Wireless Messaging API (WMA)

for Java™ 2 Micro Edition

Version 1.1

JSR 120 Expert Group
JSR-120-EG@JCP.ORG

Java Community Process (JCP)
Contents

Preface ................................................................................................................................. v

1 Overview ........................................................................................................................ 1

2 javax.microedition.io .................................................................................................. 5
   Connector ......................................................................................................................... 6

3 javax.wireless.messaging ............................................................................................ 11
   BinaryMessage ............................................................................................................... 13
   Message .......................................................................................................................... 15
   MessageConnection ....................................................................................................... 17
   MessageListener ............................................................................................................ 22
   TextMessage .................................................................................................................. 25

A GSM SMS Adapter ...................................................................................................... 27

B GSM Cell Broadcast Adapter ..................................................................................... 35

C CDMA IS-637 SMS Adapter ......................................................................................... 37

D Deploying JSR 120 Interfaces on a MIDP 2.0 Platform ............................................. 39
   Almanac ......................................................................................................................... 45

Index ................................................................................................................................. 47
Preface

This book provides information on the messaging API which is included in the JSR 120 Wireless Messaging API (WMA) specification. It also describes Sun Microsystem’s reference implementation (RI) of the API.

Who Should Use This Book
This book is intended primarily for those individuals and companies who want to implement WMA, or to port the WMA RI to a new platform.

Before You Read This Book
This book assumes that you have experience programming in the C and Java™ languages, and that you have experience with the platforms to which you are porting the RI. 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


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


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)

Related Documentation

Preface


Terms, Acronyms, and Abbreviations Used in this Book
SMS - Short Message Service
URL - Uniform Resource Locator

Typographic Conventions

<table>
<thead>
<tr>
<th>Typeface</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>AaBbCc123</td>
<td>The names of commands, files, and directories; on-screen computer output</td>
<td>Edit your .login file. Use ls -a to list all files. % You have mail.</td>
</tr>
</tbody>
</table>
| 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. |

Accessing Sun Documentation Online
The docs.sun.com web site enables you to access Sun technical documentation on the Web. You can browse the docs.sun.com archive or search for a specific book title or subject at:

http://docs.sun.com

Sun Welcomes Your Comments
We are interested in improving our documentation and welcome your comments and suggestions. You can email your comments to us at:

wma-comments@sun.com
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 both text and binary messages. These are represented by two subinterfaces of Message: TextMessage and BinaryMessage. These subinterfaces provide ways to manipulate the payload of the message as Strings and byte arrays, 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 and BinaryMessage 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 receive 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, they 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

The appendices of this specification include the definition of SMS and CBS 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 and CBS adapters 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.

**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` 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 JSR120 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.

**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.

```java
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();
            // 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 package includes the platform networking interfaces which have been modified for use on platforms that support message connections.
Package
javax.microedition.io

Description
This package 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 interpreted as a SecurityException.

Since: MIDP2.0

<table>
<thead>
<tr>
<th>Class Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interfaces</td>
</tr>
<tr>
<td>Classes</td>
</tr>
<tr>
<td>Connector</td>
</tr>
<tr>
<td>Exceptions</td>
</tr>
</tbody>
</table>
javax.microedition.io

Connector

Declaration

public class Connector

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

Description

This class is factory 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.
- **target** is normally some kind of network address.
- **parms** are formed as a series of equates of the form ;x=y. For 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 example, 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, convenience functions are provided for this purpose. See also: DatagramConnection for information relating to datagram addressing

Since: CLDC 1.0

<table>
<thead>
<tr>
<th>Member Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fields</strong></td>
</tr>
<tr>
<td>static int READ</td>
</tr>
<tr>
<td>static int READ_WRITE</td>
</tr>
<tr>
<td>static int WRITE</td>
</tr>
</tbody>
</table>
**Member Summary**

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>static Connection open(java.lang.String name)</td>
</tr>
<tr>
<td>static Connection open(java.lang.String name, int mode)</td>
</tr>
<tr>
<td>static Connection open(java.lang.String name, int mode, boolean timeouts)</td>
</tr>
<tr>
<td>static java.io.DataInputStream openDataInputStream(java.lang.String name)</td>
</tr>
<tr>
<td>static java.io.DataOutputStream openDataOutputStream(java.lang.String name)</td>
</tr>
<tr>
<td>static java.io.InputStream openInputStream(java.lang.String name)</td>
</tr>
<tr>
<td>static java.io.OutputStream openOutputStream(java.lang.String name)</td>
</tr>
</tbody>
</table>

**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.

**READ_WRITE**

Declaration:
public static final int READ_WRITE

Description:
Access mode READ_WRITE.

**WRITE**

Declaration:
public static final int WRITE

Description:
Access mode WRITE.
Connector
javax.microedition.io
open(String)

Methods

open(String)

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

Description:
Creates and opens 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:
Creates and opens 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:
Creates and opens a Connection.
openDataInputStream(String)

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

**Description:**
Creates and opens 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 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 access to the requested stream is not permitted

openDataOutputStream(String)

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

**Description:**
 Creates and opens 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:**
Creates and opens 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:**
Creates and opens 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
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, 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.
javax.wireless.messaging

Since: WMA 1.0

### Class Summary

<table>
<thead>
<tr>
<th>Interfaces</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryMessage</td>
<td>An interface representing a binary message.</td>
</tr>
<tr>
<td>Message</td>
<td>This is the base interface for derived interfaces that represent various types of messages.</td>
</tr>
<tr>
<td>MessageConnection</td>
<td>The MessageConnection interface defines the basic functionality for sending and receiving messages.</td>
</tr>
<tr>
<td>MessageListener</td>
<td>The MessageListener interface provides a mechanism for the application to be notified of incoming messages.</td>
</tr>
<tr>
<td>TextMessage</td>
<td>An interface representing a text message.</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte[] getPayloadData()</td>
</tr>
<tr>
<td>void setPayloadData(byte[] data)</td>
</tr>
</tbody>
</table>

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[])
BinaryMessage

javax.wireless.messaging

setPayloadData(byte[])  

setPayloadData(byte[])

**Declaration:**

```java
generic 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, 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 Datagrams 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

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.String getAddress</td>
</tr>
<tr>
<td>java.util.Date getTimestamp</td>
</tr>
<tr>
<td>void setAddress(String addr)</td>
</tr>
</tbody>
</table>

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) 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 MessageConnections 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 TEXT_MESSAGE

Methods
- Message newMessage(java.lang.String type)
- Message newMessage(java.lang.String type, java.lang.String address)
### BINARY_MESSAGE

**Declaration:**
```java
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.

### TEXT_MESSAGE

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

**Description:**
Constructs a new message object of a given type. When the string text is passed in, the created object implements the TextMessage interface. When the binary constant is passed in, the created object implements the BinaryMessage interface. Adapter definitions for messaging protocols can define new constants and new subinterfaces for the Messages. 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 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 parameter is not equal to the value of
  `TEXT_MESSAGE`, `BINARY_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:
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 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` or any other type value specified in a private or publicly
  standardized adapter specification that is supported by the implementation

See Also: `newMessage(String)`

decimalOfSegments(Message)

Declaration:
public int decimalOfSegments(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 Messages 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 or because of 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.

**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
**MessageListener**
javax.wireless.messaging

**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 signalling threads to terminate when the MIDlet is paused or destroyed.
javax.wireless.messaging
MessageListener
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;
            public void run() {
                // 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
                                }
                            }
                        }
MessageListener javax.wireless.messaging
notifyIncomingMessage(MessageConnection)

```java
    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**

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>void notifyIncomingMessage(MessageConnection conn)</td>
</tr>
</tbody>
</table>

**Methods**

**notifyIncomingMessage(MessageConnection)**

**Declaration:**

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

**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

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 Strings encoded in Java. The underlying implementation will convert the Strings 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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.String getPayloadText()</td>
<td></td>
</tr>
<tr>
<td>void setPayloadText(java.lang.String data)</td>
<td></td>
</tr>
</tbody>
</table>

Inherited Member Summary

Methods inherited from interface Message

getAddress(), getTimestamp(), setAddress(String)

Methods

getPayloadText()
**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()`
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 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 1: Number of SMS protocol messages needed for different payload lengths**

<table>
<thead>
<tr>
<th>Optional Headers Encoding</th>
<th>No port number present (message to be displayed to the end user)</th>
<th>Port number present (message targeted at an application)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Length</td>
<td>SMS messages</td>
</tr>
<tr>
<td>GSM 7-bit alphabet</td>
<td>0-160 chars</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>161-304 chars</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>305-456 chars</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0-140 bytes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>41-266 bytes</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>267-399 bytes</td>
<td>3</td>
</tr>
<tr>
<td>8-bit binary data</td>
<td>0-70 chars</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>71-132 chars</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>133-198 chars</td>
<td>3</td>
</tr>
<tr>
<td>UCS-2 alphabet</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

**Character Mapping**

<table>
<thead>
<tr>
<th>GSM 7-bit</th>
<th>UCS-2</th>
<th>Character name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x0040</td>
<td>COMMERCIAL AT</td>
</tr>
<tr>
<td>0x01</td>
<td>0x00a3</td>
<td>POUND SIGN</td>
</tr>
<tr>
<td>0x02</td>
<td>0x0024</td>
<td>DOLLAR SIGN</td>
</tr>
<tr>
<td>0x03</td>
<td>0x00a5</td>
<td>YEN SIGN</td>
</tr>
<tr>
<td>0x04</td>
<td>0x00e8</td>
<td>LATIN SMALL LETTER E WITH GRAVE</td>
</tr>
<tr>
<td>0x05</td>
<td>0x00e9</td>
<td>LATIN SMALL LETTER E WITH ACUTE</td>
</tr>
<tr>
<td>GSM 7-bit</td>
<td>UCS-2</td>
<td>Character name</td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
<td>----------------------------------------------------</td>
</tr>
<tr>
<td>0x06</td>
<td>0x00f9</td>
<td>LATIN SMALL LETTER U WITH GRAVE</td>
</tr>
<tr>
<td>0x07</td>
<td>0x00ec</td>
<td>LATIN SMALL LETTER I WITH GRAVE</td>
</tr>
<tr>
<td>0x08</td>
<td>0x00f2</td>
<td>LATIN SMALL LETTER O WITH GRAVE</td>
</tr>
<tr>
<td>0x09</td>
<td>0x00c7</td>
<td>LATIN CAPITAL LETTER C WITH CEDILLA</td>
</tr>
<tr>
<td>0xa</td>
<td>0x000a</td>
<td>control: line feed</td>
</tr>
<tr>
<td>0xb</td>
<td>0x00d8</td>
<td>LATIN CAPITAL LETTER O WITH STROKE</td>
</tr>
<tr>
<td>0xc</td>
<td>0x00f8</td>
<td>LATIN SMALL LETTER O WITH STROKE</td>
</tr>
<tr>
<td>0xd</td>
<td>0x000d</td>
<td>control: carriage return</td>
</tr>
<tr>
<td>0xe</td>
<td>0x00c5</td>
<td>LATIN CAPITAL LETTER A WITH RING ABOVE</td>
</tr>
<tr>
<td>0xf</td>
<td>0x00e5</td>
<td>LATIN SMALL LETTER A WITH RING ABOVE</td>
</tr>
<tr>
<td>0x10</td>
<td>0x0394</td>
<td>GREEK CAPITAL LETTER DELTA</td>
</tr>
<tr>
<td>0x11</td>
<td>0x005f</td>
<td>LOW LINE</td>
</tr>
<tr>
<td>0x12</td>
<td>0x03a6</td>
<td>GREEK CAPITAL LETTER PHI</td>
</tr>
<tr>
<td>0x13</td>
<td>0x0393</td>
<td>GREEK CAPITAL LETTER GAMMA</td>
</tr>
<tr>
<td>0x14</td>
<td>0x039b</td>
<td>GREEK CAPITAL LETTER LAMDA</td>
</tr>
<tr>
<td>0x15</td>
<td>0x03a9</td>
<td>GREEK CAPITAL LETTER OMEGA</td>
</tr>
<tr>
<td>0x16</td>
<td>0x03a0</td>
<td>GREEK CAPITAL LETTER PI</td>
</tr>
<tr>
<td>0x17</td>
<td>0x03a8</td>
<td>GREEK CAPITAL LETTER PSI</td>
</tr>
<tr>
<td>0x18</td>
<td>0x03a3</td>
<td>GREEK CAPITAL LETTER SIGMA</td>
</tr>
<tr>
<td>0x19</td>
<td>0x0398</td>
<td>GREEK CAPITAL LETTER THETA</td>
</tr>
<tr>
<td>0x1a</td>
<td>0x039e</td>
<td>GREEK CAPITAL LETTER XI</td>
</tr>
<tr>
<td>0x1b</td>
<td>xxxx</td>
<td>escape to extension table</td>
</tr>
<tr>
<td>0x1c</td>
<td>0x00c6</td>
<td>LATIN CAPITAL LETTER AE</td>
</tr>
<tr>
<td>0x1d</td>
<td>0x00e6</td>
<td>LATIN SMALL LETTER AE</td>
</tr>
<tr>
<td>0x1e</td>
<td>0x00df</td>
<td>LATIN SMALL LETTER SHARP S</td>
</tr>
<tr>
<td>0x1f</td>
<td>0x00e9</td>
<td>LATIN CAPITAL LETTER E WITH ACUTE</td>
</tr>
<tr>
<td>0x20</td>
<td>0x0020</td>
<td>SPACE</td>
</tr>
<tr>
<td>0x21</td>
<td>0x0021</td>
<td>EXCLAMATION MARK</td>
</tr>
<tr>
<td>0x22</td>
<td>0x0022</td>
<td>QUOTATION MARK</td>
</tr>
<tr>
<td>0x23</td>
<td>0x0023</td>
<td>NUMBER SIGN</td>
</tr>
<tr>
<td>0x24</td>
<td>0x00a4</td>
<td>CURRENCY SIGN</td>
</tr>
<tr>
<td>0x25</td>
<td>0x0025</td>
<td>PERCENT SIGN</td>
</tr>
<tr>
<td>0x26</td>
<td>0x0026</td>
<td>AMPERSDAND</td>
</tr>
<tr>
<td>0x27</td>
<td>0x0027</td>
<td>APOSTROPE</td>
</tr>
<tr>
<td>0x28</td>
<td>0x0028</td>
<td>LEFT PARENTHESES</td>
</tr>
<tr>
<td>0x29</td>
<td>0x0029</td>
<td>RIGHT PARENTHESES</td>
</tr>
<tr>
<td>0x2a</td>
<td>0x002a</td>
<td>ASTERISK</td>
</tr>
<tr>
<td>0x2b</td>
<td>0x002b</td>
<td>PLUS SIGN</td>
</tr>
<tr>
<td>0x2c</td>
<td>0x002c</td>
<td>COMMA</td>
</tr>
<tr>
<td>0x2d</td>
<td>0x002d</td>
<td>HYPHEN-MINUS</td>
</tr>
<tr>
<td>0x2e</td>
<td>0x002e</td>
<td>FULL STOP</td>
</tr>
<tr>
<td>0x2f</td>
<td>0x002f</td>
<td>SOLIDUS</td>
</tr>
<tr>
<td>0x30</td>
<td>0x0030</td>
<td>DIGIT ZERO</td>
</tr>
<tr>
<td>0x31</td>
<td>0x0031</td>
<td>DIGIT ONE</td>
</tr>
<tr>
<td>0x32</td>
<td>0x0032</td>
<td>DIGIT TWO</td>
</tr>
<tr>
<td>0x33</td>
<td>0x0033</td>
<td>DIGIT THREE</td>
</tr>
<tr>
<td>0x34</td>
<td>0x0034</td>
<td>DIGIT FOUR</td>
</tr>
<tr>
<td>0x35</td>
<td>0x0035</td>
<td>DIGIT FIVE</td>
</tr>
<tr>
<td>0x36</td>
<td>0x0036</td>
<td>DIGIT SIX</td>
</tr>
<tr>
<td>0x37</td>
<td>0x0037</td>
<td>DIGIT SEVEN</td>
</tr>
<tr>
<td>GSM 7-bit</td>
<td>UCS-2</td>
<td>Character name</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>0x38</td>
<td>0x0038</td>
<td>DIGIT EIGHT</td>
</tr>
<tr>
<td>0x39</td>
<td>0x0039</td>
<td>DIGIT NINE</td>
</tr>
<tr>
<td>0x3a</td>
<td>0x003a</td>
<td>COLON</td>
</tr>
<tr>
<td>0x3b</td>
<td>0x003b</td>
<td>SEMICOLON</td>
</tr>
<tr>
<td>0x3c</td>
<td>0x003c</td>
<td>LESS-THAN SIGN</td>
</tr>
<tr>
<td>0x3d</td>
<td>0x003d</td>
<td>EQUALS SIGN</td>
</tr>
<tr>
<td>0x3e</td>
<td>0x003e</td>
<td>GREATER-TAN SIGN</td>
</tr>
<tr>
<td>0x3f</td>
<td>0x003f</td>
<td>QUESTION MARK</td>
</tr>
<tr>
<td>0x40</td>
<td>0x00a1</td>
<td>INVERTED EXCLAMATION MARK</td>
</tr>
<tr>
<td>0x41</td>
<td>0x0041</td>
<td>LATIN CAPITAL LETTER A</td>
</tr>
<tr>
<td>0x42</td>
<td>0x0042</td>
<td>LATIN CAPITAL LETTER B</td>
</tr>
<tr>
<td>0x43</td>
<td>0x0043</td>
<td>LATIN CAPITAL LETTER C</td>
</tr>
<tr>
<td>0x44</td>
<td>0x0044</td>
<td>LATIN CAPITAL LETTER D</td>
</tr>
<tr>
<td>0x45</td>
<td>0x0045</td>
<td>LATIN CAPITAL LETTER E</td>
</tr>
<tr>
<td>0x46</td>
<td>0x0046</td>
<td>LATIN CAPITAL LETTER F</td>
</tr>
<tr>
<td>0x47</td>
<td>0x0047</td>
<td>LATIN CAPITAL LETTER G</td>
</tr>
<tr>
<td>0x48</td>
<td>0x0048</td>
<td>LATIN CAPITAL LETTER H</td>
</tr>
<tr>
<td>0x49</td>
<td>0x0049</td>
<td>LATIN CAPITAL LETTER I</td>
</tr>
<tr>
<td>0x4a</td>
<td>0x004a</td>
<td>LATIN CAPITAL LETTER J</td>
</tr>
<tr>
<td>0x4b</td>
<td>0x004b</td>
<td>LATIN CAPITAL LETTER K</td>
</tr>
<tr>
<td>0x4c</td>
<td>0x004c</td>
<td>LATIN CAPITAL LETTER L</td>
</tr>
<tr>
<td>0x4d</td>
<td>0x004d</td>
<td>LATIN CAPITAL LETTER M</td>
</tr>
<tr>
<td>0x4e</td>
<td>0x004e</td>
<td>LATIN CAPITAL LETTER N</td>
</tr>
<tr>
<td>0x4f</td>
<td>0x004f</td>
<td>LATIN CAPITAL LETTER O</td>
</tr>
<tr>
<td>0x50</td>
<td>0x0050</td>
<td>LATIN CAPITAL LETTER P</td>
</tr>
<tr>
<td>0x51</td>
<td>0x0051</td>
<td>LATIN CAPITAL LETTER Q</td>
</tr>
<tr>
<td>0x52</td>
<td>0x0052</td>
<td>LATIN CAPITAL LETTER R</td>
</tr>
<tr>
<td>0x53</td>
<td>0x0053</td>
<td>LATIN CAPITAL LETTER S</td>
</tr>
<tr>
<td>0x54</td>
<td>0x0054</td>
<td>LATIN CAPITAL LETTER T</td>
</tr>
<tr>
<td>0x55</td>
<td>0x0055</td>
<td>LATIN CAPITAL LETTER U</td>
</tr>
<tr>
<td>0x56</td>
<td>0x0056</td>
<td>LATIN CAPITAL LETTER V</td>
</tr>
<tr>
<td>0x57</td>
<td>0x0057</td>
<td>LATIN CAPITAL LETTER W</td>
</tr>
<tr>
<td>0x58</td>
<td>0x0058</td>
<td>LATIN CAPITAL LETTER X</td>
</tr>
<tr>
<td>0x59</td>
<td>0x0059</td>
<td>LATIN CAPITAL LETTER Y</td>
</tr>
<tr>
<td>0x5a</td>
<td>0x005a</td>
<td>LATIN CAPITAL LETTER Z</td>
</tr>
<tr>
<td>0x5b</td>
<td>0x00c4</td>
<td>LATIN CAPITAL LETTER A WITH DIARESIS</td>
</tr>
<tr>
<td>0x5c</td>
<td>0x00d6</td>
<td>LATIN CAPITAL LETTER O WITH DIARESIS</td>
</tr>
<tr>
<td>0x5d</td>
<td>0x00d1</td>
<td>LATIN CAPITAL LETTER N WITH TILDE</td>
</tr>
<tr>
<td>0x5e</td>
<td>0x00dc</td>
<td>LATIN CAPITAL LETTER U WITH DIARESIS</td>
</tr>
<tr>
<td>0x5f</td>
<td>0x00a7</td>
<td>SECTION SIGN</td>
</tr>
<tr>
<td>0x60</td>
<td>0x00bf</td>
<td>INVERTED QUESTION MARK</td>
</tr>
<tr>
<td>0x61</td>
<td>0x0061</td>
<td>LATIN SMALL LETTER A</td>
</tr>
<tr>
<td>0x62</td>
<td>0x0062</td>
<td>LATIN SMALL LETTER B</td>
</tr>
<tr>
<td>0x63</td>
<td>0x0063</td>
<td>LATIN SMALL LETTER C</td>
</tr>
<tr>
<td>0x64</td>
<td>0x0064</td>
<td>LATIN SMALL LETTER D</td>
</tr>
<tr>
<td>0x65</td>
<td>0x0065</td>
<td>LATIN SMALL LETTER E</td>
</tr>
<tr>
<td>0x66</td>
<td>0x0066</td>
<td>LATIN SMALL LETTER F</td>
</tr>
<tr>
<td>0x67</td>
<td>0x0067</td>
<td>LATIN SMALL LETTER G</td>
</tr>
<tr>
<td>0x68</td>
<td>0x0068</td>
<td>LATIN SMALL LETTER H</td>
</tr>
<tr>
<td>0x69</td>
<td>0x0069</td>
<td>LATIN SMALL LETTER I</td>
</tr>
</tbody>
</table>
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 2.

#### Table 2: Connection Strings for Message Addresses

<table>
<thead>
<tr>
<th>String</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>smsurl</td>
<td>::= &quot;sms://&quot; address_part</td>
</tr>
<tr>
<td>address_part</td>
<td>::= foreign_host_address</td>
</tr>
<tr>
<td>local_host_address</td>
<td>::= port_number_part</td>
</tr>
<tr>
<td>port_number_part</td>
<td>::= &quot;;&quot; digits</td>
</tr>
</tbody>
</table>
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`. 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 `MessageConnections` 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 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 3.

Table 3: Property Name and Description for SMSC Addresses

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wireless.messaging.sms.smsc</td>
<td>The address of the SMS expressed using the syntax expressed by the msisdn item of the following BNF definition:</td>
</tr>
<tr>
<td>digit ::= &quot;0&quot;</td>
<td>&quot;1&quot;</td>
</tr>
<tr>
<td>digits ::= digit</td>
<td>digit digits</td>
</tr>
</tbody>
</table>

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 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 4: Port Numbers Restricted to SMS Messages

<table>
<thead>
<tr>
<th>Port number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2805</td>
<td>WAP WTA secure connection-less session service</td>
</tr>
<tr>
<td>2923</td>
<td>WAP WTA secure session service</td>
</tr>
<tr>
<td>2948</td>
<td>WAP Push connectionless session service (client side)</td>
</tr>
<tr>
<td>2949</td>
<td>WAP Push secure connectionless session service (client side)</td>
</tr>
<tr>
<td>5502</td>
<td>Service Card reader</td>
</tr>
<tr>
<td>5503</td>
<td>Internet access configuration reader</td>
</tr>
<tr>
<td>5508</td>
<td>Dynamic Menu Control Protocol</td>
</tr>
<tr>
<td>5511</td>
<td>Message Access Protocol</td>
</tr>
<tr>
<td>5512</td>
<td>Simple Email Notification</td>
</tr>
<tr>
<td>9200</td>
<td>WAP connectionless session service</td>
</tr>
<tr>
<td>9201</td>
<td>WAP session service</td>
</tr>
<tr>
<td>9202</td>
<td>WAP secure connectionless session service</td>
</tr>
<tr>
<td>9203</td>
<td>WAP secure session service</td>
</tr>
<tr>
<td>9207</td>
<td>WAP vCal Secure</td>
</tr>
<tr>
<td>49996</td>
<td>SyncML OTA configuration</td>
</tr>
<tr>
<td>49999</td>
<td>WAP OTA configuration</td>
</tr>
</tbody>
</table>
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**

The URL connection strings that specify the address use the following syntax:

<table>
<thead>
<tr>
<th>String</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbsurl</td>
<td>::= &quot;cbs://&quot; address_part</td>
</tr>
<tr>
<td>address_part</td>
<td>::= message_identifier_part</td>
</tr>
<tr>
<td>message_identifier_part</td>
<td>::= &quot;:&quot; digits</td>
</tr>
<tr>
<td>digit</td>
<td>::= &quot;0&quot;</td>
</tr>
<tr>
<td>digits</td>
<td>::= digit</td>
</tr>
</tbody>
</table>


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.
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.
Deploying JSR 120 Interfaces on a MIDP 2.0 Platform

D.1.0 Introduction

This section provides implementation notes for platform developers deploying the JSR 120 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 and CBS messages
- write applications to remain portable between the MIDP 1.0 and MIDP 2.0 platforms

D.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.

D.2.1 Permissions for Opening Connections

The JSR 118 MIDP NG 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 120 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>
<thead>
<tr>
<th>Permission</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.microedition.io.Connector.</td>
<td>sms</td>
</tr>
<tr>
<td>sms</td>
<td></td>
</tr>
</tbody>
</table>
### D.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>
<thead>
<tr>
<th>Permission</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>javax.microedition.io.Connector</code></td>
<td><code>cbs</code></td>
</tr>
<tr>
<td><code>javax.wireless.messaging.sms.send</code></td>
<td><code>sms</code></td>
</tr>
<tr>
<td><code>javax.wireless.messaging.sms.receive</code></td>
<td><code>sms</code></td>
</tr>
<tr>
<td><code>javax.wireless.messaging.cbs.receive</code></td>
<td><code>cbs</code></td>
</tr>
</tbody>
</table>

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 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 `SecurityExceptions` 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.

### D.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.

#### D.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 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 a 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.

For the CBS: protocol, the filtering is not performed and only "*" MUST be used as the filter.

For example:

```
MIDlet-Push-1: sms://:12345, SmsExample, 123456789
MIDlet-Push-2: cbs://:54321, CbsExample, *
```

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 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.

### D.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 interfaces 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*.

<table>
<thead>
<tr>
<th>Name</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>RealtimeThread</td>
<td>javax.realtime</td>
</tr>
<tr>
<td>Object</td>
<td></td>
</tr>
<tr>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Runnable</td>
<td></td>
</tr>
<tr>
<td>Schedulable</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Modifiers</th>
<th>Access Modifiers</th>
<th>Constructors and Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>protected</td>
<td>constructor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>field</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte[] getPayloadData()</td>
<td>Get payload data</td>
</tr>
<tr>
<td>void setPayloadData(byte[] data)</td>
<td>Set payload data</td>
</tr>
</tbody>
</table>

### Connector

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection open(String name)</td>
<td>Open connection with name</td>
</tr>
<tr>
<td>Connection open(String name, int mode)</td>
<td>Open connection with name and mode</td>
</tr>
<tr>
<td>Connection open(String name, int mode, boolean timeouts)</td>
<td>Open connection with name, mode, and timeouts</td>
</tr>
<tr>
<td>java.io.DataInputStream openDataInputStream(String name)</td>
<td>Open data input stream</td>
</tr>
<tr>
<td>java.io.DataOutputStream openDataOutputStream(String name)</td>
<td>Open data output stream</td>
</tr>
<tr>
<td>java.io.InputStream openInputStream(String name)</td>
<td>Open input stream</td>
</tr>
<tr>
<td>java.io.OutputStream openOutputStream(String name)</td>
<td>Open output stream</td>
</tr>
<tr>
<td>int READ</td>
<td>Connection mode: READ</td>
</tr>
<tr>
<td>int READ_WRITE</td>
<td>Connection mode: READ_WRITE</td>
</tr>
<tr>
<td>int WRITE</td>
<td>Connection mode: WRITE</td>
</tr>
</tbody>
</table>

### Message

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String getAddress()</td>
<td>Get address</td>
</tr>
<tr>
<td>java.util.Date getTimestamp()</td>
<td>Get timestamp</td>
</tr>
<tr>
<td>void setAddress(String addr)</td>
<td>Set address</td>
</tr>
</tbody>
</table>

### MessageConnection

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String BINARY_MESSAGE</td>
<td>Message type: BINARY_MESSAGE</td>
</tr>
<tr>
<td>Message newMessage(String type)</td>
<td>Create new message with type</td>
</tr>
<tr>
<td>Message newMessage(String type, String address)</td>
<td>Create new message with type and address</td>
</tr>
<tr>
<td>int numberOfSegments(Message msg)</td>
<td>Get number of segments</td>
</tr>
<tr>
<td>Message receive()</td>
<td>Receive message with IOException, and InterruptedIOException</td>
</tr>
</tbody>
</table>
### MessageListener

*javax.wireless.messaging*

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void send(Message msg) throws IOException, InterruptedIOException</td>
<td></td>
</tr>
<tr>
<td>void setMessageListener(MessageListener l) throws IOException</td>
<td></td>
</tr>
<tr>
<td>String TEXT_MESSAGE</td>
<td></td>
</tr>
</tbody>
</table>

### TextMessage

*javax.wireless.messaging*

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String getPayloadText()</td>
<td></td>
</tr>
<tr>
<td>void setPayloadText(String data)</td>
<td></td>
</tr>
<tr>
<td>void notifyIncomingMessage(MessageConnection conn)</td>
<td></td>
</tr>
</tbody>
</table>
Index

B

BINARY_MESSAGE
  of javax.wireless.messaging.MessageConnection 18

BinaryMessage
  of javax.wireless.messaging 13

C

Connector
  of javax.microedition.io 6

G

getAddress()
  of javax.wireless.messaging.Message 15
getPayloadData()
  of javax.wireless.messaging.BinaryMessage 13
getPayloadText()
  of javax.wireless.messaging.TextMessage 25
getTimestamp()
  of javax.wireless.messaging.Message 16

M

Message
  of javax.wireless.messaging 15

MessageConnection
  of javax.wireless.messaging 17

MessageListener
  of javax.wireless.messaging 22

N

newMessage(String)
  of javax.wireless.messaging.MessageConnection 18

newMessage(String, String)
  of javax.wireless.messaging.MessageConnection 19

notifyIncomingMessage(MessageConnection)
  of javax.wireless.messaging.MessageListener 24

numberOfSegments(Message)
  of javax.wireless.messaging.MessageConnection 19

O

open(String)
  of javax.microedition.io.Connector 8

open(String, int)
  of javax.microedition.io.Connector 8

open(String, int, boolean)
  of javax.microedition.io.Connector 8

openDataInputStream(String)
  of javax.microedition.io.Connector 9

openDataOutputStream(String)
  of javax.microedition.io.Connector 9

openInputStream(String)
  of javax.microedition.io.Connector 10

openOutputStream(String)
  of javax.microedition.io.Connector 10

R

READ
  of javax.microedition.io.Connector 7

READ_WRITE
  of javax.microedition.io.Connector 7

receive()
  of javax.wireless.messaging.MessageConnection 20

S

send(Message)
  of javax.wireless.messaging.MessageConnection 20

setAddress(String)
  of javax.wireless.messaging.Message 16

setMessageListener(MessageListener)
  of javax.wireless.messaging.MessageConnection 21

setPayloadData(byte[])
  of javax.wireless.messaging.BinaryMessage 14

setPayloadText(String)
  of javax.wireless.messaging.TextMessage 26

T

TEXT_MESSAGE
  of javax.wireless.messaging.MessageConnection 18

TextMessage
  of javax.wireless.messaging 25
Index

W

WRITE
  of javax.microedition.io.Connector 7