

A MIDlet Example Using the Wireless Messaging API and the Nokia SMS API: Chat

Version 1.0; January 31, 2003

Java

NOKIA

Contents

1	Introduction	4
1.1	Purpose	4
1.2	Comparison of WMA and the Nokia SMS API	5
2	ChatMIDlet	6
2.1	User Interface	6
2.2	Design	8
2.3	ChatMIDlet.java	9
2.4	ChatScreen.java	16
2.5	SmsHandler.java	18
2.6	NokiaSmsHandler.java	21
2.7	WmaSmsHandler.java	23
2.8	WriteMessageScreen.java	26
2.9	PeerScreen.java	27
2.10	PeerList.java	28
2.11	PhoneBook.java	31
2.12	InputNameScreen.java	35
2.13	ErrorScreen.java	36
2.14	IncomingTextMessageListener.java	36
2.15	OutgoingTextMessageListener.java	37
2.16	OutgoingTextMessage.java	37
2.17	Chat.jad	38
3	References	39

Change History

31 January 2003	V1.0	Initial document release.
-----------------	------	---------------------------

Disclaimer

The information in this document is provided "as is," with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Furthermore, information provided in this document is preliminary, and may be changed substantially prior to final release. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this specification at any time, without notice.

The phone UI images shown in this document are for illustrative purposes and do not represent any real device.

Copyright © 2003 Nokia Corporation.

Nokia and Nokia Connecting People are registered trademarks of Nokia Corporation.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc.

Other product and company names mentioned herein may be trademarks or trade names of their respective owners.

License

A license is hereby granted to download and print a copy of this specification for personal use only. No other license to any other intellectual property rights is granted herein.

A MIDlet Example Using the Wireless Messaging API and the Nokia SMS API: Chat

Version 1.0; January 31, 2003

1 Introduction

1.1 Purpose

The Wireless Messaging API (WMA) defined in JSR120 is a set of optional APIs that provide standard access to wireless communication resources [WMA]. For example, the Wireless Messaging API can be used to develop networked MIDlets that send and receive SMS (Short Message Service) [GSMSMS] messages. The Nokia SMS API [SMSAPI] is a vendor-specific API that provides functionality similar to WMA. The following document briefly describes how to use both messaging APIs through a simple MIDlet example called Chat.

This document assumes that you are familiar with Java programming and know the basics of MIDP programming, for instance by having read the Forum Nokia paper *A Brief Introduction to MIDP Programming* [MIDPPROG]. It also assumes that you have some knowledge of the WMA and Nokia SMS API by having read JSR 120 and Nokia SMS API JavaDoc documentation [SMSAPI]. It also assumes that you are familiar with technical aspects of sending SMS messages and with binary and text SMS messages, for example by having read [GSMSMS] and [SMSALPHABETS].

After reading this document, you will better understand how to use the two messaging APIs for:

- Setting up both client and server mode SMS message connections
- Creating new SMS messages
- Sending SMS messages in client mode
- Receiving and replying to SMS messages in server mode

1.2 Comparison of WMA and the Nokia SMS API

The Nokia SMS API pre-dates the public release of the Wireless Messaging API, but is functionally very similar. The Nokia SMS API is available in the Nokia 3410 device. In future MIDP Java phones, when a wireless messaging API is included WMA will be supported instead of the Nokia SMS API. You should use WMA in your MIDlets in devices where it is available.

The differences between the two APIs are summarized in the table below.

	Wireless Messaging API	Nokia SMS API
Package name	<code>javax.wireless.messaging</code>	<code>com.nokia.mid.messaging</code>
Method	<code>int numberOfSegment(Message msg)</code> (see <code>MessageConnection</code>)	<code>int calculateNumberOfSmsNeeded(Message msg)</code> (see <code>GsmMessageConnection</code>)
Method	<code>void setMessageListener(MessageListener listener)</code> (see <code>MessageConnection</code>)	<code>void registerMessageListener(MessageListener listener)</code> (see <code>GsmMessageConnection</code>)
SMS URL	<code>"sms://" + <address_part></code>	<code>"com.nokia.sms://" + <address_part></code>
JAD attribute		Nokia-SMS-Handler-<N> can be used to start a MIDlet when an SMS is received on an indicated port.

2 ChatMIDlet

2.1 User Interface

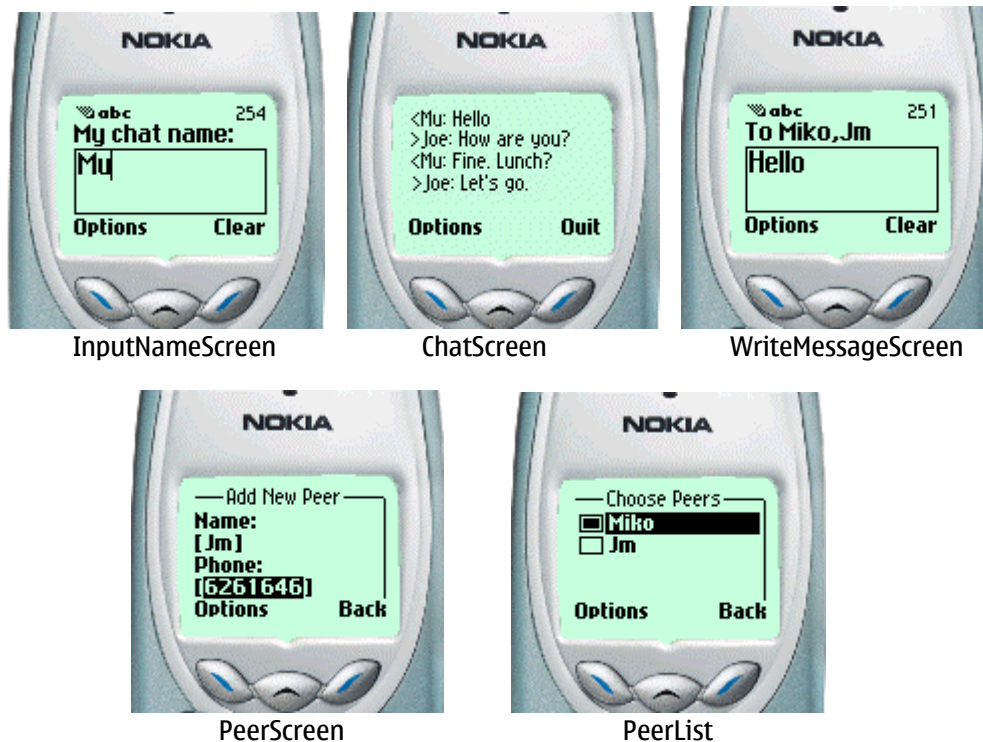


Figure 1: Shots of the Chat MIDlet screens

The user interface and screens of the ChatMIDlet are shown above. There are five main screens:

- **InputNameScreen:** This screen is used to set the user's chat name for a chat session. It switches the UI to a chat screen when input of the chat name has been completed.
- **ChatScreen:** This is the main screen of the MIDlet. It shows the chat content. It has two commands: "To" and "Quit". The "To" command causes the MIDlet to switch to a PeerList screen for selecting chat peers. When one or more chat peers are chosen or a message is received from a chat peer, then a "Write" command is added.
- **WriteMessageScreen:** This screen is used to write message to new peers or reply to a chat peer.
- **PeerScreen:** This screen is used to update a chat peer's name and phone number in a local address book.
- **PeerList:** This screen is used to select chat peers from a list of peer names stored in a local phone book. Multiple chat peers can be selected. Peers can also be deleted from the PeerList and local address book. If no peer is selected, a "New" command can be used to switch to a PeerScreen from where a new peer's name and phone number can be input.

The figure below shows the user interface transitions between InputNameScreen, ChatScreen, PeerScreen, PeerList, and WriteMessageScreen.

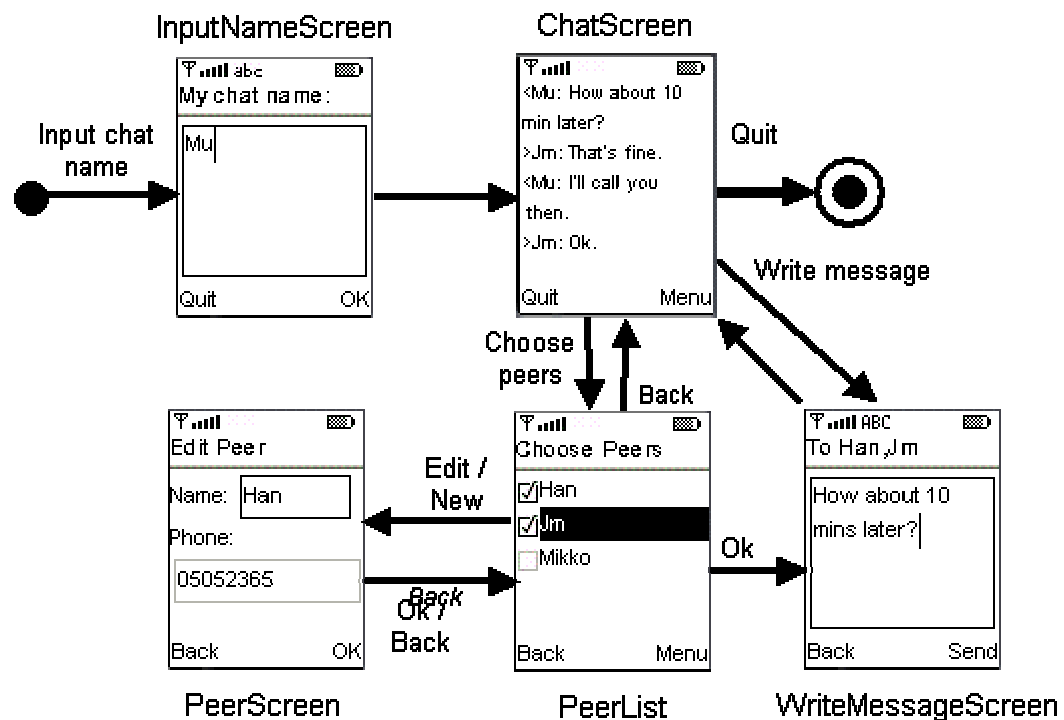


Figure 2: User interface screen transitions

When the ChatMIDlet starts, it first shows an InputNameScreen prompting the user to input his or her chat name. The chat user's name is included as a string prefix in each outgoing message to identify the sending chat user. The ChatMIDlet controls the state of which screen is displayed. The displayed screen state changes by callbacks from various screens to the ChatMIDlet.

When InputNameScreen is done, the main user interface ChatScreen will be shown. From ChatScreen, the user can choose chat peers from a PeerList screen, or write a message on a WriteMessageScreen, or quit the MIDlet.

The user can select a number of chat peers from the PeerList screen. Those peers' names and phone numbers can be further edited on a PeerScreen. The PeerScreen can also be used to add a new chat peer's contact information when there is no chat peer selected from PeerList. After updating the chat peer's name and phone number (if needed), the current screen will be switched back from PeerScreen to PeerList by a callback to the ChatMIDlet.

The WriteMessageScreen is used to input text for the outgoing SMS message. The screen state will be switched back to the ChatScreen when input of the message text is complete.

An ErrorScreen may be displayed, showing an appropriate error message, whenever an error occurs. It is displayed for five seconds before the screen state is switched back to the previously displayed screen.

2.2 Design

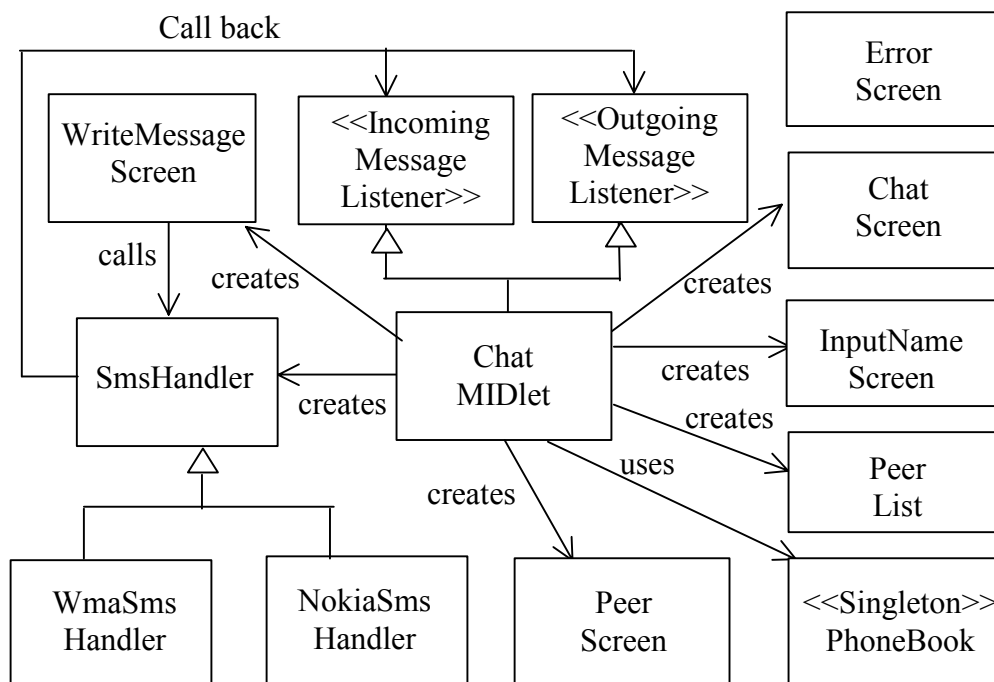


Figure 3: The ChatMIDlet's classes

As shown in Figure 3, the ChatMIDlet is responsible for creating the user interface screens, the PhoneBook, and the SmsHandler. It also initializes the ErrorScreen, which is used to show error messages.

The PhoneBook is used to store update chat peers' address information in the local MIDP record store. It uses the Singleton design pattern to "wrap" use of the MIDP record store.

SmsHandler handles message sending and receiving using the WMA or Nokia SMS API depending on which set of API is available on the MIDP device. The Strategy design pattern is applied here as shown in Figure 4.

SmsHandler is an abstract class that defines common properties and methods for both the WMA and Nokia SMS API classes. The methods, which directly refer to either WMA or Nokia SMS API for message connection setting up, message sending, and receiving, are defined as abstract methods. NokiaSmsHandler and WmaSmsHandler, which inherit SmsHandler, define concrete properties and implement the abstract methods of SmsHandler to send and receive messages using WMA and Nokia SMS API, respectively.

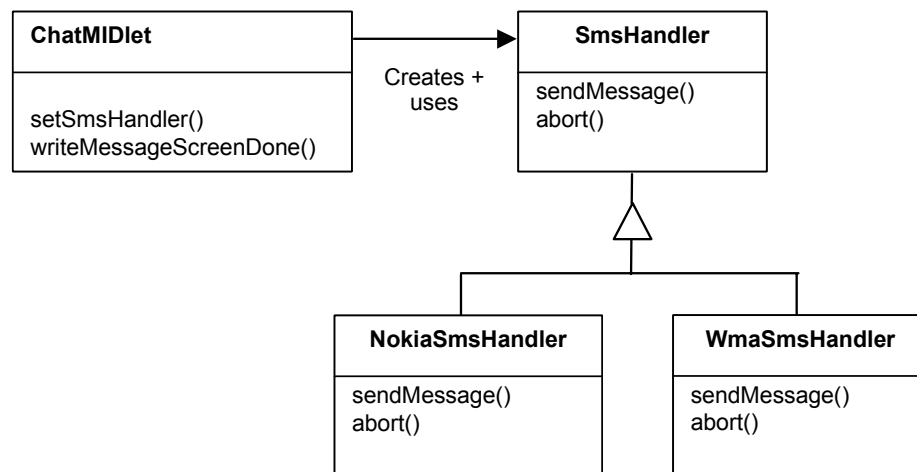


Figure 4: The strategy design pattern is used for message handling

The `setSmsHandler()` method of the `ChatMIDlet` is used to determine which concrete message handler (`WmaSmsHandler` or `NokiaSmsHandler`) is used at runtime based on whether the WMA or Nokia SMS API is available.

The `ChatMIDlet` itself implements `IncomingTextMessageListener` for handling incoming messages (via `ChatScreen`) and `OutgoingTextMessageListener` for handling the results of sending outgoing messages (success or failure).

Since the `MessageConnection.send()` method may block (which means the caller has to wait before the method call returned), `SmsHandler` uses a separate thread to send outgoing messages buffered in a message queue and to receive incoming messages. The `SmsHandler` thread will empty the queue of outgoing messages one by one, using `MessageConnection.send()` to send them. In this way, the user interface should not block because of sending or receiving SMS messages.

For incoming messages, both `WmaSmsHandler` and `NokiaSmsHandler` implement `MessageListener` defined in the relevant messaging APIs. Whenever a message comes, the `WmaSmsHandler` thread will be notified and call `MessageConnection.receive()` to get incoming messages one by one. It uses `ChatMIDlet` (as an `IncomingTextMessageListener`) to handle them.

More design details of each class are described in the sections that follow.

2.3 ChatMIDlet.java

The `ChatMIDlet` is the “owner” of whichever screen is displayed. Transitions of the UI screen state are thus done by callbacks from screens to the `ChatMIDlet`. The primary UI screen is `ChatScreen`. The `ChatMIDlet` also handles callbacks for sent and received SMS messages.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
```

```
public class ChatMIDlet
    extends MIDlet
```

```

implements IncomingTextMessageListener,
               OutgoingTextMessageListener,
               CommandListener
{
    private final static PhoneBook phoneBook = PhoneBook.getPhoneBook();

    private static SmsHandler smsHandler = null;
    private String port = null;
    private ChatScreen chatScreen = null;
    private WriteMessageScreen writeMessageScreen = null;
    private PeerList peerList = null;
    private PeerScreen peerScreen = null;
    private String phoneNumbers[];
    private String prompt;

    public ChatMIDlet()
    {
    }

    public void startApp()
    {
        Displayable current = Display.getDisplay(this).getCurrent();

        if (current == null)
        {
            if (setSmsHandler())
            {
                InputNameScreen inputNameScreen =
                    new InputNameScreen(this);

                Display.getDisplay(this).setCurrent(inputNameScreen);
            }
            else
            {
                Form quitScreen = new Form("Error");

                quitScreen.append("Neither WMA nor " +
                                "Nokia SMS API is available!");
                quitScreen.addCommand(new Command("Quit", Command.EXIT, 2));
                quitScreen.setCommandListener(this);
                Display.getDisplay(this).setCurrent(quitScreen);
            }
        }
        else
        {
            Display.getDisplay(this).setCurrent(current);
        }
    }

    private boolean setSmsHandler()
    {
        port = getAppProperty("Default-Port");

        try
        {
            Class.forName("javax.wireless.messaging.Message");
            smsHandler =
                (SmsHandler) Class.forName("WmaSmsHandler").newInstance();
        }
        catch (Exception e)
        {
            return false;
        }
    }
}

```

```

    }
    catch (Exception e)
    {
    }

    if (smsHandler == null)
    {
        try
        {
            Class.forName("com.nokia.mid.messaging.Message");
            smsHandler =
                (SmsHandler) Class.forName("NokiaSmsHandler").newInstance();
        }
        catch (Exception e)
        {
        }

        String attr = getAppProperty("Nokia-SMS-Handler-1");

        if (attr != null)
        {
            port = attr.substring(attr.indexOf(':') + 4,
                attr.indexOf(', '));
        }
    }

    return (smsHandler != null);
}

```

```

public void commandAction(Command c, Displayable d)
{
    exitRequested();
}

```

// InputNameScreen Callbacks

```

void inputNameScreenDone(String chatName)
{
    if (smsHandler != null)
    {
        ErrorScreen.init(null, Display.getDisplay(this));

        smsHandler.init(this, this, port);

        prompt = "<" + chatName + ": ";

        chatScreen = new ChatScreen(this, prompt);
        Display.getDisplay(this).setCurrent(chatScreen);
    }
    else
    {
        exitRequested();
    }
}

```

```

void inputNameScreenQuit()
{

```

```

        exitRequested();
    }

    // ChatScreen Callbacks

    void chatScreenChoosePeer()
    {
        choosePeer();
    }

    void chatScreenWriteMessage()
    {
        writeMessage(null);
    }

    void chatScreenQuit()
    {
        exitRequested();
    }

    private void choosePeer()
    {
        String[] peerNames = phoneBook.getNames();

        if (peerNames != null)
        {
            if (peerList == null)
            {
                peerList = new PeerList(this, peerNames);
            }
            else
            {
                peerList.init(peerNames);
            }
            Display.getDisplay(this).setCurrent(peerList);
        }
        else
        {
            updatePhoneBook(null);
        }
    }

    private void updatePhoneBook(String name)
    {
        if (peerScreen == null)
        {
            peerScreen = new PeerScreen(this);
        }

        String phoneNumber = (name == null) ?
                               null : phoneBook.getPhoneNumber(name);

        peerScreen.init(name, phoneNumber);
        Display.getDisplay(this).setCurrent(peerScreen);
    }

```

```

private void writeMessage(String[] peerNames)
{
    if (writeMessageScreen == null)
    {
        writeMessageScreen = new WriteMessageScreen(this);
    }

    if (peerNames == null)
    {
        peerNames = phoneBook.getNames(phoneNumbers);
    }
    writeMessageScreen.setTitle(peerNames);
    Display.getDisplay(this).setCurrent(writeMessageScreen);
}

// PeerList screen Callbacks

void peerListBack()
{
    Display.getDisplay(this).setCurrent(chatScreen);
}

void peerListEdit(String name)
{
    updatePhoneBook(name);
}

void peerListDone(String[] peerNames)
{
    if (peerNames != null)
    {
        phoneNumbers = new String[peerNames.length];

        for (int i = phoneNumbers.length - 1; i >= 0; i--)
        {
            phoneNumbers[i] = phoneBook.getPhoneNumber(peerNames[i]);
        }

        chatScreen.enableWriteCommand();
        writeMessage(peerNames);
    }
    else
    {
        Display.getDisplay(this).setCurrent(chatScreen);
    }
}

void peerListNew()
{
    updatePhoneBook(null);
}

void peerListRemove(String[] names)
{
    phoneBook.removeEntries(names);
}

```

```

}

// PeerScreen callbacks

void peerScreenDone(String name, String phoneNumber)
{
    phoneBook.update(name, phoneNumber);
    choosePeer();
}

void peerScreenBack()
{
    if (peerList != null && peerList.size() > 0)
    {
        Display.getDisplay(this).setCurrent(peerList);
    }
    else
    {
        Display.getDisplay(this).setCurrent(chatScreen);
    }
}

// WriteMessageScreen callbacks

void writeMessageScreenDone(String content)
{
    Display.getDisplay(this).setCurrent(chatScreen);

    if (content != null)
    {
        String message = prompt + content;

        chatScreen.update(message);
        smsHandler.sendMessage(phoneNumbers, message);
    }
}

void writeMessageScreenBack()
{
    Display.getDisplay(this).setCurrent(chatScreen);
}

// IncomingTextMessageListener interface methods

public void handleIncomingMessage(String phoneNumber, String content)
{
    String fromName = content.substring(1, content.indexOf(':'));

    fromName = phoneBook.validate(fromName, phoneNumber);

    chatScreen.update(">" + content.substring(1));
    chatScreen.enableWriteCommand();

    if (Display.getDisplay(this).getCurrent() != writeMessageScreen)
    {
        phoneNumbers = new String[1];
    }
}

```

```

        phoneNumbers[0] = phoneNumber;
    }
}

public void handleIncomingError(String errorStr)
{
    ErrorScreen.showError(errorStr);
}

// OutgoingTextMessageListener interface methods

public void handleOutgoingOk(OutgoingTextMessage message)
{
}

public void handleOutgoingCancel(OutgoingTextMessage message,
    String type)
{
    writeMessageScreen.setTitle(type);
}

public void handleOutgoingError(OutgoingTextMessage message,
    String errorStr)
{
    ErrorScreen.showError(errorStr);
}

// MIDlet methods

public void pauseApp()
{
}

public void destroyApp(boolean unconditional)
{
}

// A convenience method for exiting.

void exitRequested()
{
    if (smsHandler != null)
    {
        smsHandler.abort();
    }

    destroyApp(true);
    notifyDestroyed();
}
}

```

2.4 ChatScreen.java

This class provides the main UI for the ChatMIDlet. It shows all incoming and outgoing messages. The messages are automatically scrolled during a chat session. A long message will be separated into multiple display lines according to the screen width before it is shown.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;

class ChatScreen
    extends Canvas
    implements CommandListener
{
    private final ChatMIDlet parent;
    private final Vector chatText = new Vector();
    private final Font font;
    private final int lineHeight, lineWidth, maxLines;
    private final Command toCommand, quitCommand;
    private final String prompt;

    private Command writeCommand = null;
    private int topLine = 0;

    ChatScreen(ChatMIDlet parent, String prompt)
    {
        this.parent = parent;
        this.prompt = prompt;

        font = Font.getFont(Font.FACE_PROPORTIONAL,
                           Font.STYLE_PLAIN,
                           Font.SIZE_SMALL);

        lineWidth = getWidth() - 2;
        lineHeight = font.getHeight() + 1;
        maxLines = (getHeight() - 2) / lineHeight;

        toCommand = new Command("To", Command.SCREEN, 1);
        quitCommand = new Command("Quit", Command.EXIT, 2);
        addCommand(toCommand);
        addCommand(quitCommand);
        setCommandListener(this);
    }

    void update(String text)
    {
        if (font.stringWidth(text) > lineWidth)
        {
            // seperate message text into multiple lines and
            // add each to chatText
            int offset = 0;
            int len = 1;

            do
            {
                // calculate the max length of substring fitting one screen line
                while (((offset + len) <= text.length()) &&
```

```

        (font.substringWidth(text, offset, len) <= lineWidth))
    {
        len++;
    }
    chatText.addElement(text.substring(offset, offset + len - 1));
    offset += len - 1;
    len = 1;
} while (offset + len <= text.length());
}
else
{
    chatText.addElement(text);
}

if (chatText.size() > maxLines)
{
    topLine = chatText.size() - maxLines;
}
repaint();
}

```

```

public void paint(Graphics g)
{
    g.setColor(0xFFFFFFFF);
    g.fillRect(0, 0, getWidth(), getHeight());
    g.setColor(0);
    g.setFont(font);

    if (chatText.size() == 0)
    {
        g.drawString(prompt, 1, 1, (Graphics.TOP | Graphics.LEFT));
    }
    else
    {
        for (int i = 0; i < maxLines && i < chatText.size(); i++)
        {
            g.drawString((String) chatText.elementAt(topLine + i),
                1,
                (lineHeight * i + 1),
                (Graphics.TOP | Graphics.LEFT));
        }
    }
}

```

```

protected void keyReleased(int keyCode)
{
    switch (getGameAction(keyCode))
    {
        case Canvas.UP:
            if (topLine != 0)
            {
                topLine--;
                repaint();
            }
            break;

        case Canvas.DOWN:
            if ((topLine + maxLines) < chatText.size())
            {

```

```

        topLine++;
        repaint();
    }
    break;

default:
    break;
}
}

public void commandAction(Command c, Displayable d)
{
    if (c == toCommand)
    {
        parent.chatScreenChoosePeer();
    }
    else if (c == writeCommand)
    {
        parent.chatScreenWriteMessage();
    }
    else if (c == quitCommand)
    {
        parent.chatScreenQuit();
    }
}

void enableWriteCommand()
{
    if (writeCommand == null)
    {
        writeCommand = new Command("Write", Command.SCREEN, 1);
        addCommand(writeCommand);
    }
}
}

```

2.5 SmsHandler.java

This class is used to set up message connections, send messages, and receive messages. It's an abstract class that defines only properties and methods not related to either the WMA or Nokia SMS API. The properties and methods that relate to the messaging API are defined in the subclasses WmaSmsHandler and NokiaSmsHandler, respectively. SmsHandler creates a thread and uses a message queue so that other threads of the MIDlet will block when sending a message using the blocking method MessageConnection.send() in either the WMA or Nokia SMS API.

```

import java.io.*;
import java.util.*;

abstract class SmsHandler
    implements Runnable
{
    protected final Vector outgoingTextMessages = new Vector();

    protected IncomingTextMessageListener incomingListener;
    protected OutgoingTextMessageListener outgoingListener;
    protected String port = null;
}

```

```

protected volatile boolean aborting = false;
protected volatile int incomingMessagesPending = 0;

protected SmsHandler()
{
}

void init(IncomingTextMessageListener incomingListener,
    OutgoingTextMessageListener outgoingListener,
    String port)
{
    this.incomingListener = incomingListener;
    this.outgoingListener = outgoingListener;
    this.port = port;

    // If we are able to open the server mode connection,
    // then start the thread that does listening + sending.
    if (openServerModeConnection())
    {
        Thread t = new Thread(this);
        t.start();
    }
}

public void run()
{
    while (!aborting)
    {
        // 1. Wait if there is nothing to send or receive.
        synchronized (this)
        {
            if ((outgoingTextMessages.size() == 0) &&
                (incomingMessagesPending == 0))
            {
                try
                {
                    wait(); // releases lock
                }
                catch (InterruptedException e)
                {
                }
            }
        }

        if (aborting)
        {
            break;
        }

        // 2. If there are any outgoing messages on the queue, then send one.
        while (outgoingTextMessages.size() > 0)
        {
            OutgoingTextMessage outgoing =
                (OutgoingTextMessage) outgoingTextMessages.firstElement();

            doSend(outgoing);
            outgoingTextMessages.removeElementAt(0);
        }
    }
}

```

```

        // 3. If there is an incoming message pending, then read it.
        while (incomingMessagesPending > 0)
        {
            doReceive();

            synchronized (this)
            {
                incomingMessagesPending--;
            }
        }
    }
}

protected abstract boolean openServerModeConnection();

protected abstract void doSend(OutgoingTextMessage outgoing);

protected abstract void doReceive();

void sendMessage(String[] addrs, String content)
{
    for (int i = 0; i < addrs.length; i++)
    {
        if (addrs[i] != null)
        {
            OutgoingTextMessage outgoing =
                new OutgoingTextMessage(addrs[i], content);

            outgoingTextMessages.addElement(outgoing);
        }
    }

    synchronized (this)
    {
        notify(); // wake up the sending thread
    }
}

void abort()
{
    // Set the state to 'aborting'.

    aborting = true;
    synchronized (this)
    {
        notify(); // wake up the SmsHandler thread, to kill it
    }
}
}

```

2.6 NokiaSmsHandler.java

This class is a subclass of SmsHandler and implements its abstract methods using the Nokia SMS API.

```
import java.io.*;
import javax.microedition.io.*;
import com.nokia.mid.messaging.*;

class NokiaSmsHandler
    extends SmsHandler
    implements MessageListener
{
    private final static String PROTOCOL = "com.nokia.sms://";

    private MessageConnection conn;

    NokiaSmsHandler()
    {
    }

    protected void doSend(OutgoingTextMessage outgoing)
    {
        try
        {
            TextMessage message =
                (TextMessage) conn.newMessage(MessageConnection.TEXT_MESSAGE);

            String outgoingAddress = outgoing.getAddress();

            if (outgoingAddress.indexOf(':') == -1)
            {
                outgoingAddress += ":" + port;
            }
            message.setAddress(outgoingAddress);
            message.setPayloadText(outgoing.getMessage());
            conn.send(message);

            outgoingListener.handleOutgoingOk(outgoing);
        }
        catch (SecurityException e)
        {
            outgoingListener.handleOutgoingCancel(outgoing, "Security");
        }
        catch (InterruptedException e)
        {
            outgoingListener.handleOutgoingCancel(outgoing, "Interrupted");
        }
        catch (Exception e)
        {
            outgoingListener.handleOutgoingError(outgoing,
                "Outgoing Error:" + e);
        }
    }

    protected void doReceive()
    {
    }
}
```

```

try
{
    Message message = conn.receive();

    if (message instanceof TextMessage)
    {
        TextMessage textMessage = (TextMessage) message;
        String payload = textMessage.getPayloadText();
        String address = textMessage.getAddress();

        String phoneNumber = address; // default
        // strip off PROTOCOL prefix if any
        int beginIndex = 0;
        if (address.startsWith(PROTOCOL))
        {
            beginIndex = PROTOCOL.length();
        }
        // strip off portnumber ending if any
        int endIndex = address.lastIndexOf(':');
        if (endIndex > beginIndex)
        {
            phoneNumber = address.substring(beginIndex, endIndex);
        }
        else
        {
            phoneNumber = address.substring(beginIndex);
        }

        incomingListener.handleIncomingMessage(phoneNumber, payload);
    }
    else
    {
        incomingListener.handleIncomingError(
            "Incoming Error: Non-text message received");
    }
}
catch (Exception e)
{
    incomingListener.handleIncomingError(
        "Incoming Error: '" + e.getMessage() + "'");
}
}

public synchronized void notifyIncomingMessage(MessageConnection conn)
{
    incomingMessagesPending++;

    synchronized (this)
    {
        notify(); // wake up the SmsHandler thread
    }
}

protected boolean openServerModeConnection()
{
    try
    {
        conn = (MessageConnection) Connector.open(PROTOCOL + ":" + port);
        conn.registerMessageListener(this);
    }
}

```

```

    }
    catch (Exception e)
    {
        conn = null;
    }

    return (conn != null);
}

void abort()
{
    // Cleanup the server connection. Note that 'conn' is created in the
    // constructor and destroyed here, so sending/receiving are no longer
    // possible after an abort, unless a new NokHandler object is created.

    if (conn != null)
    {
        try
        {
            conn.registerMessageListener(null);
            conn.close();
        }
        catch (Exception e)
        {
            // Since we are aborting, no error handling needed.
        }
        conn = null;
    }

    super.abort();
}
}

```

2.7 WmaSmsHandler.java

This class is a subclass of SmsHandler and implements its abstract methods using the WMA.

```

import java.io.*;
import javax.microedition.io.*;
import javax.wireless.messaging.*;

public class WmaSmsHandler
    extends SmsHandler
    implements MessageListener
{
    private final static String PROTOCOL = "sms://";

    private MessageConnection conn;

    WmaSmsHandler()
    {
    }

    protected void doSend(OutgoingTextMessage outgoing)
    {
    }
}

```

```

try
{
    TextMessage message =
        (TextMessage) conn.newMessage(MessageConnection.TEXT_MESSAGE);

    String outgoingAddress = outgoing.getAddress();

    if (outgoingAddress.indexOf(':') == -1)
    {
        outgoingAddress += ":" + port;
    }

    message.setAddress(PROTOCOL + outgoingAddress);
    message.setPayloadText(outgoing.getMessage());
    conn.send(message);

    outgoingListener.handleOutgoingOk(outgoing);
}
catch (SecurityException e)
{
    outgoingListener.handleOutgoingCancel(outgoing, "Security");
}
catch (InterruptedException e)
{
    outgoingListener.handleOutgoingCancel(outgoing, "Interrupted");
}
catch (Exception e)
{
    outgoingListener.handleOutgoingError(outgoing,
        "Outgoing Error:" + e);
}
}

```

```

protected void doReceive()
{
    try
    {
        Message message = conn.receive();

        if (message instanceof TextMessage)
        {
            TextMessage textMessage = (TextMessage) message;
            String payload = textMessage.getPayloadText();
            String address = textMessage.getAddress();

            String phoneNumber = address; // default
            // strip off PROTOCOL prefix if any
            int beginIndex = 0;
            if (address.startsWith(PROTOCOL))
            {
                beginIndex = PROTOCOL.length();
            }
            // strip off portnumber ending if any
            int endIndex = address.lastIndexOf(':');
            if (endIndex > beginIndex)
            {
                phoneNumber = address.substring(beginIndex, endIndex);
            }
            else
            {

```

```

        phoneNumber = address.substring(beginIndex);
    }

    incomingListener.handleIncomingMessage(phoneNumber, payload);
}
else
{
    incomingListener.handleIncomingError(
        "Incoming Error: Non-text message received");
}
}
catch (Exception e)
{
    incomingListener.handleIncomingError(
        "Incoming Error: '" + e.getMessage() + "'");
}
}

// this method is from MessageListener
public synchronized void notifyIncomingMessage(MessageConnection conn)
{
    incomingMessagesPending++;

    synchronized (this)
    {
        notify(); // wake up the SmsHandler thread
    }
}

protected boolean openServerModeConnection()
{
    try
    {
        conn = (MessageConnection) Connector.open(PROTOCOL + ":" + port);
        conn.setMessageListener(this);
    }
    catch (Exception e)
    {
        conn = null;
    }
    return (conn != null);
}

void abort()
{
    // Cleanup the server connection. Note that 'conn' is created in the
    // constructor and destroyed here, so sending/receiving are no longer
    // possible after an abort, unless a new WmaHandler object is created.

    if (conn != null)
    {
        try
        {
            conn.setMessageListener(null);
            conn.close();
        }
        catch (Exception e)
        {
        }
    }
}

```

```

        // Since we are aborting, no error handling needed.
    }
    conn = null;
}
super.abort();
}
}

```

2.8 WriteMessageScreen.java

This class is used to input text used for outgoing message. The screen title shows the chat peers' name separated by commas.

```

import javax.microedition.lcdui.*;
import javax.microedition.io.*;

class WriteMessageScreen
    extends TextBox
    implements CommandListener
{
    private final ChatMIDlet parent;
    private final Command sendCommand, backCommand;

    WriteMessageScreen(ChatMIDlet parent)
    {
        super("", "", 256, TextField.ANY);
        this.parent = parent;

        sendCommand = new Command("Send", Command.SCREEN, 1);
        backCommand = new Command("Back", Command.BACK, 2);
        addCommand(sendCommand);
        addCommand(backCommand);

        setCommandListener(this);
    }

    void setTitle(String[] peerNames)
    {
        StringBuffer peerNameBuffer = new StringBuffer("To ");

        for (int i = 0; i < peerNames.length; i++)
        {
            peerNameBuffer.append(peerNames[i]);
            peerNameBuffer.append((i != peerNames.length - 1) ? "," : "");
        }
        setTitle(peerNameBuffer.toString());
    }

    public void commandAction(Command c, Displayable d)
    {
        if (c == sendCommand)
        {
            parent.writeMessageScreenDone(getString());
        }
    }
}

```

```

        else
        {
            // backCommand
            parent.writeMessageScreenBack();
        }
    }
}

```

2.9 PeerScreen.java

This class provides the UI for the user to edit or create a chat peer's contact information including name and phone number. It is initialized with a peer's information stored in PhoneBook if the peer's name is chosen in the previous PeerList screen. Otherwise a blank PeerScreen form is used for the user to input a new peer's name and phone number.

```

import javax.microedition.lcdui.*;

class PeerScreen
    extends Form
    implements CommandListener
{
    private final ChatMIDlet parent;
    private final TextField nameField;
    private final TextField phoneNumberField;
    private final Command okCommand, backCommand;

    PeerScreen(ChatMIDlet parent)
    {
        super("");
        this.parent = parent;

        nameField = new TextField(" Name: ", null, 5, TextField.ANY);
        phoneNumberField = new TextField(" Phone:",
                                         null,
                                         20,
                                         TextField.PHONENUMBER);

        append(nameField);
        append(phoneNumberField);

        okCommand = new Command("OK", Command.OK, 1);
        backCommand = new Command("Back", Command.BACK, 2);
        addCommand(okCommand);
        addCommand(backCommand);
        setCommandListener(this);
    }

    void init(String name, String phoneNumber)
    {
        nameField.setString(name);
        phoneNumberField.setString(phoneNumber);
        setTitle((name != null) ? "Edit Peer" : "Add New Peer");
    }

    public void commandAction(Command c, Displayable d)
    {

```

```

if (c == okCommand)
{
    String name = nameField.getString();
    String phoneNumber = phoneNumberField.getString();

    if (name.length() > 0 && phoneNumber.length() > 0)
    {
        parent.peerScreenDone(name, phoneNumber);
    }
    else
    {
        ErrorScreen.showError("Name or Address is invalid!");
    }
}
else
{
    // backCommand
    parent.peerScreenBack();
}
}
}

```

2.10 PeerList.java

This class provides a simple list of known chat peers for the user to choose from by name. Multiple choices of the list may be selected. If no name is selected, the user can input a new peer's contact information using the "New" command. If one or more names are selected, the user can edit, remove selected peers' contact information in the PhoneBook, or confirm the selection by using the Edit, Remove, or Ok command, respectively.

```

import javax.microedition.lcdui.*;

class PeerList
    extends Form
    implements CommandListener, ItemStateListener
{
    private final ChatMIDlet parent;
    private final ChoiceGroup peers;
    private final Command okCommand, backCommand, newCommand;
    private final Command editCommand, removeCommand;

    private boolean isElementSelected = false;

    PeerList(ChatMIDlet parent, String[] peerNames)
    {
        super("Choose Peers");
        this.parent = parent;

        peers = new ChoiceGroup("", List.MULTIPLE, peerNames, null);
        append(peers);

        okCommand = new Command("OK", Command.OK, 1);
        backCommand = new Command("Back", Command.BACK, 2);
        newCommand = new Command("New", Command.SCREEN, 2);
        editCommand = new Command("Edit", Command.SCREEN, 2);
        removeCommand = new Command("Remove", Command.SCREEN, 2);
        addCommand(backCommand);
    }
}

```

```

        addCommand(newCommand);

        setCommandListener(this);
        setItemStateListener(this);
    }

    void init(String[] names)
    {
        while (peers.size() > 0)
        {
            peers.delete(0);
        }
        for (int i = 0; i < names.length; i++)
        {
            peers.append(names[i], null);
        }
        elementUnselected();
    }

    public void itemStateChanged(Item item)
    {
        for (int i = 0; i < peers.size(); i++)
        {
            if (peers.isSelected(i))
            {
                if (!isElementSelected)
                {
                    elementSelected();
                }
                return;
            }
        }
        elementUnselected();
    }

    private void elementUnselected()
    {
        removeCommand(newCommand);
        removeCommand(okCommand);
        removeCommand(editCommand);
        removeCommand(removeCommand);
        addCommand(newCommand);
        isElementSelected = false;
    }

    private void elementSelected()
    {
        removeCommand(okCommand);
        removeCommand(editCommand);
        removeCommand(removeCommand);
        removeCommand(newCommand);
        addCommand(okCommand);
        addCommand(editCommand);
        addCommand(removeCommand);
        isElementSelected = true;
    }

```

```

public void commandAction(Command c, Displayable d)
{
    if (c == okCommand || c == removeCommand)
    {
        // get selected names and put them to an array
        int peerNum = 0;
        int i;

        for (i = 0; i < peers.size(); i++)
        {
            peerNum += peers.isSelected(i) ? 1 : 0;
        }

        String[] peerNames = new String[peerNum];
        int k = 0;

        for (i = 0; i < peers.size() && k < peerNum; i++)
        {
            if (peers.isSelected(i))
            {
                peerNames[k] = peers.getString(i);
                k++;
            }
        }

        if (c == okCommand)
        {
            parent.peerListDone(peerNames);
        }
        else
        {
            // removeCommand

            for (i = peers.size() - 1, k = peerNum; i >= 0 && k > 0; i--)
            {
                if (peers.isSelected(i))
                {
                    peers.delete(i);
                    k--;
                }
            }
            elementUnselected();
            parent.peerListRemove(peerNames);
        }
    }
    else if (c == newCommand)
    {
        parent.peerListNew();
    }
    else if (c == editCommand)
    {
        int i = 0;

        for (; i < peers.size() - 1 && !peers.isSelected(i); i++)
        {
            ;
        }
        parent.peerListEdit(peers.getString(i));
    }
    else

```

```

        {
            // backCommand
            parent.peerListBack();
        }
    }

    public int size()
    {
        return peers.size();
    }
}

```

2.11 PhoneBook.java

This class encapsulates a record store that stores the name and phone numbers of all peers that the user has been chatting with. It uses the Singleton design pattern so that it can be opened and closed at most once from a single place in the MIDlet application code. Using the Singleton design pattern also avoids creating many PhoneBook objects.

When the class is instantiated, it gets all peers' names and phone numbers from the record store and keeps them in two vectors. Whenever a peer's name or phone number is changed, all peers' name and phone number from the vectors will be saved to a new record store. The old record store will be deleted.

```

import javax.microedition.rms.*;
import java.util.*;

class PhoneBook
{
    private final static String RSNAME = "phonebook";
    private final static char DELIMITER = '|';

    private static PhoneBook phoneBook;
    private static Vector names = new Vector();
    private static Vector phones = new Vector();

    static PhoneBook getPhoneBook()
    {
        if (phoneBook == null)
        {
            phoneBook = new PhoneBook();
        }
        return phoneBook;
    }

    private PhoneBook()
    {
        try
        {
            RecordStore rs = RecordStore.openRecordStore(RSNAME, true);
            int peerNum = rs.getNumRecords();

            for (int i = 0; i < peerNum; i++)

```

```

        {
            byte[] byteRec = rs.getRecord(i + 1);
            String peer = new String(byteRec);

            names.addElement(peer.substring(0, peer.indexOf(DELIMITER)));
            phones.addElement(peer.substring(peer.indexOf(DELIMITER) + 1));
        }
        rs.closeRecordStore();
    }
    catch (RecordStoreException e)
    {
        ErrorScreen.showError("Open RMS Error");
    }
}

```

```

synchronized String getName(String phoneNumber)
{
    int index = phones.indexOf(phoneNumber);

    return (index != -1) ? (String) names.elementAt(index) : null;
}

```

```

synchronized String getPhoneNumber(String name)
{
    int index = names.indexOf(name);

    return (index != -1) ? (String) phones.elementAt(index) : null;
}

```

```

synchronized String[] getNames()
{
    String[] names = null;

    if (this.names.size() > 0)
    {
        names = new String[this.names.size()];
        this.names.copyInto(names);
    }
    return names;
}

```

```

synchronized String[] getNames(String[] phones)
{
    String[] names = new String[phones.length];

    for (int i = 0; i < names.length; i++)
    {
        names[i] = getName(phones[i]);
    }
    return names;
}

```

```

synchronized String validate(String peerName, String peerAddress)
{
    int i;
    String name = null;
}

```

```

String peerNum = (peerAddress.indexOf(':') == -1) ?
    peerAddress :
    peerAddress.substring(0, peerAddress.indexOf(':'));

for (i = 0; i < phones.size(); i++)
{
    String phoneNumber = (String) phones.elementAt(i);
    String phoneNum = (phoneNumber.indexOf(':') == -1) ?
        phoneNumber :
        phoneNumber.substring(0, phoneNumber.indexOf(':'));

    // Only simple string-based matching of the phone numbers is
    // performed below. More advanced matching based on the 'number
    // structure' or 'numbering plan' of phone numbers is not
    // performed. Specifically, we do not check if a phone number
    // string using global number structure (e.g. "+3585012345"),
    // matches a previously saved phone number that used a local
    // structure within some geographic area (e.g. "05012345"
    // dialed within Finland).

    if (peerNum.equals(phoneNum))
    {
        name = (String) names.elementAt(i);
        update(name, peerAddress);
        break;
    }
}

if (i == phones.size())
{
    // new peer
    for (i = 0; i < names.size(); i++)
    {
        name = (String) names.elementAt(i);
        if (peerName.toLowerCase().equals(name.toLowerCase()))
        {
            // name exists, use name' instead
            name = peerName + '\';
            update(name, peerAddress);
            break;
        }
    }

    if (i == names.size())
    {
        // brand new, save to rms
        update(peerName, peerAddress);
        name = peerName;
    }
}

return name;
}

synchronized void update(String peerName, String peerAddress)
{
    String peername = peerName.toLowerCase();

    int i;

```

```

for (i = 0; i < names.size(); i++)
{
    String name = ((String) names.elementAt(i)).toLowerCase();
    String phoneNumber = (String) phones.elementAt(i);

    if (peername.equals(name) || peerAddress.equals(phoneNumber))
    {
        if (peername.equals(name) && peerAddress.equals(phoneNumber))
        {
            return;
        }
        names.setElementAt(peerName, i);
        phones.setElementAt(peerAddress, i);
        break;
    }
}

if (i == names.size())
{
    names.insertElementAt(peerName, 0);
    phones.insertElementAt(peerAddress, 0);
}
updateRecordStore();
}

```

```

synchronized void removeEntries(String[] peerNames)
{
    for (int i = 0; i < peerNames.length; i++)
    {
        int index = names.indexOf(peerNames[i]);

        names.removeElementAt(index);
        phones.removeElementAt(index);
    }
    updateRecordStore();
}

```

```

synchronized void updateRecordStore()
{
    try
    {
        RecordStore.deleteRecordStore(RSNAME);

        RecordStore rs = RecordStore.openRecordStore(RSNAME, true);

        for (int i = 0; i < names.size(); i++)
        {
            String peer = (String) names.elementAt(i) + DELIMITER +
                (String) phones.elementAt(i);

            rs.addRecord(peer.getBytes(), 0, peer.length());
        }

        rs.closeRecordStore();
    }
    catch (RecordStoreException e)
    {
        ErrorScreen.showError("Update RMS Error!");
    }
}

```

```

    }
    return;
}
}

```

2.12 InputNameScreen.java

This class provides a simple UI for the user to input a chat name. It only pops up once before the main UI screen, i.e., ChatScreen, is shown.

```

import javax.microedition.lcdui.*;
import javax.microedition.io.*;

class InputNameScreen
    extends TextBox
    implements CommandListener
{
    private final ChatMIDlet parent;
    private final Command okCommand, quitCommand;

    InputNameScreen(ChatMIDlet parent)
    {
        super("My chat name:", "", 256, TextField.ANY);
        this.parent = parent;

        okCommand = new Command("OK", Command.OK, 1);
        quitCommand = new Command("Quit", Command.EXIT, 2);
        addCommand(okCommand);
        addCommand(quitCommand);

        setCommandListener(this);
    }

    public void commandAction(Command c, Displayable d)
    {
        if (c == okCommand)
        {
            parent.inputNameScreenDone(getString());
        }
        else
        {
            // quitCommand
            parent.inputNameScreenQuit();
        }
    }
}

```

2.13 ErrorScreen.java

This class displays an error message for five seconds before switching back to the previous screen.

```
import javax.microedition.lcdui.*;
import java.util.Timer;
import java.util.TimerTask;

class ErrorScreen
    extends Alert
{
    private static ErrorScreen instance = null;
    private static Display display;

    private ErrorScreen(Image image, Display display)
    {
        super("Error", null, image, AlertType.ERROR);
        setTimeout(5000);
        this.display = display;
    }

    static void init(Image image, Display display)
    {
        if (instance == null)
        {
            instance = new ErrorScreen(image, display);
        }
    }

    static void showError(String message)
    {
        instance.setString(message);
        display.setCurrent(instance);
    }
}
```

2.14 IncomingTextMessageListener.java

This interface defines methods for handling incoming messages.

```
interface IncomingTextMessageListener
{
    public void handleIncomingMessage(String fromAddress, String content);

    public void handleIncomingError(String errorStr);
}
```

2.15 OutgoingTextMessageListener.java

This interface defines methods for handling outgoing messages.

```
interface OutgoingTextMessageListener
{
    public void handleOutgoingOk(OutgoingTextMessage message);

    public void handleOutgoingCancel(OutgoingTextMessage message, String type);

    public void handleOutgoingError(OutgoingTextMessage message,
                                    String errorStr);
}
```

2.16 OutgoingTextMessage.java

This class is used to store the phone number of the chat peer and message content for each outgoing message. If one message is sent to multiple peers, then one instance will be created (per peer) with the peer's phone number and the same message content.

```
class OutgoingTextMessage
{
    private final String phoneNumber;
    private final String message;

    OutgoingTextMessage(String address, String message)
    {
        this.address = address;
        this.message = message;
    }

    String getAddress()
    {
        return address;
    }

    String getMessage()
    {
        return message;
    }
}
```

2.17 Chat.jad

The following is the application descriptor for the ChatMIDlet.

The line that starts with “Nokia-SMS-Handler-1” is specific to the Nokia SMS API. It is used to start the MIDlet when a message is received on the indicated port.

```
MIDlet-Name: Chat
MIDlet-Version: 0.2.2
MIDlet-Vendor: Forum Nokia
MIDlet-Jar-URL: Chat.jar
MIDlet-Jar-Size: 21896
MIDlet-Description: Chat MIDlet
MIDlet-Icon: /logo.png
Default-Port: 6535
Nokia-SMS-Handler-1: com.nokia.sms://:6535,, ChatMIDlet
MIDlet-1: Chat, /logo.png, ChatMIDlet
```

3 References

MIDPPROG	<i>Brief Introduction to MIDP Programming</i> Forum Nokia, 2002 http://www.forum.nokia.com
GSMSMS	GSM 03.40 v7.4.0, Digital cellular telecommunications system (Phase 2+), Technical realization of the Short Message Service (SMS) ETSI 2000
SMSAPI	Nokia SMS API Forum Nokia, 2001 http://www.forum.nokia.com
SMSALPHABETS	TS 100 900 v7.2.0 (GSM 03.38) Digital cellular telecommunications system (Phase 2+), Alphabets and language-specific information ETSI 1999
WMA	JSR 120: Wireless Messaging API (WMA) http://www.jcp.org/en/jsr/detail?id=120

Build Test Sell

Developing and marketing mobile applications with Nokia

1

[Go to Forum.Nokia.com](http://Forum.Nokia.com)

Forum.Nokia.com provides the tools and resources you need for content and application development as well as the channels for sales to operators, enterprises, and consumers.

Forum.Nokia.com

2

[Download tools and emulators](#)

Forum.Nokia.com/tools has links to tools from Nokia and other industry leaders including Borland, Adobe, AppForge, Macromedia, Metrowerks, and Sun.

Forum.Nokia.com/tools

3

[Get documents and specifications](#)

The documents area contains useful white papers, FAQs, tutorials, and APIs for Symbian OS and Series 60 Platform, J2ME, messaging (including MMS), and other technologies. Forum.Nokia.com/devices lists detailed technical specifications for Nokia devices.

Forum.Nokia.com/documents

Forum.Nokia.com/devices

4

[Test your application and get support](#)

Forum Nokia offers free and fee-based support that provides you with direct access to Nokia engineers and equipment and connects you with other developers around the world. The Nokia OK testing program enables your application to enjoy premium placement in Nokia's sales channels.

Forum.Nokia.com/support

Forum.Nokia.com/ok

[Market through Nokia channels](#)

5

Go to Forum.Nokia.com/business to learn about all of the marketing channels open to you, including Nokia Tradepoint, an online B2B marketplace.

Forum.Nokia.com/business

[Reach buyers around the globe](#)

6

Place your applications in Nokia Tradepoint and they're available to dozens of buying organizations around the world, ranging from leading global operators and enterprises to regional operators and XSPs. Your company and applications will also be considered for the regional Nokia Software Markets as well as other global and regional opportunities, including personal introductions to operators, on-device and in-box placement, and participation in invitation-only events around the world.

Forum.Nokia.com/business