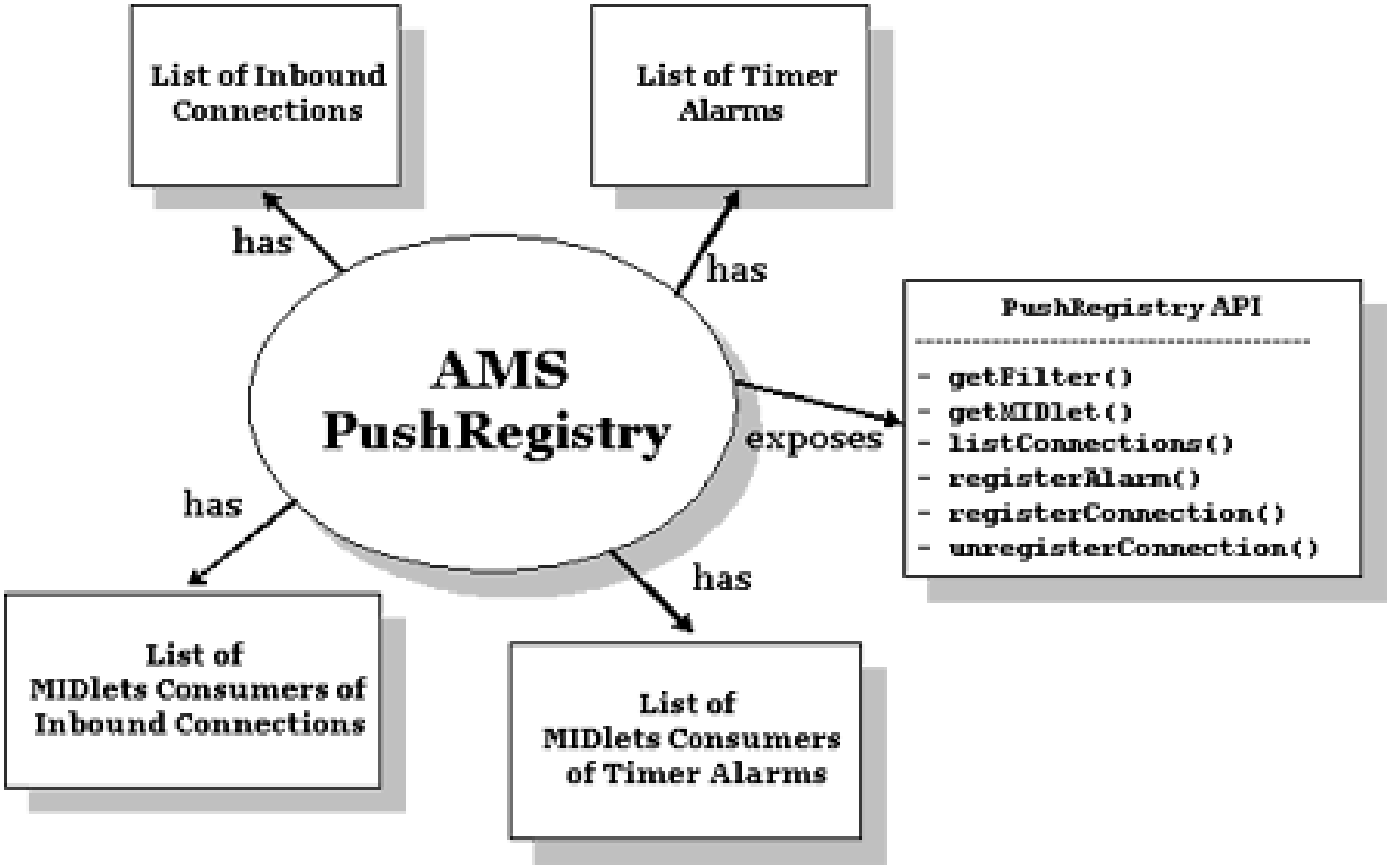
MIDP
Push Registry

Push technology

- "Push" is a very powerful concept and typically refers to the mechanism or ability to receive and act on information asynchronously, as information becomes available, instead of forcing the application to use synchronous polling techniques that increase resource use or latency.
- The *Push Registry* enables MIDlets to set themselves up to be launched automatically, without user initiation. The push registry manages *network*- and *timer*-initiated MIDlet activation; that is, it enables an inbound network connection or a timer-based alarm to wake a MIDlet up.
- For example, you can write a workgroup application that employs network activation to wake up and process newly received email, or new appointments that have been scheduled. Or you can use timer-based activation to schedule your MIDlet to synchronize with a server every so often then go to sleep.

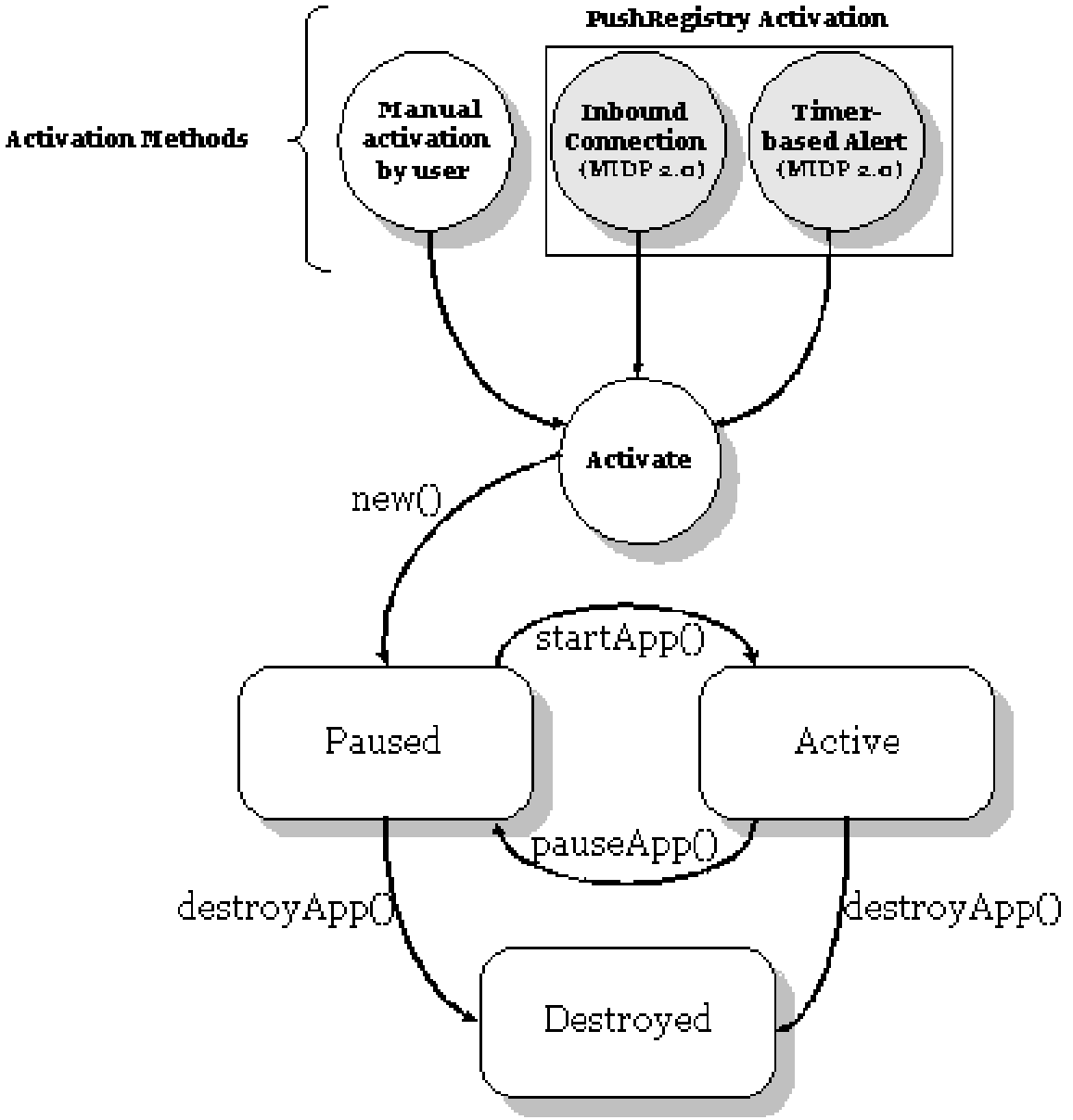
The push registry

- The push registry is part of the application management system (AMS), the software in the device that's responsible for each application's life-cycle (installation, activation, execution, and removal). The push registry is the component of the AMS that exposes the push API and keeps track of push registrations.



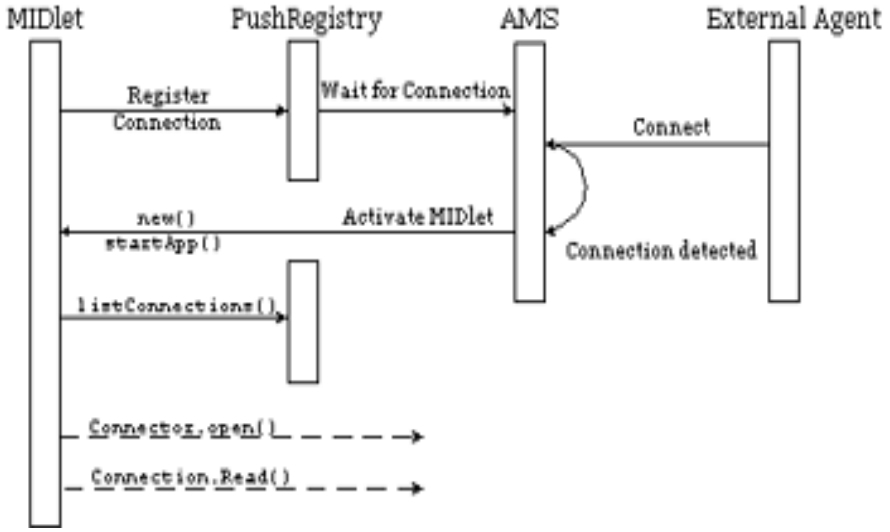
MIDlet activation

- The advent of the push registry doesn't change the MIDlet life-cycle, but it does introduce two new ways a MIDlet may be activated.



Push and AMS

- In MIDP 2.0 the responsibility for push is shared between the MIDlet and the AMS. Once a MIDlet has registered itself with the push registry, the responsibility for push processing is split as follows:
  - 1. When the MIDlet is not active, the AMS monitors registered push events on behalf of the MIDlet. When a push event occurs, the AMS activates the appropriate MIDlet to handle it. The figure illustrates this sequence for a network activation:



  - 2. If the MIDlet is active (running), the MIDlet itself is responsible for all push events. It must set up and monitor inbound connections, and schedule any cyclic tasks it needs - basically standard networking and timer processing.

Static and dynamic registration

- To become push-enabled, MIDlets must register with the push registry, using one of two types of registration:
  - **Static Registration** - Registrations of static (well known) connections occur during the installation of the MIDlet suite. You specify them by listing MIDlet-Push attributes in the MIDlet suite's JAD file or JAR manifest. The installation will fail if you attempt to register an address that's already bound. Uninstalling a MIDlet suite automatically unregisters the connection.
  - **Dynamic Registration** - You register dynamic connections and timer alarms at runtime, using the PushRegistry API.
- In some cases you may want to register a static connection conditionally, or to ensure that push exceptions will not preclude your MIDlet suite from installing. In such cases you can use the PushRegistry API to register your static connection and catch any IOExceptions or SecurityExceptions thrown. Typically, however, you'll register a static connection using the JAD file, and let the system unregister it when the MIDlet suite is uninstalled.
- Note that, while you can use either method to register inbound connections, timer-based activation can be registered only at runtime.

Static registration

- Static registrations are defined by listing one or more *MIDlet-Push* attributes in the JAD file or JAR manifest. The AMS performs static registration when the MIDlet suite is installed. Similarly, when the MIDlet suite is uninstalled, the AMS automatically unregisters all its associated push registrations.
- The format of the *MIDlet-Push* attribute is:
  - *MIDlet-Push-<n>: <ConnectionURL>, <MIDletClassName>, <AllowedSender>*

  where:
  - *MIDlet-Push-<n>* is the property name that identifies push registration, and where <n> is a number starting from 1; for example, MIDlet-Push-1. Note that multiple push entries are allowed.
  - *<ConnectionURL>* is a URL connection string that identifies the inbound endpoint to register, in the same URL format used when invoking *Connector.open()*. For example, *socket://:5000* reserves an inbound server socket connection on port 5000.
  - *<MIDletClassName>* is the fully qualified class name of the MIDlet to be activated when network activity in <ConnectionURL> is detected; for example, *mypackage.MyPushMIDlet*.
  - *<Allowed-Sender>* is a filter used to restrict the servers that can activate *<MIDletClassName>*. You can use wildcards; a * indicates one or more characters and a ? indicates one character. For example, 192.168.1.190, or 192.168.1.*, or 192.168.19?.1, or simply *.

JAD example

```
MIDlet-1: PushMIDlet,,mypackage.MyPushMIDlet

MIDlet-2: WMAMIDlet,,mypackage.WMAMIDlet

MIDlet-Name: MyMIDletSuite

MIDlet-Vendor: Sun Microsystems, Inc.

MIDlet-Version: 1.0

MIDlet-Jar-Size: 4735

MIDlet-Jar-URL: http://myhost.org/basicpush.jar

MicroEdition-Configuration: CLDC-1.0

MicroEdition-Profile: MIDP-1.0

MIDlet-Push-1: socket://:5000, mypackage.MyPushMIDlet, *

MIDlet-Permissions: javax.microedition.io.PushRegistry,
   javax.microedition.io.Connector.ServerSocket
```

The push registry API

- The *push registry API* allows you to register push alarms and connections, and to retrieve information about push connections. A typical push registry maintains lists of connection and alarm registrations in both memory and persistent storage.
- The push registry is part of the Generic Connection Framework (GCF) and is encapsulated within a single class, *javax.microedition.io.PushRegistry*, which exposes all the push-related methods.
  - *static long registerAlarm(String midlet, long time)* Register a time to launch the specified application.
  - *static void registerConnection(String connection, String midlet, String filter)* Register a dynamic connection with the application management software.
  - *static String[] listConnections(boolean available)* Return a list of registered connections for the current MIDlet suite.
  - *static boolean unregisterConnection(String connection)* Remove a dynamic connection registration.
  - *static String getFilter(String connection)* Retrieve the registered filter for a requested connection.
  - *static String getMIDlet(String connection)* Retrieve the registered MIDlet for a requested connection.

Registering a timer alarm (1)

- Your MIDlet may be required to perform some cyclic processing every so often. For example, it may need to synchronize with a server every 5 minutes. Your MIDlet will typically use a *TimerTask* to schedule a thread for cyclic processing, as illustrated in the following:

```
...
// cyclic background task info.
long REFRESH_TIME = 1000*60*5; // every 5 minutes
Timer aTimer = new Timer();
MyTask myTask = new MyTask();
aTimer.schedule(myTask, 0, REFRESH_TIME);
...
class MyTask extends TimerTask {
    // Constructor.
    public MyTask() {
    }
    ...
     // Thread run method.
    public void run() { // ... Your Task Logic    }
}
...
```

Registering a timer alarm (2)

- If your MIDlet needs to continue its cyclic processing even when it's not running, you can use push alarms to schedule your MIDlet for future execution. To schedule a MIDlet launch by the AMS the MIDlet invokes the *PushRegistry.registerAlarm()* method, passing as arguments the fully qualified class name of the MIDlet to launch, and the time for the launch. Passing a time of zero disables the alarm. Note that only one outstanding alarm per MIDlet is supported, and invoking this method overwrites any previously scheduled alarm.

```
private void scheduleMIDlet(long delta)

    throws ClassNotFoundException, ConnectionNotFoundException,

   SecurityException {

  String cn = this.getClass().getName();

  // Get the current time by calling Date.getTime()

  Date alarm = new Date();

  long t = PushRegistry.registerAlarm(cn, alarm.getTime()+delta);

}
```

If the call to *registerAlarm()* overwrites a previous timer, it returns that timer's scheduled time; if not, it returns 0.

Registering a timer alarm (3)

- Note that the PushRegistry API doesn't provide a way to discover whether the MIDlet was activated by an alarm.
- A MIDlet that requires push alarms must schedule them before it exits:

```
public void destroyApp(boolean uc) throws
    MIDletStateChangeException {
    // Release resources
    ...
    // Set up the alarm and force the MIDlet to exit.
    scheduleMIDlet(defaultDeltaTime);
}
```

Registering an inbound connection

```
...
//  MIDlet class name.
String midletClassName = this.getClass().getName();
//  Register a static connection.
String url = "socket://:5000";
//  Use an unrestricted filter.
String filter = "*";
try {
    // Open the connection.
    ServerSocketConnection ssc = (ServerSocketConnection)Connector.open(url);
    PushRegistry.registerConnection(url, midletClassName, filter);
    // Now wait for inbound network activity.
    SocketConnection sc = (SocketConnection)ssc.acceptAndOpen();
    // Read data from inbound connection.
    InputStream is = sc.openInputStream();
    //  ..... read from the input stream.
    // Here process the inbound data.
    //  .....
}
catch(SecurityException e) { ... }
catch(ClassNotFoundException e) { ... }
catch(IOException e) { ... }
…
```

The MIDlet calls *registerConnection()* to register the newly created inbound (server) socket connection.

Register the connection now so that when this MIDlet exits (is destroyed), the AMS can activate our MIDlet when network activity is detected. The AMS will remember the registered URL even when the MIDlet is not active.

Unregistering an inbound connection

- Because the AMS maintains the registration even after the MIDlet exits, it's important that the MIDlet unregister the connection when it's no longer needed. To unregister an inbound connection, use the *unregisterConnection()* method:

```
...
try {
    boolean status;
    // unregisterConnection returns false if it was
    // unsuccessful and true if successful.
    status = PushRegistry.unregisterConnection(url);
}
catch(SecurityException e) {...}
...
```

Discovering whether a MIDlet was push-activated

- The *PushRegistry.listConnections()* method allows you to discover all the inbound connections registered by the MIDlet suite. You can also use it to discover whether the MIDlet was activated by an incoming connection.
- If you pass *false* to *listConnections()*, the method returns a string array that identifies all the inbound connections registered by the MIDlet suite, but if you pass a *true* argument the method reports only the registered connections *with available data* - indicating that MIDlet activation was due to incoming network data .

```
private boolean handlePushActivation() {
    //  Discover if there are pending push inbound
    //  connections and if so, dispatch a PushProcessor for each one.
    String[] connections = PushRegistry.listConnections(true);
    if (connections != null && connections.length > 0) {
        for (int i=0; i < connections.length; i++)
          PushProcessor pp = new PushProcessor(connections[i]);
        return(true);
    }
    return(false);
}
```

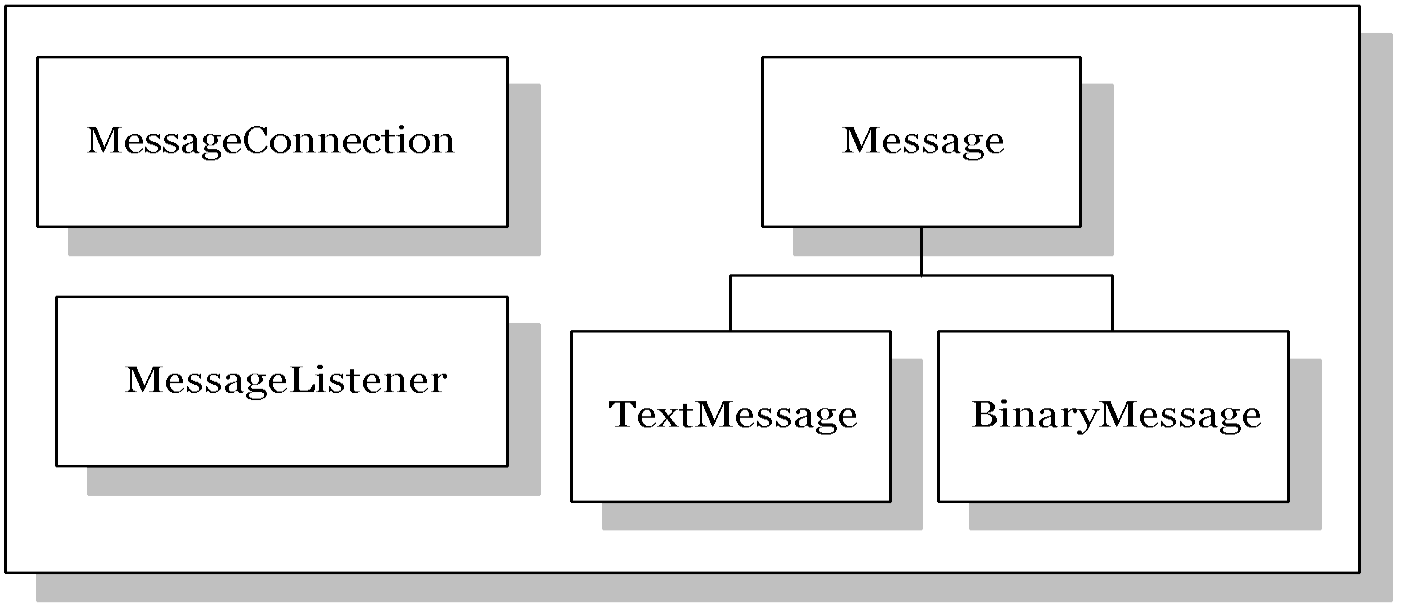
Managing connections

```
class PushProcessor implements Runnable {
    Thread th = new Thread(this);
    String url;
    String midletClassName;
    public PushProcessor(String url) { this.url = url; th.start(); }
    public void run() {
      ServerSocketConnection ssc = null;
      SocketConnection sc = null;
      InputStream is = null;
      try {
        //  "Open" connection.
        ssc = (ServerSocketConnection)Connector.open(url);
        //  Wait for (and accept) inbound connection.
        sc = (SocketConnection) ssc.acceptAndOpen();
        is = sc.openInputStream();
        //  Read data from inbound connection.
        int ch;
        byte[] data = null;
        ByteArrayOutputStream tmp = new ByteArrayOutputStream();
        while( ( ch = is.read() ) != -1 ) {
          tmp.write( ch );
        }
        data = tmp.toByteArray();
        } catch (Exception e){// Exception handling ...}
    }
} //  PushProcessor
```

Wireless Messaging API

WMA

- The WMA is an optional package based on the Generic Connection Framework (GCF) and targets the Connected Limited Device Configuration (CLDC). It thus supports JME applications targeted at cell phones and other devices that can send and receive wireless messages.
- All the WMA components are contained in a single package, *javax.wireless.messaging*, which defines all the interfaces required for sending and receiving wireless messages, both binary and text.



Message

- The interface *javax.wireless.messaging.Message* is the base for all types of messages communicated using the WMA - a *Message* is what is sent and received, produced and consumed.
- In some respects, a *Message* looks similar to a *Datagram*: it has source and destination addresses, a payload, and ways to send and block for a message. The WMA provides additional functionality, such as support for binary and text messages and a listener interface for receiving messages asynchronously.
- The WMA defines two subinterfaces, *BinaryMessage* and *TextMessage*, and the specification is extensible, to allow for support of additional message types.
- How messages (and related control information) are encoded for transmission is protocol-specific and transparent to the WMA.
- Methods: *String getAddress()*, *Date getTimestamp()*, *void setAddress(String a)*.

Binary and Text Messages

- The *BinaryMessage* subinterface represents a message with a binary payload, and declares methods to set and get it. General methods to set and get the address of the message and get its time stamp are inherited from *Message*.
  - *byte[] getPayloadData()*
  - *void setPayloadData(byte[] data)*
- The *TextMessage* subinterface represents a message with a text payload, such as an SMS-based short text message. The *TextMessage* interface provides methods to set and get text payloads (instances of *String*). Before the text message is sent or received, the underlying implementation is responsible for properly encoding or decoding the *String* to or from the appropriate format, for example GSM 7-bit. General methods to set and get the address of the message and get its time stamp are inherited from *Message*.
  - *String getPayloadText()*
  - *void setPayloadText(String data)*

MessageConnection

- The *MessageConnection* interface is a subinterface of the Generic Connection Framework's *Connection*.
- As with any other GCF connection, to create a *Connection* (in this context a *MessageConnection*) call *Connector.open()*, and to close it call *Connection.close()*. You can have multiple *MessageConnection*s open simultaneously.
- A *MessageConnection* can be created in one of two modes: as a client connection or as a server connection. As a client it can only send messages, while server connections can both send and receive messages. You specify a connection as either client or server the same way as when making other GCF connections, by way of the URL. A URL for a client connection includes a destination address, as in:

MessageConnection mc = (MessageConnection)Connector.open("sms://ˆ+5121234567:5000");

- And a URL for a server connection specifies a local address (no host, just a protocol-specific local address, typically a port number), as in: *(MessageConnection)Connector.open("sms://:5000");*
- Trying to bind to an already reserved local address causes an IOException to be thrown.

MessageConnection

- *MessageConnection* provides the methods to create, send, and receive Message objects:
  - *Message newMessage(String type)*
  - *Message newMessage(String type, String address)*
  - *Message receive()*
  - *void send(Message msg)*

- This interface also defines two *String* constants, *BINARY_MESSAGE* and T*EXT_MESSAGE*, one of which is passed to the *newMessage()* factory method to create the appropriate *Message* object type.

Message Listener

- Receiving messages:
  - only a server *MessageConnection* can receive messages;
  - to wait for messages you normally dispatch a thread during initialization;
  - this thread invokes *receive()* on a *MessageConnection* to wait for messages.
- Notification of incoming messages:
  - The *MessageListener* interface is used to implement the Listener design pattern for receiving *Message* objects asynchronously; that is, without blocking while waiting for messages.
  - This interface comprises a single method:

    void notifyIncomingMessage(MessageConnection mc) that is invoked by the platform each time a message is received.

Message Listener

- To set up a message listener you must perform a number of steps.

  1) Define your application (MIDlet) as implementing the *MessageListener* interface (*public class WMAMIDlet extends MIDlet implements MessageListener...*)

  2) Define a *notifyIncomingMessage()* method within your MIDlet class

     you have to minimize the amount of processing within this method (otherwise, you can create a thread).

  3) Register the midlet as the message listener for the server connection:

```
...
  // Open the messaging inbound port.
  mc = new MessageConnection("sms://:5000");
  // Register the listener for inbound messages.
  mc.setMessageListener(this); // this is the reference to midlet
...
```