set to the address identified when this `Connection` was created.

When this method is called from a *server* mode connection, the newly created `Message` does not have the destination address set. It must be set by the application before trying to send the message.

If the connection has been closed, this method still returns a `Message` instance.

Parameters:
`type` - the type of message to be created. There are constants for basic types defined in this interface.

Returns: `Message` object for a given type of message

Throws:
`java.lang.IllegalArgumentException` - if the type parameters is not equal to the value of `TEXT_MESSAGE`, `BINARY_MESSAGE`, `MULTIPART_MESSAGE` or any other type value specified in a private or publicly standardized adapter specification that is supported by the implementation

newMessage(String, String)

Declaration:
```
public javax.wireless.messaging.Message newMessage(java.lang.String type, java.lang.
            String address)
```

Description:
Constructs a new `Message` object of a given type and initializes it with the given destination address. The semantics related to the parameter `type` are the same as for the method signature with just the `type` parameter.

If the connection has been closed, this method still returns a `Message` instance.

Parameters:
`type` - the type of message to be created. There are constants for basic types defined in this interface.

`address` - destination address for the new message

Returns: `Message` object for a given type of message

Throws:

`java.lang.IllegalArgumentException` - if the type parameters is not equal to the value of `TEXT_MESSAGE`, `BINARY_MESSAGE`, `MULTIPART_MESSAGE` or any other type value specified in a private or publicly standardized adapter specification that is supported by the implementation

See Also: `newMessage(String)`

numberOfSegments(Message)

Declaration:

```
public int numberOfSegments(javax.wireless.messaging.Message msg)
```

Description:

Returns the number of segments in the underlying protocol that would be needed for sending the specified `Message`.

Note that this method does not actually send the message. It will only calculate the number of protocol segments needed for sending the message.

This method will calculate the number of segments needed when this message is split into the protocol segments using the appropriate features of the underlying protocol. This method does not take into account possible limitations of the implementation that may limit the number of segments that can be sent using this feature. These limitations are protocol-specific and are documented with the adapter definition for that protocol.

If the connection has been closed, this method returns a count of the message segments that would be sent for the provided `Message`.

Parameters:

`msg` - the message to be used for the calculation

Returns: number of protocol segments needed for sending the message. Returns `0` if the `Message` object cannot be sent using the underlying protocol.

receive()

Declaration:

```
public javax.wireless.messaging.Message receive()
            throws IOException, InterruptedIOException
```

Description:

Receives a message.

If there are no `Message`s for this `MessageConnection` waiting, this method will block until either a message for this `Connection` is received or the `MessageConnection` is closed.

Returns: a `Message` object representing the information in the received message

Throws:

`java.io.IOException` - if any of these situations occur:

- there is an error while receiving a message
- this method is called while the connection is closed
- this method is called on a client mode `MessageConnection`

`java.io.InterruptedIOException` - if this `MessageConnection` object is closed during this receive method call

java.lang.SecurityException - if the application does not have permission to receive messages using the given port number

See Also: `send(Message)`

send(Message)

Declaration:

```
public void send(javax.wireless.messaging.Message msg)
            throws IOException, InterruptedIOException
```

Description:
Sends a message.

Parameters:

`msg` - the message to be sent

Throws:

`java.io.IOException` - if the message could not be sent due to a network failure or if the connection is closed

`java.lang.IllegalArgumentException` - if the message is incomplete or contains invalid information. This exception is also thrown if the payload of the message exceeds the maximum length for the given messaging protocol. One specific case when the message is considered to contain invalid information is if the `Message` is not of the right type to be sent using this `MessageConnection`; the `Message` should be created using the `newMessage()` method of the same `MessageConnection` as will be used for sending it to ensure that it is of the right type.

`java.io.InterruptedIOException` - if a timeout occurs while either trying to send the message or if this `Connection` object is closed during this `send` operation

java.lang.NullPointerException - if the parameter is null

java.lang.SecurityException - if the application does not have permission to send the message

See Also: `receive()`

setMessageListener(MessageListener)

Declaration:

```
public void setMessageListener(javax.wireless.messaging.MessageListener l)
            throws IOException
```

Description:
Registers a `MessageListener` object that the platform can notify when a message has been received on this `MessageConnection`.

If there are incoming messages in the queue of this `MessageConnection` that have not been retrieved by the application prior to calling this method, the newly registered listener object will be notified immediately once for each such incoming message in the queue.

There can be at most one listener object registered for a `MessageConnection` object at any given point in time. Setting a new listener will de-register any previously set listener.

Passing `null` as the parameter will de-register any currently registered listener.

Parameters:

`l` - `MessageListener` object to be registered. If `null`, any currently registered listener will be de-registered and will not receive notifications.

setMessageListener(MessageListener)

Throws:

java.lang.SecurityException - if the application does not have permission to receive messages using the given port number

`java.io.IOException` - if the connection has been closed, or if an attempt is made to register a listener on a client connection

javax.wireless.messaging
MessageListener

Declaration
`public interface` **`MessageListener`**

Description
The `MessageListener` interface provides a mechanism for the application to be notified of incoming messages.

When an incoming message arrives, the `notifyIncomingMessage()` method is called. The application MUST retrieve the message using the `receive()` method of the `MessageConnection`. `MessageListener` should not call `receive()` directly. Instead, it can start a new thread which will receive the message or call another method of the application (which is outside of the listener) that will call `receive()`. For an example of how to use MessageListener, see A Sample MessageListener Implementation.

The listener mechanism allows applications to receive incoming messages without needing to have a thread blocked in the `receive()` method call.

If multiple messages arrive very closely together in time, the implementation has the option of calling this listener from multiple threads in parallel. Applications MUST be prepared to handle this and implement any necessary synchronization as part of the application code, while obeying the requirements set for the listener method.

A Sample MessageListener Implementation

The following sample code illustrates how lightweight and resource-friendly a `MessageListener` can be. In the sample, a separate thread is spawned to handle message reading. The MIDlet life cycle is respected by releasing connections and signaling threads to terminate when the MIDlet is paused or destroyed.

setMessageListener(MessageListener)

```
// Sample message listener program.
import java.io.IOException;
import javax.microedition.midlet.*;
import javax.microedition.io.*;
import javax.wireless.messaging.*;
public class Example extends MIDlet implements MessageListener
{
     MessageConnection messconn;
     boolean done;
     Reader reader;
     // Initial tests setup and execution.
     public void startApp()
     {
        try
         {
            // Get our receiving port connection.
            messconn = (MessageConnection)
        Connector.open("sms://:6222");
            // Register a listener for inbound messages.
            messconn.setMessageListener(this);
            // Start a message-reading thread.
            done = false;
            reader = new Reader();
            new Thread(reader).start();
        } catch (IOException e)
         {
            // Handle startup errors
        }
     }
     // Asynchronous callback for inbound message.

     public void notifyIncomingMessage(MessageConnection conn)
     {
        if (conn == messconn)
         {
            reader.handleMessage();
        }
     }

     // Required MIDlet method - release the connection and
     // signal the reader thread to terminate.
     public void pauseApp()
     {
        done = true;
        try
         {
            messconn.close();
        } catch (IOException e)
         {
            // Handle errors
        }
     }

     // Required MIDlet method - shutdown.
     // @param unconditional forced shutdown flag
     public void destroyApp(boolean unconditional)
     {
        done = true;
        try
         {
            messconn.setMessageListener(null);
            messconn.close();
        } catch (IOException e)
         {
            // Handle shutdown errors.
        }
     }
}
```

```
// Isolate blocking I/O on a separate thread, so callback
// can return immediately.
class Reader implements Runnable
{
    private int pendingMessages = 0;

    // The run method performs the actual message reading.
    public void run()
    {
        while (!done)
        {
            synchronized(this)
            {
           if (pendingMessages == 0)
                {
               try
                    {
               wait();
           } catch (Exception e)
                    {
               // Handle interruption
           }
           }
           pendingMessages--;
            }

            // The benefit of the MessageListener is here.
            // This thread could via similar triggers be
            // handling other kind of events as well in
            // addition to just receiving the messages.

            try
            {
                Message mess = messconn.receive();
       } catch (IOException ioe)
            {
           // Handle reading errors
       }
       }
   }

    public synchronized void handleMessage()
    {
        pendingMessages++;
        notify();
    }

}
```

Member Summary	
Methods	
	void notifyIncomingMessage(MessageConnection conn)

Methods

notifyIncomingMessage(MessageConnection)

Declaration:

```
public void notifyIncomingMessage(javax.wireless.messaging.MessageConnection conn)
```

notifyIncomingMessage(MessageConnection)

Description:

Called by the platform when an incoming message arrives to a `MessageConnection` where the application has registered this listener object.

This method is called once for each incoming message to the `MessageConnection`.

NOTE: The implementation of this method MUST return quickly and MUST NOT perform any extensive operations. The application SHOULD NOT receive and handle the message during this method call. Instead, it should act only as a trigger to start the activity in the application's own thread.

Parameters:

`conn` - the `MessageConnection` where the incoming message has arrived

javax.wireless.messaging
MessagePart

Declaration
```
public class MessagePart

java.lang.Object
  |
  +--javax.wireless.messaging.MessagePart
```

Description
Instances of the `MessagePart` class can be added to a `MultipartMessage`. Each `MessagePart` consists of the content element, MIME type and content-id. The Content can be of any type. Additionally it's possible to specify the content location and the encoding scheme.

Since: WMA 2.0

Member Summary

Constructors

	`MessagePart(byte[] contents, int offset, int length, java.lang.String mimeType, java.lang.String contentId, java.lang.String contentLocation, java.lang.String enc)`
	`MessagePart(byte[] contents, java.lang.String mimeType, java.lang.String contentId, java.lang.String contentLocation, java.lang.String enc)`
	`MessagePart(java.io.InputStream is, java.lang.String mimeType, java.lang.String contentId, java.lang.String contentLocation, java.lang.String enc)`

Methods

`byte[]`	`getContent()`
`java.io.InputStream`	`getContentAsStream()`
`java.lang.String`	`getContentID()`
`java.lang.String`	`getContentLocation()`
`java.lang.String`	`getEncoding()`
`int`	`getLength()`
`java.lang.String`	`getMIMEType()`

Inherited Member Summary

Methods inherited from class `Object`

`equals(Object), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(), wait()`

Constructors

MessagePart(byte[], int, int, String, String, String, String)

Declaration:

```
public MessagePart(byte[] contents, int offset, int length, java.lang.String mimeType,
            java.lang.String contentId, java.lang.String contentLocation, java.lang.
            String enc)
            throws SizeExceededException
```

Description:

Constructs a `MessagePart` object from a subset of the byte array. This constructor is only useful, if the data size is small (roughly less than 10K). For larger content the `InputStream` based constructor should be used.

Parameters:

`contents` - byte array containing the contents for the `MessagePart`

`offset` - start position

`length` - the number of bytes to be included in the `MessagePart`

`mimeType` - the MIME Content-Type for the `MessagePart` [RFC 2046]

`contentId` - the content-id header field value for the `MessagePart` [RFC 2045]. The content-id is unique over all `MessageParts` of a `MultipartMessage` and must always be set for each message part

`contentLocation` - the content location which specifies the file name of the file that is attached. If the content location is set to `null` no content location will be set for this `MessagePart`.

`enc` - the encoding scheme for the `MessagePart`. If enc is set to `null` no encoding will be used for this `MessagePart`.

Throws:

`java.lang.IllegalArgumentException` - if mimeType or contentId is `null`. This exception will be thrown if `contentID` or `contentLocation` contains other characters than specified in US-ASCII format This exception will be thrown if either `length` is less than `0` or `offset + length` exceeds the `length` of the `content` or if `offset` is less than `0` or if the specified encoding scheme is unknown.

`SizeExceededException` - if the `contents` is larger than the available memory or supported size for the message part.

MessagePart(byte[], String, String, String, String)

Declaration:

```
public MessagePart(byte[] contents, java.lang.String mimeType, java.lang.
            String contentId, java.lang.String contentLocation, java.lang.String enc)
            throws SizeExceededException
```

Description:

Constructs a `MessagePart` object from a byte array. This constructor is only useful, if the data size is small (roughly less than 10K). For larger content the `InputStream` based constructor should be used.

Parameters:

`contents` - byte array containing the contents for the `MessagePart`. The contents of the array will be copied into the `MessagePart`.

`mimeType` - the MIME Content-Type for the `MessagePart` [RFC 2046]

`contentId` - the content-id header field value for the `MessagePart` [RFC 2045]. The content-id is unique over all `MessageParts` of a `MultipartMessage` and must always be set for each message part

`contentLocation` - the content location which specifies the file name of the file that is attached. If the content location is set to `null` no content location will be set for this `MessagePart`.

`enc` - the encoding scheme for the `MessagePart`. If enc is set to `null` no encoding will be used for this `MessagePart`.

Throws:

`java.lang.IllegalArgumentException` - if mimeType or contentId is `null`. This exception will be thrown if `contentID` or `contentLocation` contains other characters than specified in US-ASCII format or if the specified encoding scheme is unknown

`SizeExceededException` - if the `contents` is larger than the available memory or supported size for the message part.

MessagePart(InputStream, String, String, String, String)

Declaration:

```
public MessagePart(java.io.InputStream is, java.lang.String mimeType, java.lang.
            String contentId, java.lang.String contentLocation, java.lang.String enc)
            throws IOException, SizeExceededException
```

Description:

Constructs a `MessagePart` object from an `InputStream`. The contents of the `MessagePart` are loaded from the `InputStream` during the constructor call until the end of stream is reached.

Parameters:

`is` - `InputStream` from which the contents of the `MessagePart` are read

`mimeType` - the MIME Content-Type for the `MessagePart` [RFC 2046]

`contentId` - the content-id header field value for the `MessagePart` [RFC 2045]. The content-id is unique over all `MessageParts` of a `MultipartMessage` and must always be set for each message part

`contentLocation` - the content location which specifies the file name of the file that is attached. If the content location is set to `null` no content location will be set for this `MessagePart`.

`enc` - the encoding scheme for the `MessagePart`. If enc is set to `null` no encoding will be used for this `MessagePart`.

Throws:

`java.io.IOException` - if the reading of the `InputStream` causes an exception other than `EOFException`

`java.lang.IllegalArgumentException` - if mimeType or contentId is `null`. This exception will be thrown if `contentID` or `contentLocation` contains other characters than specified in US-ASCII format or if the specified encoding scheme is unknown.

`SizeExceededException` - if the content from the `InputStream` is larger than the available memory or supported size for the message part.

Methods

getContent()

Declaration:
`public byte[] getContent()`

Description:
Returns the content of the `MessagePart` as an array of bytes. If it's not possible to create an array, which can contain all data, this method must throw an `OutOfMemoryError`.

Returns: `MessagePart` data as byte array

getContentAsStream()

Declaration:
`public java.io.InputStream getContentAsStream()`

Description:
Returns an `InputStream` for reading the contents of the `MessagePart`. Returns an empty stream if no content is available.

Returns: an `InputStream` that can be used for reading the contents of this `MessagePart`

getContentID()

Declaration:
`public java.lang.String getContentID()`

Description:
Returns the content-id value of the `MessagePart`

Returns: the value of content-id as a String, or `null` if the content-id is not set It might happen if the message was sent from a not JSR 205 compliant client.

getContentLocation()

Declaration:
`public java.lang.String getContentLocation()`

Description:
Returns content location of the `MessagePart`

Returns: content location

getEncoding()

Declaration:
`public java.lang.String getEncoding()`

Description:
Returns the encoding of the content, e.g. "US-ASCII", "UTF-8", "UTF-16", ... as a `String`

Returns: encoding of the `MessagePart` content or `null` if the encoding scheme of the `MessagePart` cannot be determined

getLength()

Declaration:
```
public int getLength()
```

Description:
Returns the content size of this `MessagePart`

Returns: Content size (in bytes) of this `MessagePart` or `0` if the `MessagePart` is empty.

getMIMEType()

Declaration:
```
public java.lang.String getMIMEType()
```

Description:
Returns the mime type of the `MessagePart`

Returns: MIME type of the `MessagePart`

javax.wireless.messaging
MultipartMessage

Declaration
`public interface` **`MultipartMessage extends`** `Message`

All Superinterfaces: `Message`

Description
An interface representing a multipart message. This is a subinterface of `Message` which contains methods to add and get `MessageParts`. The interface also allows to specify the subject of the message. The basic methods for manipulating the address portion of the message are inherited from `Message`. Additional this interface defines methods for adding and removing addresses to/from the “to”, “cc” or “bcc” fields. Furthermore it offers methods to get and set special header fields of the message. The contents of the MultipartMessage are assembled during the invocation of the MessageConnection.send() method. The contents of each MessagePart are copied before the send message returns. Changes to the MessagePart contents after send must not appear in the transmitted message.

Since: WMA 2.0

Member Summary

Methods

boolean	addAddress(java.lang.String type, java.lang.String address)
void	addMessagePart(MessagePart part)
java.lang.String	getAddress()
java.lang.String[]	getAddresses(java.lang.String type)
java.lang.String	getHeader(java.lang.String headerField)
MessagePart	getMessagePart(java.lang.String contentID)
MessagePart[]	getMessageParts()
java.lang.String	getStartContentId()
java.lang.String	getSubject()
boolean	removeAddress(java.lang.String type, java.lang.String address)
void	removeAddresses()
void	removeAddresses(java.lang.String type)
boolean	removeMessagePart(MessagePart part)
boolean	removeMessagePartId(java.lang.String contentID)
boolean	removeMessagePartLocation(java.lang.String contentLocation)
void	setAddress(java.lang.String addr)
void	setHeader(java.lang.String headerField, java.lang.String headerValue)
void	setStartContentId(java.lang.String contentId)
void	setSubject(java.lang.String subject)

Inherited Member Summary
Methods inherited from interface `Message`
`getTimestamp()`

Methods

addAddress(String, String)

Declaration:
```
public boolean addAddress(java.lang.String type, java.lang.String address)
```

Description:
Adds an address to the multipart message.

Parameters:
`type` - the address type ("to", "cc" or "bcc") as a `String`. Each message can have none or multiple "to", "cc" and "bcc" addresses. Each address is added separately. The type is not case sensitive. The implementation of `MessageConnection.send()` makes sure that the "from" address is set correctly.

`address` - the address as a `String`

Returns: `true` if it was possible to add the address, else `false`

Throws:
`java.lang.IllegalArgumentException` - if type is none of "to", "cc", or "bcc" or if `address` is not valid.

See Also: `setAddress(String)`

addMessagePart(MessagePart)

Declaration:
```
public void addMessagePart(javax.wireless.messaging.MessagePart part)
            throws SizeExceededException
```

Description:
Attaches a `MessagePart` to the multipart message

Parameters:
`part` - `MessagePart` to add

Throws:
`java.lang.IllegalArgumentException` - if the Content-ID of the `MessagePart` conflicts with a Content-ID of a `MessagePart` already contained in this `MultipartMessage`. The Content-IDs must be unique within a MultipartMessage.

NullPointerException - if the parameter is `null`

`SizeExceededException` - if it's not possible to attach the `MessagePart`.

getAddress()

Declaration:
```
public java.lang.String getAddress()
```

getAddresses(String)

Description:
Returns the "from" address associated with this message, e.g. address of the sender. If message is a newly created message, e.g. not a received one, then the first "to" address is returned.

Returns `null`, if the "from" or "to" addresses for the message, dependent on the case, are not set.

Note: This design allows sending responses to a received message easily by reusing the same `Message` object and just replacing the payload. The address field can normally be kept untouched (unless the used messaging protocol requires some special handling of the address).

Overrides: `getAddress` in interface `Message`

Returns: the "from" or "to" address of this message, or `null` if the address that is expected as a result of the method is not set

See Also: `setAddress(String)`

getAddresses(String)

Declaration:
```
public java.lang.String[] getAddresses(java.lang.String type)
```

Description:
Gets the addresses of the multipart message of the specified type. (e.g. "to", "cc", "bcc" or "from") as `String`. The method is not case sensitive.

Returns: the addresses as a `String array` or `null` if the address of the specified type is not present.

getHeader(String)

Declaration:
```
public java.lang.String getHeader(java.lang.String headerField)
```

Description:
Gets the content of the specific header field of the multipart message.

Parameters:
`headerField` - the name of the header field as a `String`

Returns: the content of the specified header field as a `String` or `null` if the specified header field is not present.

Throws:
SecurityException - if the access to specified header field is restricted

`java.lang.IllegalArgumentException` - if `headerField` is unknown

See Also: `Appendix D for known headerFields`

getMessagePart(String)

Declaration:
```
public javax.wireless.messaging.MessagePart getMessagePart(java.lang.String contentID)
```

Description:
This method returns a `MessagePart` from the message that matches the content-id passed as a parameter

Parameters:
`contentID` - the content-id for the `MessagePart` to be returned

Returns: `MessagePart` that matches the provided content-id or `null` if there is no `MessagePart` in this message with the provided content-id

Throws:
NullPointerException - if the parameter is `null`

getMessageParts()

Declaration:
```
public javax.wireless.messaging.MessagePart[] getMessageParts()
```

Description:
Returns an array of all `MessageParts` of this message

Returns: array of `MessageParts`, or `null`, if no `MessageParts` are available

getStartContentId()

Declaration:
```
public java.lang.String getStartContentId()
```

Description:
Returns the `contentId` of the start `MessagePart`. The start `MessagePart` is set in setStartContentId(String)

Returns: the content-id of the start `MessagePart` or `null` if the start `MessagePart` is not set.

See Also: `setStartContentId(String)`

getSubject()

Declaration:
```
public java.lang.String getSubject()
```

Description:
Gets the subject of the multipart message.

Returns: the message subject as a `String` or `null` if this value is not present.

removeAddress(String, String)

Declaration:
```
public boolean removeAddress(java.lang.String type, java.lang.String address)
```

Description:
Removes an address from the multipart message.

Parameters:
`type` - the address type (“to”, “cc”, or “bcc”) as a `String`.

`address` - the address as a `String`

Returns: `true` if it was possible to delete the address, else `false`

Throws:
NullPointerException - if type is `null`

`java.lang.IllegalArgumentException` - if type is none of “to”, “cc”, or “bcc”

removeAddresses()

Declaration:
`public void removeAddresses()`

Description:
Removes all addresses of types “to”, “cc”, and bcc“ from the multipart message.

See Also: `setAddress(String)`, `addAddress(String, String)`

removeAddresses(String)

Declaration:
`public void removeAddresses(java.lang.String type)`

Description:
Removes all addresses of the specified type from the multipart message.

Parameters:
`type` - the address type (“to”, “cc”, or “bcc”) as a `String`.

Throws:
NullPointerException - if type is `null`

`java.lang.IllegalArgumentException` - if type is none of “to”, “cc”, or “bcc”

removeMessagePart(MessagePart)

Declaration:
`public boolean removeMessagePart(javax.wireless.messaging.MessagePart part)`

Description:
Removes a `MessagePart` from the multipart message

Parameters:
`part` - `MessagePart` to delete

Returns: `true`, if it was possible to remove the `MessagePart`, else `false`

Throws:
NullPointerException - if the parameter is `null`

removeMessagePartId(String)

Declaration:
`public boolean removeMessagePartId(java.lang.String contentID)`

Description:
Removes a `MessagePart` with the specific `contentID` from the multipart message

Parameters:
`contentID` - identifiers which `MessagePart` must be deleted.

Returns: `true`, if it was possible to remove the `MessagePart`, else `false`

Throws:
NullPointerException - if the parameter is `null`

removeMessagePartLocation(String)

Declaration:
`public boolean removeMessagePartLocation(java.lang.String contentLocation)`

Description:
Removes `MessageParts` with the specific content location from the multipart message. All `MessageParts` with the specified `contentLocation` are removed

Parameters:
`contentLocation` - content location (file name) of the `MessagePart`

Returns: `true`, if it was possible to remove the `MessagePart`, else `false`

Throws:
NullPointerException - if the parameter is `null`

setAddress(String)

Declaration:
```
public void setAddress(java.lang.String addr)
```

Description:
Sets the "to" address associated with this message. It works the same way as addAddress ("to", addr) The address may be set to `null`.

Overrides: `setAddress` in interface `Message`

Parameters:
`addr` - address for the message

Throws:
`java.lang.IllegalArgumentException` - if `address` is not valid.

See Also: `getAddress()`, `addAddress(String, String)`

setHeader(String, String)

Declaration:
```
public void setHeader(java.lang.String headerField, java.lang.String headerValue)
```

Description:
Sets the specific header of the multipart message. The header value can be `null`.

Parameters:
`headerField` - the name of the header field as a `String`

`headerValue` - the value of the header as a `String`

Throws:
`java.lang.IllegalArgumentException` - if `headerField` is unknown, or if `headerValue` is not correct (depends on `headerField`!)

NullPointerException - if `headerField` is `null`

SecurityException - if the access to specified header field is restricted

See Also: `getHeader(String)`, `Appendix D`

setStartContentId(String)

Declaration:
```
public void setStartContentId(java.lang.String contentId)
```

Description:
Sets the `Content-ID` of the start `MessagePart` of a multipart related message. The `Content-ID` may be set to `null`. The `StartContentId` is set for the MessagePart that is used to reference the other MessageParts of the `MultipartMessage` for presentation or processing purposes.

Parameters:
`contentId` - as a `String`

Throws:
`java.lang.IllegalArgumentException` - if `contentId` is none of the added `MessageParts` objects matches the `contentId`

See Also: `getStartContentId()`

setSubject(String)

Declaration:
```
public void setSubject(java.lang.String subject)
```

Description:
Sets the Subject of the multipart message. This value can be `null`.

Parameters:
`subject` - the message subject as a `String`

See Also: `getSubject()`

javax.wireless.messaging
SizeExceededException

Declaration
```
public class SizeExceededException extends java.io.IOException
```

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--java.io.IOException
                    |
                    +--javax.wireless.messaging.SizeExceededException
```

Description
Inidicates, that an operation is not executable due to insufficient system resources.

Since: WMA 2.0

Member Summary
Constructors
SizeExceededException(java.lang.String reason)

Inherited Member Summary
Methods inherited from class `Object`
`equals(Object), getClass(), hashCode(), notify(), notifyAll(), wait(), wait(), wait()`
Methods inherited from class `Throwable`
`getMessage(), printStackTrace(), toString()`

Constructors

SizeExceededException(String)

Declaration:
```
public SizeExceededException(java.lang.String reason)
```

Description:
Constructs e new exception

SizeExceededException(String)

Parameters:

`reason` - The reason why this exception occurs

javax.wireless.messaging
TextMessage

Declaration
```
public interface TextMessage extends Message
```

All Superinterfaces: Message

Description
An interface representing a text message. This is a subinterface of `Message` which contains methods to get and set the text payload. The `setPayloadText` method sets the value of the payload in the data container without any checking whether the value is valid in any way. Methods for manipulating the address portion of the message are inherited from `Message`.

Object instances implementing this interface are just containers for the data that is passed in.

Character Encoding Considerations
Text messages using this interface deal with `String`s encoded in Java. The underlying implementation will convert the `String`s into a suitable encoding for the messaging protocol in question. Different protocols recognize different character sets. To ensure that characters are transmitted correctly across the network, an application should use the character set(s) recognized by the protocol. If an application is unaware of the protocol, or uses a character set that the protocol does not recognize, then some characters might be transmitted incorrectly.

Member Summary	
Methods	
java.lang.String	getPayloadText()
void	setPayloadText(java.lang.String data)

Inherited Member Summary
Methods inherited from interface Message
getAddress(), getTimestamp(), setAddress(String)

Methods

getPayloadText()

Declaration:
```
public java.lang.String getPayloadText()
```

Description:
Returns the message payload data as a `String`.

Returns: the payload of this message, or `null` if the payload for the message is not set

See Also: `setPayloadText(String)`

setPayloadText(String)

Declaration:
```
public void setPayloadText(java.lang.String data)
```

Description:
Sets the payload data of this message. The payload data may be `null`.

Parameters:
`data` - payload data as a `String`

See Also: `getPayloadText()`

APPENDIX A

GSM SMS Adapter

This appendix describes an adapter that uses the messaging API with the GSM Short Message Service.

A.1.0 GSM SMS Message Structure

The GSM SMS messages are defined in the GSM 03.40 standard [1]. The message consists of a fixed header and a field called TP-User-Data. The TP-User-Data field carries the payload of the short message and optional header information that is not part of the fixed header. This optional header information is contained in a field called User-Data-Header. The presence of optional header information in the TP-User-Data field is indicated by a separate field that is part of the fixed header.

The TP-User-Data can use different encodings depending on the type of the payload content. Possible encodings are a 7-bit alphabet defined in the GSM 03.38 standard, 8-bit binary data, or 16-bit UCS-2 alphabet.

A.1.1 Message Payload Length

The maximum length of the SMS protocol message payload depends on the encoding and whether there are optional headers present in the TP-User-Data field. If the optional header information specifies a port number, then the payload which fits into the SMS protocol message will be smaller. Typically, the message is displayed to the end user. However, this Java API supports the use of port numbers to specify a Java application as the message target.

The messages that the Java application sends can be too long to fit in a single SMS protocol message. In this case, the implementation MUST use the concatenation feature specified in sections 9.2.3.24.1 and 9.2.3.24.8 of the GSM 03.40 standard [1]. This feature can be used to split the message payload given to the Java API into multiple SMS protocol messages. Similarly, when receiving messages, the implementation MUST automatically concatenate the received SMS protocol messages and pass the fully reassembled payload to the application via the API.

A.1.2 Message Payload Concatenation

The GSM 03.40 standard [1] specifies two mechanisms for the concatenation, specified in sections 9.2.3.24.1 and 9.2.3.24.8. They differ in the length of the reference number. For messages that are sent, the implementation

can use either mechanism. For received messages, implementations MUST accept messages with both mechanisms.

Note: Depending on which mechanism is used for sending messages, the maximum length of the payload of a single SMS protocol message differs by one character/byte. For concatenation to work, regardless of which mechanism is used by the implementation, applications are recommended to assume the 16-bit reference number length when estimating how many SMS protocol messages it will take to send a given message. The lengths in Table A-1 below are calculated assuming the 16-bit reference number length.

Implementations of this API MUST support at least 3 SMS protocol messages to be received and concatenated together. Similarly, for sending, messages that can be sent with up to 3 SMS protocol messages MUST be supported. Depending on the implementation, these limits may be higher. However, applications are advised not to send messages that will take up more than 3 SMS protocol messages, unless they have reason to assume that the recipient will be able to handle a larger number. The `MessageConnection.numberOfSegments` method allows the application to check how many SMS protocol messages a given message will use when sent.

Table A-1: Number of SMS protocol messages needed for different payload lengths

Optional Headers Encoding	No port number present (message to be displayed to the end user)		Port number present (message targeted at an application)	
	Length	**SMS messages**	**Length**	**SMS messages**
GSM 7-bit alphabet	0-160 chars	1	0-152 chars	1
	161-304 chars	2	153-290 chars	2
	305-456 chars	3	291-435 chars	3
8-bit binary data	0-140 bytes	1	0-133 bytes	1
	41-266 bytes	2	134-254 bytes	2
	267-399 bytes	3	255-381 bytes	3
UCS-2 alphabet	0-70 chars	1	0-66 chars	1
	71-132 chars	2	67-126 chars	2
	133-198 chars	3	127-189 chars	3

Table A-1 assumes for the GSM 7-bit alphabet that only characters that can be encoded with a single septet are used. If a character that encodes into two septets (using the escape code to the extension table) is used, it counts as two characters in this length calculation.

Note: the values in Table A-1 include a concatenation header in all messages, when the message can not be sent in a single SMS protocol message.

Character Mapping Table

GSM 7-bit	UCS-2	Character name
0x00	0x0040	COMMERCIAL AT
0x01	0x00a3	POUND SIGN
0x02	0x0024	DOLLAR SIGN
0x03	0x00a5	YEN SIGN
0x04	0x00e8	LATIN SMALL LETTER E WITH GRAVE
0x05	0x00e9	LATIN SMALL LETTER E WITH ACUTE
0x06	0x00f9	LATIN SMALL LETTER U WITH GRAVE

GSM 7-bit	UCS-2	Character name
0x07	0x00ec	LATIN SMALL LETTER I WITH GRAVE
0x08	0x00f2	LATIN SMALL LETTER O WITH GRAVE
0x09	0x00c7	LATIN CAPITAL LETTER C WITH CEDILLA
0x0a	0x000a	control: line feed
0x0b	0x00d8	LATIN CAPITAL LETTER O WITH STROKE
0x0c	0x00f8	LATIN SMALL LETTER O WITH STROKE
0x0d	0x000d	control: carriage return
0x0e	0x00c5	LATIN CAPITAL LETTER A WITH RING ABOVE
0x0f	0x00e5	LATIN SMALL LETTER A WITH RING ABOVE
0x10	0x0394	GREEK CAPITAL LETTER DELTA
0x11	0x005f	LOW LINE
0x12	0x03a6	GREEK CAPITAL LETTER PHI
0x13	0x0393	GREEK CAPITAL LETTER GAMMA
0x14	0x039b	GREEK CAPITAL LETTER LAMDA
0x15	0x03a9	GREEK CAPITAL LETTER OMEGA
0x16	0x03a0	GREEK CAPITAL LETTER PI
0x17	0x03a8	GREEK CAPITAL LETTER PSI
0x18	0x03a3	GREEK CAPITAL LETTER SIGMA
0x19	0x0398	GREEK CAPITAL LETTER THETA
0x1a	0x039e	GREEK CAPITAL LETTER XI
0x1b	xxx	escape to extension table
0x1c	0x00c6	LATIN CAPITAL LETTER AE
0x1d	0x00e6	LATIN SMALL LETTER AE
0x1e	0x00df	LATIN SMALL LETTER SHARP S
0x1f	0x00c9	LATIN CAPITAL LETTER E WITH ACUTE
0x20	0x0020	SPACE
0x21	0x0021	EXCLAMATION MARK
0x22	0x0022	QUOTATION MARK
0x23	0x0023	NUMBER SIGN
0x24	0x00a4	CURRENCY SIGN
0x25	0x0025	PERCENT SIGN
0x26	0x0026	AMPERSAND
0x27	0x0027	APOSTROPHE
0x28	0x0028	LEFT PARENTHESIS
0x29	0x0029	RIGHT PARENTHESIS
0x2a	0x002a	ASTERISK
0x2b	0x002b	PLUS SIGN
0x2c	0x002c	COMMA
0x2d	0x002d	HYPHEN-MINUS
0x2e	0x002e	FULL STOP
0x2f	0x002f	SOLIDUS
0x30	0x0030	DIGIT ZERO
0x31	0x0031	DIGIT ONE
0x32	0x0032	DIGIT TWO
0x33	0x0033	DIGIT THREE
0x34	0x0034	DIGIT FOUR
0x35	0x0035	DIGIT FIVE
0x36	0x0036	DIGIT SIX
0x37	0x0037	DIGIT SEVEN
0x38	0x0038	DIGIT EIGHT

GSM 7-bit	UCS-2	Character name
0x39	0x0039	DIGIT NINE
0x3a	0x003a	COLON
0x3b	0x003b	SEMICOLON
0x3c	0x003c	LESS-THAN SIGN
0x3d	0x003d	EQUALS SIGN
0x3e	0x003e	GREATER-THAN SIGN
0x3f	0x003f	QUESTION MARK
0x40	0x00a1	INVERTED EXCLAMATION MARK
0x41	0x0041	LATIN CAPITAL LETTER A
0x42	0x0042	LATIN CAPITAL LETTER B
0x43	0x0043	LATIN CAPITAL LETTER C
0x44	0x0044	LATIN CAPITAL LETTER D
0x45	0x0045	LATIN CAPITAL LETTER E
0x46	0x0046	LATIN CAPITAL LETTER F
0x47	0x0047	LATIN CAPITAL LETTER G
0x48	0x0048	LATIN CAPITAL LETTER H
0x49	0x0049	LATIN CAPITAL LETTER I
0x4a	0x004a	LATIN CAPITAL LETTER J
0x4b	0x004b	LATIN CAPITAL LETTER K
0x4c	0x004c	LATIN CAPITAL LETTER L
0x4d	0x004d	LATIN CAPITAL LETTER M
0x4e	0x004e	LATIN CAPITAL LETTER N
0x4f	0x004f	LATIN CAPITAL LETTER O
0x50	0x0050	LATIN CAPITAL LETTER P
0x51	0x0051	LATIN CAPITAL LETTER Q
0x52	0x0052	LATIN CAPITAL LETTER R
0x53	0x0053	LATIN CAPITAL LETTER S
0x54	0x0054	LATIN CAPITAL LETTER T
0x55	0x0055	LATIN CAPITAL LETTER U
0x56	0x0056	LATIN CAPITAL LETTER V
0x57	0x0057	LATIN CAPITAL LETTER W
0x58	0x0058	LATIN CAPITAL LETTER X
0x59	0x0059	LATIN CAPITAL LETTER Y
0x5a	0x005a	LATIN CAPITAL LETTER Z
0x5b	0x00c4	LATIN CAPITAL LETTER A WITH DIARESIS
0x5c	0x00d6	LATIN CAPITAL LETTER O WITH DIARESIS
0x5d	0x00d1	LATIN CAPITAL LETTER N WITH TILDE
0x5e	0x00dc	LATIN CAPITAL LETTER U WITH DIARESIS
0x5f	0x00a7	SECTION SIGN
0x60	0x00bf	INVERTED QUESTION MARK
0x61	0x0061	LATIN SMALL LETTER A
0x62	0x0062	LATIN SMALL LETTER B
0x63	0x0063	LATIN SMALL LETTER C
0x64	0x0064	LATIN SMALL LETTER D
0x65	0x0065	LATIN SMALL LETTER E
0x66	0x0066	LATIN SMALL LETTER F
0x67	0x0067	LATIN SMALL LETTER G
0x68	0x0068	LATIN SMALL LETTER H
0x69	0x0069	LATIN SMALL LETTER I
0x6a	0x006a	LATIN SMALL LETTER J

GSM 7-bit	UCS-2	Character name
0x6b	0x006b	LATIN SMALL LETTER K
0x6c	0x006c	LATIN SMALL LETTER L
0x6d	0x006d	LATIN SMALL LETTER M
0x6e	0x006e	LATIN SMALL LETTER N
0x6f	0x006f	LATIN SMALL LETTER O
0x70	0x0070	LATIN SMALL LETTER P
0x71	0x0071	LATIN SMALL LETTER Q
0x72	0x0072	LATIN SMALL LETTER R
0x73	0x0073	LATIN SMALL LETTER S
0x74	0x0074	LATIN SMALL LETTER T
0x75	0x0075	LATIN SMALL LETTER U
0x76	0x0076	LATIN SMALL LETTER V
0x77	0x0077	LATIN SMALL LETTER W
0x78	0x0078	LATIN SMALL LETTER X
0x79	0x0079	LATIN SMALL LETTER Y
0x7a	0x007a	LATIN SMALL LETTER Z
0x7b	0x00e4	LATIN SMALL LETTER A WITH DIARESIS
0x7c	0x00f6	LATIN SMALL LETTER O WITH DIARESIS
0x7d	0x00f1	LATIN SMALL LETTER N WITH TILDE
0x7e	0x00fc	LATIN SMALL LETTER U WITH DIARESIS
0x7f	0x00e0	LATIN SMALL LETTER A WITH GRAVE
0x1b 0x14	0x005e	CIRCUMFLEX ACCENT
0x1b 0x28	0x007b	LEFT CURLY BRACKET
0x1b 0x29	0x007d	RIGHT CURLY BRACKET
0x1b 0x2f	0x005c	REVERSE SOLIDUS
0x1b 0x3c	0x005b	LEFT SQUARE BRACKET
0x1b 0x3d	0x007e	TILDE
0x1b 0x3e	0x005d	RIGHT SQUARE BRACKET
0x1b 0x40	0x007c	VERTICAL LINE
0x1b 0x65	0x20ac	EURO SIGN

The GSM 7-bit characters that use the escape code for a two septet combination are represented in this table with the hexadecimal representations of the two septets separately. In the encoded messages, the septets are encoded together with no extra alignment to octet boundaries.

A.2.0 Message Addressing

The syntax of the URL connection strings that specify the address are described in Table A-2.

Table A-2: Connection Strings for Message Addresses

String	Definition
smsurl	::== "sms://" address_part
address_part	::== foreign_host_address | local_host_address
local_host_address	::== port_number_part
port_number_part	::== ":" digits
foreign_host_address	::== msisdn | msisdn port_number_part
msisdn	::== "+" digits | digits
digit	::== "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
digits	::== digit | digit digits

Examples of valid URL connection strings are:

```
sms://+358401234567
sms://+358401234567:6578
sms://:3381
```

When this adapter is used and the `Connector.open()` method is passed a URL with this syntax, it MUST return an instance implementing the `javax.wireless.messaging.MessageConnection` interface.

A.2.1 Specifying Recipient Addresses

In this URL connection string, the MSISDN part identifies the recipient phone number and the port number part of the application port number address as specified in the GSM 3.40 SMS specification [1] (sections 9.2.3.24.3 and 9.2.3.24.4). The same mechanism is used, for example, for the WAP WDP messages.

When the port number is present in the address, the TP-User-Data of the SMS MUST contain a User-Data-Header with the Application port addressing scheme information element.

When the recipient address does not contain a port number, the TP-User-Data MUST NOT contain the Application port addressing header. Java applications cannot receive this kind of message, but it will be handled as usual in the recipient device; for example, text messages will be displayed to the end user.

A.2.2 Client Mode and Server Mode Connections

Messages can be sent using this API via client or server type `MessageConnection`s. When a message identifying a port number is sent from a server type `MessageConnection`, the originating port number in the message is set to the port number of the `MessageConnection`. This allows the recipient to send a response to the message that will be received by this `MessageConnection`.

However, when a client type `MessageConnection` is used for sending a message with a port number, the originating port number is set to an implementation-specific value and any possible messages received to this port number are not delivered to the `MessageConnection`.

Thus, only the server mode `MessageConnection`s can be used for receiving messages. Any messages to which the other party is expected to respond should be sent using the appropriate server mode `MessageConnection`.

A.2.3 Handling Received Messages

When SMS messages are received by an application, they are removed from the SIM/ME memory where they may have been stored.

If the message information MUST be stored more persistently, then the application is responsible for saving it. For example, the application could could save the message information by using the RMS facility of the MIDP API or any other available mechanism.

The GSM SMS protocol does not guarantee to preserve the ordering when multiple messages are sent. When a large message is split into multiple GSM SMS sections as specified in A.1.2, ordering is handled correctly when they are automatically concatenated back into a single `Message` object. If the application sends multiple `Messages` to the same recipient, they might not be delivered in the correct order. The application must be written so that it is able to deal with this issue appropriately. However, even when the ordering may change during the delivery in the network, the implementation MUST guarantee that the messages are delivered to the application in the same order as they were received by the implementation of the recipient terminal.

A.3.0 Short Message Service Center Address

Applications might need to obtain the Short Message Service Center (SMSC) address to decide which recipient number to use. For example, the application might need to do this because it is using service numbers for application servers which might not be consistent in all networks and SMSCs.

The SMSC address used for sending the messages MUST be made available using `System.getProperty` with the property name described in Table A-3.

Table A-3: Property Name and Description for SMSC Addresses

Property name	Description

`wireless.messaging.sms.smsc`	The address of the SMS expressed using the syntax expressed by the msisdn item of thefollowing BNF definition:
	`msisdn ::== "+" digits | digits`
	`digit ::== "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |"8" | "9"`
	`digits ::== digit | digit digits`

A.4.0 Using Port Numbers

The receiving application in a device is identified with the port number included in the message. When opening the server mode `MessageConnection`, the application specifies the port number that it will use for receiving messages.

The first application to allocate a given port number will get it. If other applications try to allocate the same port number while it is being used by the first application, an IOException will be thrown when they attempt to open the `MessageConnection`. The same rule applies if a port number is being used by a system application in the device. In this case, the Java applications will not be able to use that port number.

As specified in the GSM 03.40 standard [1], the port numbers are split into ranges. The IANA (Internet Assigned Numbers Authority) controls one of the ranges. If an application author wants to ensure that an application can always use a specific port number value, then it can be registered with IANA. Otherwise, the author can pick a number at random from the freely usable range and hope that the same number is not used by

another application that might be installed in the same device. This is exactly the same way that port numbers are currently used with TCP and UDP in the Internet.

A.5.0 Message Types

SMS messages can be sent using the `TextMessage` or the `BinaryMessage` message type of the API. The encodings used in the SMS protocol are defined in the GSM 03.38 standard (Part 4 SMS Data Coding Scheme) [2].

When the application uses the `TextMessage` type, the TP-Data-Coding-Scheme in the SMS MUST indicate the GSM default 7-bit alphabet or UCS-2. The TP-User-Data MUST be encoded appropriately using the chosen alphabet. The 7-bit alphabet MUST be used for encoding if the String that is given by the application only contains characters that are present in the GSM 7-bit alphabet. If the String given by the application contains at least one character that is not present in the GSM 7-bit alphabet, the UCS-2 encoding MUST be used.

When the application uses the `BinaryMessage`, the TP-Data-Coding-Scheme in the SMS MUST indicate 8-bit data.

The application is responsible for ensuring that the message payload fits in an SMS message when encoded as defined in this specification. If the application tries to send a message with a payload that is too long, the `MessageConnection.send()` method will throw an `IllegalArgumentException` and the message will not be sent. This specification contains the information that applications need to determine the maximum payload for the message type they are trying to send.

All messages sent via this API MUST be sent as Class 1 messages GSM 3.40 SMS specification [1], Section 9. 2.3.9 "TP-Protocol-Identifier".

A.6.0 Restrictions on Port Numbers for SMS Messages

For security reasons, Java applications are not allowed to send SMS messages to the port numbers listed in Table A-4. Implementations MUST throw a `SecurityException` in the `MessageConnection. send()` method if an application tries to send a message to any of these port numbers.

Table A-4: Port Numbers Restricted to SMS Messages

Port number	Description
2805	WAP WTA secure connection-less session service
2923	WAP WTA secure session service
2948	WAP Push connectionless session service (client side)
2949	WAP Push secure connectionless session service (client side)
5502	Service Card reader
5503	Internet access configuration reader
5508	Dynamic Menu Control Protocol
5511	Message Access Protocol
5512	Simple Email Notification
9200	WAP connectionless session service
9201	WAP session service
9202	WAP secure connectionless session service

Port number	Description
9203	WAP secure session service
9207	WAP vCal Secure
49996	SyncML OTA configuration
49999	WAP OTA configuration

APPENDIX B

GSM Cell Broadcast Adapter

This appendix describes an adapter that uses the messaging API with the GSM Cell Broadcast short message Service (CBS).

The Cell Broadcast service is a unidirectional data service where messages are broadcast by a base station and received by every mobile station listening to that base station. The Wireless Messaging API is used for receiving these messages.

B.1.0 GSM CBS message structure

The GSM CBS messages are defined in the GSM 03.41 standard [4].

The source/type of a CBS message is defined by its Message-Identifier field, which is used to choose topics to subscribe to. Applications can receive messages of a specific topic by opening a `MessageConnection` with a URL connection string in the format defined below. In the format, Message-Identifier is analogous to a port number.

Cell broadcast messages can be encoded using the same data coding schemes as GSM SMS messages (See Character Mapping Table in Appendix A, GSM SMS Adapter). The implementation of the API will convert messages encoded with the GSM 7-bit alphabet or UCS-2 into `TextMessage` objects and messages encoded in 8-bit binary to `BinaryMessage` objects.

Because the cell broadcast messages do not contain any timestamps, the `Message.getTimeStamp` method MUST always return null for received cell broadcast messages.

B.2.0 Addressing

Table B-1 describes the syntax of the URL connection strings that specify the address.

Table B-1: Syntax for URL Connection Strings

String	Description
cbsurl	`::== "cbs://"` address_part
address_part	`::==` message_identifier_part
message_identifier_par	`::== ":"` digits

String	Description
digit	::== "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
digits	::== digit | digit digits

Examples of valid URL connection strings are:

```
cbs://:3382
cbs://:3383
```

In this URL, the message identifier part specifies the message identifier of the cell broadcast messages that the application wants to receive.

When this adapter is used and the `Connector.open()` method is passed a URL with this syntax, it MUST return an instance implementing the `javax.wireless.messaging.MessageConnection` interface. These `MessageConnection` instances can be used only for receiving messages. Attempts to call the `send` method on these `MessageConnection` instances MUST result in an `IOException` being thrown.

APPENDIX C

CDMA IS-637 SMS Adapter

This appendix describes an adapter that uses the messaging API with the CDMA IS-637 SMS service.

C.1.0 CDMA IS-637 SMS Message Structure

CDMA SMS messages are defined in the CDMA IS-637 standard [6].

C.2.0 Addressing

The same `sms:` URL connection string is used as for GSM SMS (See Appendix A).

C.3.0 Port Numbers

The IS-637 SMS protocol does not include a port number or any other field for differentiating between recipient applications. For this purpose, the WAP WDP for IS-637 SMS defined in section 6.5 of the WAP Forum WDP specification[5] MUST be used.

Similarly, any rules for segmentation and reassembly follow the WAP WDP guidelines for adapting CDMA SMS messages for a common behavior with corresponding GSM SMS bearer capabilities.

Messages without a port number are sent as normal SMS messages targeted for presentation to the end user.

CDMA SMS messages MUST support a minimum of 3 concatenated messages to be consistent with the GSM SMS message adapter.

APPENDIX D

MMS Adapter

This appendix describes an adapter that uses the messaging API with the Multimedia Message Service.

D.1.0 MMS Message Structure

The MMS Protocol Data Unit (PDU) structure is specified in the WAP- 209-MMS-Encapsulation standard. The MMS PDU contains a header and a multipart message body. Some of the MMS header fields originate from standard RFC 822 headers and others are specific to multimedia messaging. The message body may contain any content type and MIME multipart is used to represent and encode a wide variety of media types for transmission via multimedia messaging. The MIME multipart [RFC 2045-7] is used in e-mail systems and therefore compatible. The content type of the MMS PDU is application/vnd.wap.mms-message. Figure D-1 shows the graphical representation of a conceptual model of the message encapsulation.

MMS-headers contain MMS-specific information of the PDU. This information contains mainly information how to transfer the multimedia message from the message originator to the recipient.

In the multimedia messaging use case, the message body consists of multipart/related structure [RFC2387] including multimedia objects, each in separate message part (further referred as MessagePart), as well as an optional presentation part. The order of the parts has no significance. The presentation part is related by the start parameter and its content identifier.

The MIME type of the message body and related to the MIME type parameters are stored in the Content-Type Header of the message. Multipart/related MIME type requires "start" parameter that is used for identifying the MessagePart with the presentation information (SMIL) . If the Content-Type does not contain "start" parameter, the MIME type is multipart/mixed.

If a MMS is targeted for the default MMS viewer, the application should respect the OMA-MMS-CONF-V1_2 OMA MMS conformance document V1.2 or, if available, a newer one.

Each of the MessageParts consists of multipart identification information and content. The headers of each part contain e.g. the following fields:

-MIME-Type indicating the content in the MessagePart, (e.g. image/jpeg or text/plain),
and possible
-Content-Location can be used as SMIL reference, and/or as file name hint, e.g. "image.jpeg" or "hello.txt".
-Content-Identifier (Content-ID) identifies the content (used for start parameter and as SMIL reference)

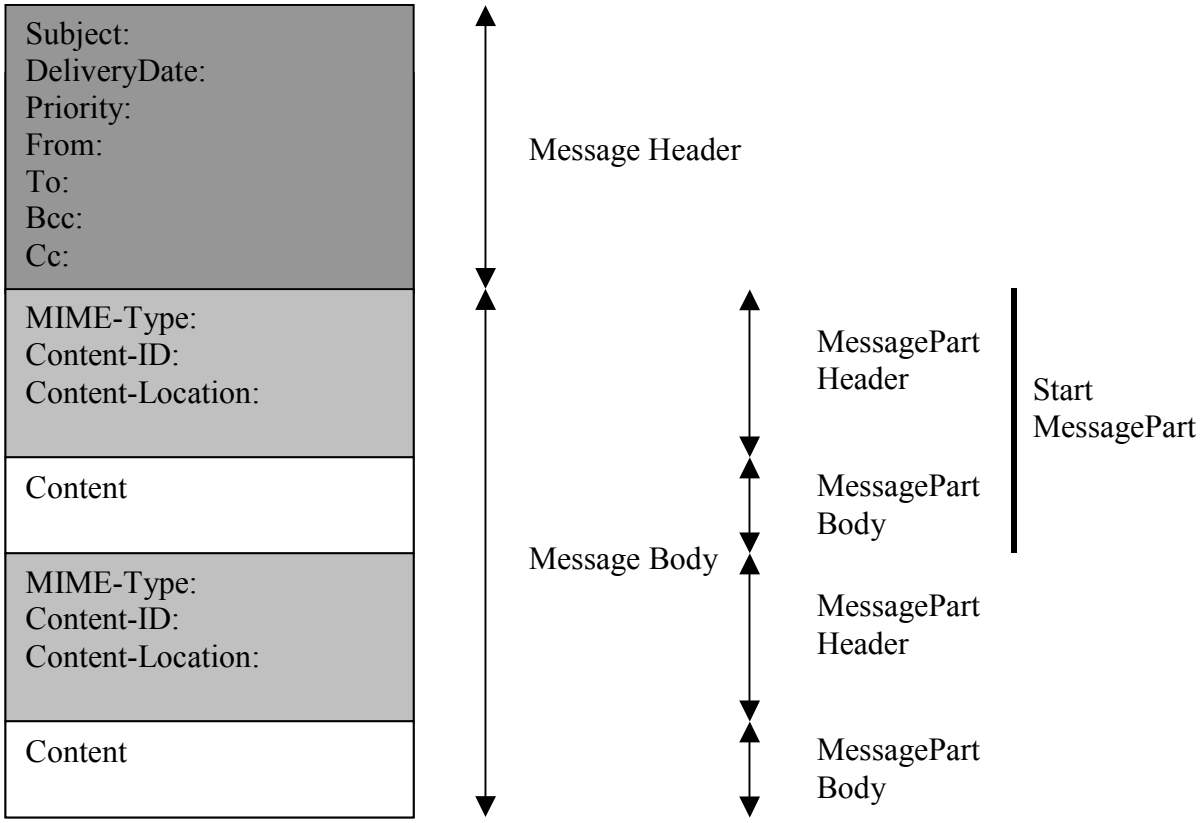


Figure D-1: Multimedia Message Structure

D.1.1 Accessible Header Fields

MMS message header contains several header fields. Nevertheless, due to the security reasons not all of them can be accessed from the Java application.

The following header fields can be accessed via the *MultipartMessage* methods *setHeader()* and *getHeader()*:

- *X-Mms-Delivery-Time*: The date and time of the delivery. This value specifies when MMS message must be delivered to the recipient. For example, a birthday card must be delivered exactly on day of birthday. The value must be given in milliseconds since 1.Jan. 1970 GMT as string value.

- *X-Mms-Priority* : The importance of the message. The value could be "high", "normal" or "low".

The following header fields are accessible indirectly via other methods of *MultipartMessage*:

- *X-Mms-Subject* : Message subject in UTF-16 encoding.
- *X-Mms-From* : From address which is set automatically.
- *X-Mms-To* : Address or list of addresses of the recipient.
- *X-Mms-CC* : Address or list of addresses where copy of the message must be sent.
- *X-Mms-BCC* : Address or list of addresses where copy of the message must be sent without notifying other recipients. *Please note that the BCC address is optional. It is up to the implementation whether it supports this header field or not.*

D.2.0 MMS Message Addressing

The multipart message addressing model contains different types of addresses:

- global telephone number of recipient user, including telephone number, ipv4, ipv6 addresses
- e-mail address as specified in RFC 822
- short-code of the service (Note: not valid for MMS version 1.0)

The syntax of the URL connection strings that specify the address are described in Table D-2. The application-id is mandatory in the case of addressing a MMS to a Java application and specifies the recipient application.

Notation

1*x includes at least one occurrence of x
*x includes zero or more occurrences of x
1*4(x) includes at least 1 and as maximum 4 occurrences of x
[x] x is an optional element
(x) x is an regular expression
x|y x and y are alternatives
"a" char ’a’ is a part of the expression
; separator for the comments
<text> non formal textual description.

Table D-2: Connection Strings for Message Addresses

Note that RFC 2822 limits the character repertoire to ASCII

Note: E-mail address grammer is specified in RFC 822

Note: IPv6 and IPv4 address format is specified in RFC 1884, 2373

String	Definition
mms_url	::== "mms:// " address-part
address-part	::== (e-mail-address| device-address| shortcode-address| application-id)
device-address	::== general-phone-address [application-id]
general-phone-address	::== (global-telephone-type) | (ipv4) | (ipv6)
global-telephone-type	::== "+" 1*digit| 1*digit
ipv6	::== ipv6-atom ":" ipv6-atom ":" ipv6-atom ":" ipv6-atom ":" ipv6-atom ":" ipv6-atom ":" ipv6-atom ":" ipv6-atom
ipv6-atom	::== 1*4 (DIGIT|HEX-ALPHA)
ipv4	::== 1*3DIGIT "." 1*3DIGIT "." 1*3DIGIT "." 1*3DIGIT
application-id	::== ":" [manufacture_domain] [1*(package-name)] class-name
manufacture_domain	::== 1*applicationID-symbol "."
package-name	::== 1*applicationID-symbol "."
class-name	::== 1*applicationID-symbol
applicationID-symbol	::== ALPHA | DIGIT |"."|"_"
e-mail-address	::== mailbox | group
group	::== phrase ":" [mailbox *("," mailbox)] ";"
phrase	::== 1* word
mailbox	::== addr-spec | [phrase] route-addr
route-addr	::== "<" [route] addr-spec ">"
route	::== ("@" domain) [1*("," ("@" domain))] ":"
addr-spec	::== local-part "@" domain
local-part	::== word *("." word)
domain	::== sub-domain *("." sub-domain)
sub-domain	::== domain-ref | domain-literal
domain-ref	::== atom
domain-literal	::== "[" *(dtext | qpair) "]"
atom	::== 1* <any CHAR except SPECIALS, CTL and SPACE >
word	::== atom | quoted-string
quoted-string	::= """ *(qtext | qpair) """
qtext	::=<any CHAR excluding """, "\" and CR and including LINEAR-WHITE-SPACE>
dtext	::= <any CHAR excluding "[", "]", "\" and CR and including LINEAR-WHITE-SPACE>
LINEAR-WHITE-SPACE	::== 1* ([CRLF] LWSP-char)
SPECIALS	::== "(" | ")" | "<" | ">" | "@" | "," | ";" | ":" | "\" | """| "."| "[" | "]"
qpair	::== "\" ALPHA
shortcode-address	::== shortcode-string
shortcode-string	::== 1* (ALPHA| DIGIT)
HEX-ALPHA	::== "A.."F"
ALPHA	::== "A.."Z"|"a.."z"
DIGIT	::== "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
CHAR	::== <any ASCII character>; (0-177, 0. -127)
LWSP-char	::== SP| HT

Note that application_id is maximum 32 characters

Examples of valid URL connection strings are:

```
mms://+35467890
mms://+356728900:com.siemens.MyMessenger
mms://:com.siemens.MyMessenger4567
mms://test@domain.com
mms://12shortcode
```

When this adapter is used and the `Connector.open()` method is passed a URL with this syntax, it MUST return an instance implementing the `javax.wireless.messaging.MessageConnection` interface.

D.2.1 Sending and Receiving MMS Messages

To ensure sending and receiving wireless messages between Java applications using MMS as a transport medium, the MMS client has to support sending of MMS messages to concrete Java applications. In order to enable this, the following additional parameters are added to the Content-Type header field as additional Content-Type parameters:

Table D-3: Additional Content-Type parameters

Content-Type Parameter	Value
Application-ID	application-id (see Table D-2)
Reply-To-Application-ID	application-id (see Table D-2)

Usually the MMS client hands over an incoming MM to the presentation level. The scope of addressing scheme is to cause the MMS client to hand over the MM that contains an application identifier to the addressed application instead of using the normal path to the presentation level. The addressing scheme has to be implemented inside the MMS PDU (protocol data unit) sent to and received from the server.

If no application-id is specified in the connection string, the implementation must not use the additional Content-Type parameters.

Table D-4: Example of MMS header

Header Field	Header Value
X-Mms-Message-Type	m-send-req
X-Mms-Transaction-ID	543210
X-Mms-Version	1.0
X-Mms-Message-Class	Personal
X-Mms-Expiry	36000
X-Mms-Priority	Normal
From	+49170123456789
To	+490171123456789
Date	Fri, 7 Jul 2000 20:59:30 +0100
Subject	A multimedia message
Content-Type	application/vnd.wap.multipart.related; start = <start>; type = application/smil; **Application-ID= com.siemens.Messenger;** **Reply-To-Application-ID = com.siemens.Messenger**
nEntries	2
Content-Type	text/plain ; name="test.txt"
content-id	<a>
text data	

Header Field	Header Value
Content-Type	application/smil; name="first.sml"
content-id	<start>
smil file	

D.2.2 Client Mode and Server Mode Connection

Messages can be sent using this API via client or server type `MessageConnection`s. When a message identifying an application-id is sent from a server type `MessageConnection`, the Reply-To-Application-ID in the message is set to the application-id of the `MessageConnection`. This allows the recipient to send a response to the message that is received by this `MessageConnection`.

However, when a client type `MessageConnection` is used for sending a message with an application-id, the Reply-To-Application-ID is set to an implementation-specific value and any possible messages received to this application-id are not delivered to the `MessageConnection`.

Thus, only the server mode `MessageConnection`s can be used for receiving messages. Any messages to which the other party is expected to respond should be sent using the appropriate server mode `MessageConnection`.

D.2.3 Handling Received Messages

MMS message is received only in case the user confirms accepting the message after receiving a notification. The application is started if the application is registered to receive MMS with the specific application_id that must be specified inside the MMS message. If user confirmation is enabled, the Java application receives the MMS only after the user has confirmed the delivery.

When MMS messages are received by an application, they may be removed from the SIM/ME memory where they may have been stored.

If the message information MUST be stored more persistently, then the application is responsible for saving it. If the application cannot save the message due to the lack of space, the message may get lost. It is the application responsibility to ensure that there is enough space for storage. For example, the application could save the message information by using the RMS facility of the MIDP API or any other available mechanism.

D.3.0 Multimedia Message Service Center Address

Applications might need to obtain the Multimedia Message Service Center (MMSC) address to decide which recipient number to use. For example, the application might need to do this because it is using service numbers for application servers which might not be consistent in all networks and MMSCs.

The MMSC address used for sending the messages MUST be made available using `System.getProperty` with the property name described in Table D-*5*.

Table D-5: Property Name and Description for MMSC Addresses

Property name	Description
`wireless.messaging.mms.mmsc`	The address of the MMSC expressed using the syntax expressed by the MSISDN item of the following BNF definition: **`url`** `::== "http://" domain` **`domain`** `::== sub-domain *("." sub-domain)` **`sub-domain`** `::== domain-ref | domain-literal` **`domain-ref`** `::== atom` **`domain-literal`** `::== "[" *(dtext | qpair) "]"` **`dtext`** `:: == <any CHAR excluding "[", "]", "\" and CR and including LINEAR-WHITE-SPACE>` **`qpair`** `:: == "\" ALPHA` **`atom`** `::== 1* <any CHAR except SPECIALS, CTL and SPACE >` **`LINEAR-WHITE-SPACE`** `::== 1* ([CRLF] LWSP-char)` **`LWSP-char`** `::== SP| HT` **`CHAR`** `::== <any ASCII character>; (0-177, 0. -127)` **`ALPHA`** `::== "A".."Z"| "a".."z"` **`DIGIT`** `::== "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |"8" | "9"`

D.4.0 Using the Application-ID

The receiving application running on a device is identified with the application-id included in the message. The application-id must be specified only if the message is addressed to a special application. When opening the server mode `MessageConnection`, the application specifies the `Application-ID` that it uses for receiving messages.

The first application to allocate a given `Application-ID` gets it. If other applications try to allocate the same `application-id` while it is being used by the first application, an IOException is thrown when they attempt to open the `MessageConnection`. The same rule applies if an `Application-ID` is being used by a system application in the device. In this case, the Java applications is not be able to use that `Application-ID`.

The application must register with the `Application-ID` specified in a form that is described in the table D-2.

It's only possible to specify one `Application-ID` per `MultipartMessage`. If the application is calling `addAddress()` or `setAddress()` with different `Application-IDs` on the same `MultipartMessage`, the implementation must throw an `IllegalArgumentException`. To avoid this exception, the application must ensure that all specified addresses that contain an `Application-ID` have the same `Application-ID`, or just one `Application-ID` is specified per message (this `Application-ID` will then be used for all recipients). Please note that this makes it impossible to send a single message to applications (with `Application-IDs`) and native MMS clients (without them) in the same operation. To send the same message to an applications and native MMS clients the application must send the message twice

- once to the application-recipients with the `Application-ID` specified and once to the non-application recipients without any `Application-ID` specified.

D 5.0 DRM and MMS Content

The content that is sent via MMS can be DRM-protected. DRM protection can only be applied to individual parts. Handling of DRM-protected content is implementation specific. Only authorized applications must have access to the DRM-protected MMSs.

APPENDIX E

Deploying JSR 205 Interfaces on a MIDP 2.0 Platform

E.1.0 Introduction

This section provides implementation notes for platform developers deploying the JSR 205 interfaces on a MIDP 2.0 platform.

This section addresses features available in a MIDP 2.0 device that can be used to enhance WMA applications. In particular, this document describes how to:

- use the MIDP 2.0 security features to control access to WMA capabilities
- use the MIDP 2.0 Push mechanism with SMS, MMS and CBS messages
- write applications to remain portable between the MIDP 1.0 and MIDP 2.0 platforms

If a custom connection type is used other than sms, mms and cbs connections, that adapter specification MUST define permissions for use with that connection. If the sms, mms or cbs connection is reused in another adapter specification, then it MUST also reuse the sms, mms and cbs permissions defined in this section.

E.2.0 Security

To send and receive messages using this API, applications MUST be granted a permission to perform the requested operation. The mechanisms for granting a permission are implementation dependent.

E.2.1 Permissions for Opening Connections

The JSR 118 MIDP 2.0 specification defines a mechanism for granting permissions to use privileged features. This mechanism is based on a policy mechanism enforced in the platform implementation. The following permissions are defined for the JSR 205 messaging functionality, when deployed with a JSR 118 MIDP 2.0 implementation.

To open a connection, a MIDlet suite requires an appropriate permission to access the `MessageConnection` implementation. If the permission is not granted, then `Connector.open` methods MUST throw a `SecurityException`. The following table indicates the permission that must be granted for each protocol.

Table E-1: Permissions for Opening Connections

Permission	Protocol
`javax.microedition.io.Connector.sms`	`sms`
`javax.microedition.io.Connector.cbs`	`cbs`
`javax.microedition.io.Connector.mms`	`mms`

E.2.2 Permissions for Send and Receive Operations

To send and receive messages, the MIDlet suite requires the appropriate permissions. If the permission is not granted, then the `MessageConnection.send` and the `MessageConnection.receive` methods MUST throw a `SecurityException`. The following table indicates the permission that must be granted for each requested operation.

Table E-2: Permissions for Send and Receive Operations

Permission	Protocol
`javax.wireless.messaging.sms.send`	`sms`
`javax.wireless.messaging.sms.receive`	`sms`
`javax.wireless.messaging.cbs.receive`	`cbs`
`javax.wireless.messaging.mms.send`	`mms`
`javax.wireless.messaging.mms.receive`	`mms`

The ability for sending and receiving MAY depend on the type of messages and addresses being used. An implementation MAY restrict an application's ability to send some types of messages and/or sending messages to certain recipient addresses. These addresses can include device addresses and/or identifiers, such as port numbers, within a device.

An implementation MAY restrict certain types of messages or connection addresses, such that sending such messages will fail and throw a `SecurityException` even when the application has the permission to send messages in general.

The applications MUST NOT assume that successfully sending one message implies that they have the permission to send all kinds of messages to all addresses.

An application should handle `SecurityException` when a connection handle is provided from `Connector.open(url)` and for any message `receive` or `send` operation that potentially engages with the network or the privileged message storage on the device.

An Application MUST receive an user permission to send a MMS. The user makes a decision based on the following information provided by the application:

- List of all recipient addresses: all addresses set in the "to", "cc" and "bcc" fields
- The total size of the message, which includes the size of all message attachments and the size of the subject.

The presentation of this information to the user is up to the implementation.

E.3.0 WMA Push Capabilities

MIDP 2.0 includes a mechanism to register a MIDlet when a connection notification event is detected. Once the MIDlet has been launched it performs the same I/O operations it would normally use to open a connection and read and write data.

For WMA applications this capability allows the application to be launched if messages arrive either while the MIDlet is not running or while another MIDlet is running.

In order to perform a Push registration for a WMA connection the suite must request the permission to use the PushRegistry and the permission to open the connection. If the application will also perform read and write data operations, it must also request those permissions for access to the send and receive methods.

E.3.1 WMA Push Registration Entry

Push registrations are either defined in the application descriptor or made dynamically at runtime via `PushRegistry`. The entry for a WMA protocol will include the connection URL string which identifies the scheme and port number or application id of the inbound message connection. The entry also contains a filter field that designates which senders are permitted to send messages that launch the registered MIDlet. An asterisk ("*") and question mark ("?") can be used in the filter field as wild cards as specified in the MIDP 2.0 specification.

For the `sms:` protocol, the filter field is matched against the MSISDN part of the sender address, as defined by the `msisdn` element of the `sms:` URL syntax in section A.2.0 of the WMA API specification. The sender port number is not included in matching the filter. Wildcard characters can be used in the filter as specified in the MIDP 2.0 specification. The leading plus of a MSIDN is also matched between a filter and a sender address.

For the `cbs:` protocol, the filtering is not performed. The filter field is ignored. The filter may be set to "*" to indicate any sender is accepted.

For the `mms:` protocol, the filter field is matched against the from address, as defined by the `address_part` element of the `mms:` URL syntax in section D.2.0 of the WMA API specification. The sender application id is not included in matching the filter. Wildcard characters can be used in the filter as specified in the MIDP 2.0 specification.

For example :

```
MIDlet-Push-1: sms://:12345, SmsExample, 123456789
MIDlet-Push-2: cbs://:54321, CbsExample, *
MIDlet-Push-3: mms://:com.siemens.Messenger, Messenger, *
```

Unlike the initial push connections defined in JSR 118 for MIDP 2.0, the SMS protocol includes an explicit buffering mechanism where messages are held until processed by some application that reads and deletes messages when they are done with the data. If a message is delivered to the device and does not pass the specified filter, the message will be deleted by the Application Management Software.

When the application is started in response to a `Push` message, the application SHOULD read and process all messages that are buffered for it. If an application fails to read and process the messages when started or if starting of the application is denied (for example, by the end user), the platform implementation MAY delete unread messages from the buffer, if it becomes necessary to do so. For example, the platform implementation may delete messages when the buffer becomes full.

Another difference between the WMA interface and other JSR 118 protocol handlers in MIDP 2.0, is that WMA includes a `MessageListener` which provides asynchronous callbacks when messages become available while the application is running.

E.4.0 Portable WMA Applications

If permitted by the device security policy, a WMA application written for a MIDP 1.0 platform will work without any modification on a MIDP 2.0 system. This behavior is defined by the JSR 118 specification of untrusted applications.

MIDP 2.0 also supports the concept of trusted applications. For these applications, the device can automatically handle trust decisions based on signed JAR files and a platform-specific policy mechanism that associates specific permissions with the signed application.

The security model also allows for the definition of user-granted permissions on a one-shot, session or blanket authorization. In many cases, the platform-dependent policy for permissions on MIDP 1.0 will be able to be mapped onto the MIDP 2.0 defined permissions.

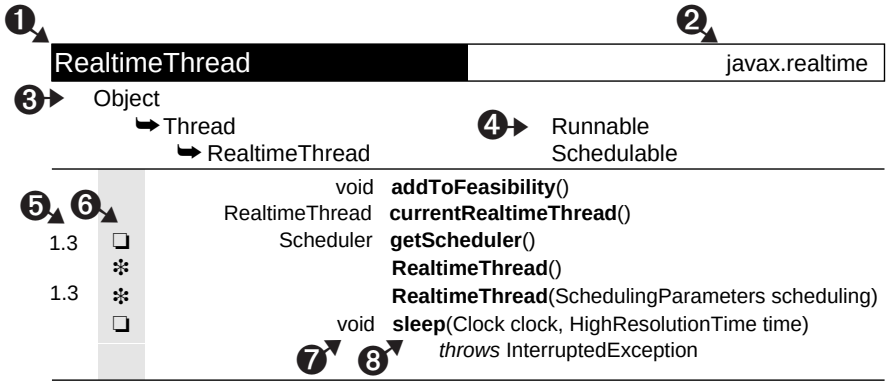
An application designed to work only on a MIDP 2.0 device can use the methods in the `PushRegistry` class to check if there are active connections (`listConnections`) or to add or remove registered connections at runtime (`registerConnection` or `unregisterConnection`).

An application designed to run portably on MIDP 1.0 or MIDP 2.0 platforms will only use the application descriptor and attributes in the manifest to describe requested permissions and push registration entries. See the JSR 118 MIDP 2.0 specification for details about the MIDlet-Permissions and MIDlet-Push-<n> attributes. On a MIDP 1.0 platforms these properties will be ignored. On a MIDP 2.0 platform, these properties will direct the application management software to perform the necessary checks and registrations when the application is installed and removed from the system.

ALMANAC LEGEND

The almanac presents classes and intefaces in alphabetic order, regardless of their package. Fields, methods and constructors are in alphabetic order in a single list.

This almanac is modeled after the style introduced by Patrick Chan in his excellent book *Java Developers Almanac*.



1. Name of the class, interface, nested class or nested interface. Interfaces are italic.
2. Name of the package containing the class or interface.
3. Inheritance hierarchy. In this example, `RealtimeThread` extends `Thread`, which extends `Object`.
4. Implemented interfaces. The interface is to the right of, and on the same line as, the class that implements it. In this example, `Thread` implements `Runnable`, and `RealtimeThread` implements `Schedulable`.
5. The first column above is for the value of the `@since` comment, which indicates the version in which the item was introduced.
6. The second column above is for the following icons. If the "protected" symbol does not appear, the member is public. (Private and package-private modifiers also have no symbols.) One symbol from each group can appear in this column.

Modifiers	**Access Modifiers**	**Constructors and Fields**
❍ abstract	♦ protected	❇ constructor
● final		✍ field
❑ static		
■ static final		

7. Return type of a method or declared type of a field. Blank for constructors.
8. Name of the constructor, field or method. Nested classes are listed in 1, not here.

Almanac

BinaryMessage — javax.wireless.messaging

BinaryMessage → Message

byte[]	**getPayloadData()**
void	**setPayloadData(byte[] data)**

Connector — javax.microedition.io

Object
➥Connector

❑	Connection	**open(String name) *throws* java.io.IOException**
❑	Connection	**open(String name, int mode) *throws* java.io.IOException**
❑	Connection	**open(String name, int mode, boolean timeouts) *throws* java.io.IOException**
❑	java.io.DataInputStream	**openDataInputStream(String name) *throws* java.io.IOException**
❑	java.io.DataOutputStream	**openDataOutputStream(String name) *throws* java.io.IOException**
❑	java.io.InputStream	**openInputStream(String name) *throws* java.io.IOException**
❑	java.io.OutputStream	**openOutputStream(String name) *throws* java.io.IOException**
✍■	int	**READ**
✍■	int	**READ_WRITE**
✍■	int	**WRITE**

Message — javax.wireless.messaging

Message

String	**getAddress()**
java.util.Date	**getTimestamp()**
void	**setAddress(String addr)**

MessageConnection — javax.wireless.messaging

MessageConnection → javax.microedition.io.Connection

	✍■ String	**BINARY_MESSAGE**
wma 2.0	✍■ String	**MULTIPART_MESSAGE**
	Message	**newMessage(String type)**
	Message	**newMessage(String type, String address)**
	int	**numberOfSegments(Message msg)**
	Message	**receive() *throws* java.io.IOException, java.io.InterruptedIOException**

	void	**send(Message msg) *throws* java.io.IOException, java.io.InterruptedIOException**
	void	**setMessageListener(MessageListener l) *throws* java.io.IOException**
✍■	String	**TEXT_MESSAGE**

MessageListener — javax.wireless.messaging

MessageListener

	void	**notifyIncomingMessage(MessageConnection conn)**

MessagePart — javax.wireless.messaging

Object
➥MessagePart

	byte[]	**getContent()**
	java.io.InputStream	**getContentAsStream()**
	String	**getContentID()**
	String	**getContentLocation()**
	String	**getEncoding()**
	int	**getLength()**
	String	**getMIMEType()**
❊		**MessagePart(byte[] contents, int offset, int length, String mimeType, String contentId, String contentLocation, String enc) *throws* SizeExceededException**
❊		**MessagePart(byte[] contents, String mimeType, String contentId, String contentLocation, String enc) *throws* SizeExceededException**
❊		**MessagePart(java.io.InputStream is, String mimeType, String contentId, String contentLocation, String enc) *throws* java.io.IOException, SizeExceededException**

MultipartMessage — javax.wireless.messaging

MultipartMessage — Message

	boolean	**addAddress(String type, String address)**
	void	**addMessagePart(MessagePart part) *throws* SizeExceededException**
	String	**getAddress()**
	String[]	**getAddresses(String type)**
	String	**getHeader(String headerField)**
	MessagePart	**getMessagePart(String contentID)**
	MessagePart[]	**getMessageParts()**
	String	**getStartContentId()**
	String	**getSubject()**
	boolean	**removeAddress(String type, String address)**
	void	**removeAddresses()**
	void	**removeAddresses(String type)**
	boolean	**removeMessagePart(MessagePart part)**
	boolean	**removeMessagePartId(String contentID)**
	boolean	**removeMessagePartLocation(String contentLocation)**
	void	**setAddress(String addr)**

void	**setHeader(String headerField, String headerValue)**
void	**setStartContentId(String contentId)**
void	**setSubject(String subject)**

SizeExceededException — javax.wireless.messaging

Object
- ➥Throwable
  - ➥Exception
    - ➥java.io.IOException
      - ➥SizeExceededException

❇	**SizeExceededException(String reason)**

TextMessage — javax.wireless.messaging

TextMessage — Message

String	**getPayloadText()**
void	**setPayloadText(String data)**