16.1 Introduction

- Message-oriented middleware
  - enables components to post messages for other components
  - two types
    - *point-to-point messaging model*
      - components send messages to *message queue*
        - messages sent to one consumer
    - *publish/subscribe messaging model*
      - components *publish* message to *topic* on server
        - multiple subscribers receive message for given topic

16.1 Introduction (cont.)

- Message
  - composed of
    - header
      - message destination
      - sending time
    - properties (optional)
      - server
        - determines type of message being sent
      - clients
        - helps determine what messages to receive
    - body
      - content of message

16.1 Introduction (cont.)

- composed of 5 types
    1. **BytesMessages**
    2. **MapMessages**
    3. **ObjectMessages**
    4. **StreamMessages**
    5. **TextMessages**

- Message-driven beans
    - Enterprise JavaBeans that support messaging
    - EJB container uses any message-driven bean for given topic
        - message-driven beans cannot maintain clients state
    - enable components to receive messages asynchronously

16.3 Point-To-Point Messaging

- Allows clients send messages to message queue.
  - receiver connects to queue to consume non-consumed messages
- Messages intended for one receiver.
- Messages stored in queue until client consumes messages.

16.3 Point-To-Point Messaging (cont.)

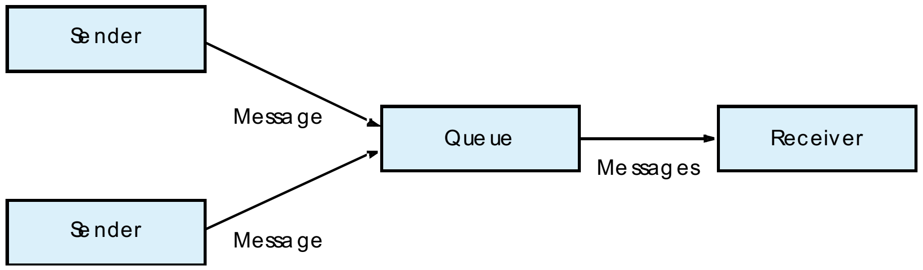


Fig. 16.2 Point-to-point messaging model.

16.3.1 Voter Application: Overview

- Tallies votes to favorite computer languages.
- Class **`Voter`**
  - sends votes as messages to **`Votes`** queue
    - messages are simple **`TextMessage`** objects
      - body contains candidate name
- Class **`VoteCollector`**
  - consumes messages and tallies votes
  - updates display

16.3.1 Voter Application: Overview (cont.)

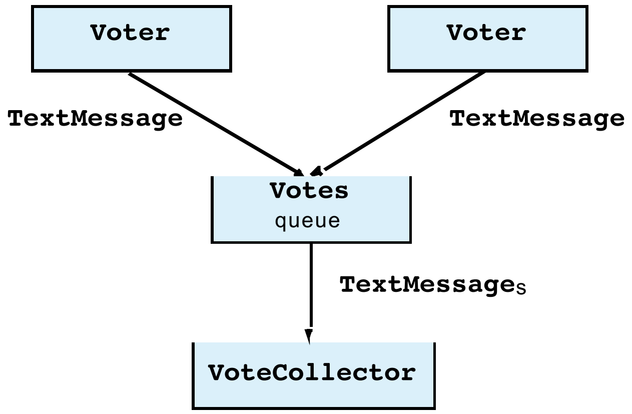


Fig. 16.3 Voter application overview.

16.3.2 Voter Application: Sender Side

- Consists of single class, **`Voter`**.
  - allows user to select programming language
  - sends vote to **`Votes`** queue

Outline

```
1     // Voter.java
2     // Voter is the GUI that allows the client to vote
3     // for a programming language. Voter sends the vote
4     // to the "Votes" queue as a TextMessage.
5     package com.deitel.advjhtp1.jms.voter;
6
7     // Java core packages
8     import java.awt.*;
9     import java.awt.event.*;
10
11    // Java extension packages
12    import javax.swing.*;
13    import javax.jms.*;
14    import javax.naming.*;
15
16    public class Voter extends JFrame {
17
18       private String selectedLanguage;
19
20       // JMS variables
21       private QueueConnection queueConnection;
22       private QueueSession queueSession;
23       private QueueSender queueSender;
24
25       // Voter constructor
26       public Voter()
27       {
28          // lay out user interface
29          super( "Voter" );
30
31          Container container = getContentPane();
32          container.setLayout( new BorderLayout() );
33
```

contains JMS API classes and interfaces

Fig. 16.4 `Voter` class submits votes as messages to queue.

Line 13

```
         JTextArea voteArea =
            new JTextArea( "Please vote for your\n" +
               "favorite programming language" );
         voteArea.setEditable( false );
         container.add( voteArea, BorderLayout.NORTH );

         JPanel languagesPanel = new JPanel();
         languagesPanel.setLayout( new GridLayout( 0, 1 ) );

         // add each language as its own JCheckBox
         // ButtonGroup ensures exactly one language selected
         ButtonGroup languagesGroup = new ButtonGroup();
         CheckBoxHandler checkBoxHandler = new CheckBoxHandler();
         String languages[] =
            { "C", "C++", "Java", "Lisp", "Python" };
         selectedLanguage = "";

         // create JCheckBox for each language
         // and add to ButtonGroup and JPanel
         for ( int i = 0; i < languages.length; i++ ) {
            JCheckBox checkBox = new JCheckBox( languages[ i ] );
            checkBox.addItemListener( checkBoxHandler );
            languagesPanel.add( checkBox );
            languagesGroup.add( checkBox );
         }

         container.add( languagesPanel, BorderLayout.CENTER );

         // create button to submit vote
         JButton submitButton = new JButton( "Submit vote!" );
         container.add( submitButton, BorderLayout.SOUTH );

```

Fig. 16.4 `Voter` class submits votes as messages to queue.

```
66          // invoke method submitVote when submitButton clicked
67          submitButton.addActionListener (
68
69             new ActionListener() {
70
71                public void actionPerformed ( ActionEvent event ) {
72                   submitVote();
73                }
74             }
75          );
76
77          // invoke method quit when window closed
78          addWindowListener(
79
80             new WindowAdapter() {
81
82                public void windowClosing( WindowEvent event ) {
83                   quit();
84                }
85             }
86          );
87
88          // connect to message queue
89          try {
90
91             // create JNDI context
92             Context jndiContext = new InitialContext();
93
94             // retrieve queue connection factory and
95             // queue from JNDI context
96             QueueConnectionFactory queueConnectionFactory =
97                ( QueueConnectionFactory )
98                jndiContext.lookup( "VOTE_FACTORY" );
99             Queue queue = ( Queue ) jndiContext.lookup( "Votes" );
100
```

Fig. 16.4 `Voter` class submits votes as messages to queue.

Line 92

Lines 96-100

create JNDI context

server administrator responsible for creating queue connection factory and queue

Outline

Fig. 16.4 `Voter` class submits votes as messages to queue

```
101            // create connection, session and sender
102            queueConnection =
103               queueConnectionFactory.createQueueConnection();
104            queueSession =
105               queueConnection.createQueueSession( false,
106                  Session.AUTO_ACKNOWLEDGE );
107            queueSender = queueSession.createSender( queue );
108         }
109
110         // process Naming exception from JNDI context
111         catch ( NamingException namingException ) {
112            namingException.printStackTrace();
113            System.exit( 1 );
114         }
115
116         // process JMS exception from queue connection or session
117         catch ( JMSException jmsException ) {
118            jmsException.printStackTrace();
119            System.exit( 1 );
120         }
121
122      } // end Voter constructor
123
124      // submit selected vote to "Votes" queue as TextMessage
125      public void submitVote()
126      {
127         if ( selectedLanguage != "" ) {
128
129            // create text message containing selected language
130            try {
131               TextMessage voteMessage =
132                  queueSession.createTextMessage();
133               voteMessage.setText( selectedLanguage );
134
```

create

create `QueueSession`

Lines 104-106

post messages through `QueueSender` instance

Lines 131-132

message instance

Line 133

set body of message

Outline

```
135               // send the message to the queue
136               queueSender.send( voteMessage );
137            }
138
139            // process JMS exception
140            catch ( JMSException jmsException ) {
141               jmsException.printStackTrace();
142            }
143         }
144
145      } // end method submitVote
146
147      // close client application
148      public void quit()
149      {
150         if ( queueConnection != null ) {
151
152            // close queue connection if it exists
153            try {
154               queueConnection.close();
155            }
156
157            // process JMS exception
158            catch ( JMSException jmsException ) {
159               jmsException.printStackTrace();
160            }
161         }
162
163         System.exit( 0 );
164
165      } // end method quit
166
```

send message

close connection to queue

Fig. 16.4 `Voter` class submits votes as ue.

Line 136

Line 154

```
   // launch Voter application
   public static void main( String args[] )
   {
      Voter voter = new Voter();
      voter.pack();
      voter.setVisible( true );
   }

   // CheckBoxHandler handles event when checkbox checked
   private class CheckBoxHandler implements ItemListener {

      // checkbox event
      public void itemStateChanged( ItemEvent event )
      {
         // update selectedLanguage
         JCheckBox source = ( JCheckBox ) event.getSource();
         selectedLanguage = source.getText();
      }
   }
}
```

Outline

Fig. 16.4 `Voter` class submits votes as messages to queue.

16.3.2 Voter Application: Sender Side (cont.)

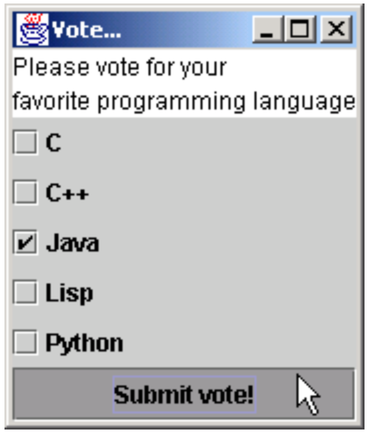


Fig. 16.5 Voter application votes for favorite programming language

16.3.3 Voter Application: Receiver Side

- Class **`VoteCollector`** intended receiver
  - tallies and displays votes
- **`Votes`** queue can be populated before **`VoteCollector`** connects.

```
// VoteCollector.java
// VoteCollector tallies and displays the votes
// posted as TextMessages to the "Votes" queue.
package com.deitel.advjhtp1.jms.voter;

// Java core packages
import java.awt.*;
import java.awt.event.*;
import java.util.*;

// Java extension packages
import javax.swing.*;
import javax.jms.*;
import javax.naming.*;

public class VoteCollector extends JFrame {

   private JPanel displayPanel;
   private Map tallies = new HashMap();

   // JMS variables
   private QueueConnection queueConnection;

   // VoteCollector constructor
   public VoteCollector()
   {
      super( "Vote Tallies" );

      Container container = getContentPane();

      // displayPanel will display tally results
      displayPanel = new JPanel();
      displayPanel.setLayout( new GridLayout( 0, 1 ) );
      container.add( new JScrollPane( displayPanel ) );

```

Fig. 16.6 `VoteCollector` class retrieves and tallies votes.

Outline

Fig. 16.6 `VoteCollector` class retrieves and tallies votes.

Line 51

Lines 55-57

Lines 66-67

```
36          // invoke method quit when window closed
37          addWindowListener(
38
39             new WindowAdapter() {
40
41                public void windowClosing( WindowEvent event ) {
42                   quit();
43                }
44             }
45          );
46
47          // connect to "Votes" queue
48          try {
49
50             // create JNDI context
51             Context jndiContext = new InitialContext();
52
53             // retrieve queue connection factory
54             // and queue from JNDI context
55             QueueConnectionFactory queueConnectionFactory =
56                ( QueueConnectionFactory )
57                jndiContext.lookup( "VOTE_FACTORY" );
58             Queue queue = ( Queue ) jndiContext.lookup( "Votes" );
59
60             // create connection, session and receiver
61             queueConnection =
62                queueConnectionFactory.createQueueConnection();
63             QueueSession queueSession =
64                queueConnection.createQueueSession( false,
65                   Session.AUTO_ACKNOWLEDGE );
66             QueueReceiver queueReceiver =
67                queueSession.createReceiver( queue );
68
```

create JNDI context

get queue connection factory

create `QueueSession`

create votes receiver

Fig. 16.6 `VoteCollector` nd tallies votes.

Lines 70-71

Lines 74

Lines 92-109

```
69            // initialize and set message listener
70            queueReceiver.setMessageListener(
71               new VoteListener( this ) );
72
73            // start connection
74            queueConnection.start();
75         }
76
77         // process Naming exception from JNDI context
78         catch ( NamingException namingException ) {
79            namingException.printStackTrace();
80            System.exit( 1 );
81         }
82
83         // process JMS exception from queue connection or session
84         catch ( JMSException jmsException ) {
85            jmsException.printStackTrace();
86            System.exit( 1 );
87         }
88
89      } // end VoteCollector constructor
90
91      // add vote to corresponding tally
92      public void addVote( String vote )
93      {
94         if ( tallies.containsKey( vote ) ) {
95
96            // if vote already has corresponding tally
97            TallyPanel tallyPanel =
98               ( TallyPanel ) tallies.get( vote );
99            tallyPanel.updateTally();
100        }
101
```

activate connection

register listener

updates tallies and display. Callback method for **`VoteListener`** instance

```
102          // add to GUI and tallies
103          else {
104             TallyPanel tallyPanel = new TallyPanel( vote, 1 );
105             displayPanel.add( tallyPanel );
106             tallies.put( vote, tallyPanel );
107             validate();
108          }
109       }
110 
111       // quit application
112       public void quit()
113       {
114          if ( queueConnection != null ) {
115 
116             // close the queue connection if it exists
117             try {
118                queueConnection.close();
119             }
120 
121             // process JMS exception
122             catch ( JMSException jmsException ) {
123                jmsException.printStackTrace();
124                System.exit( 1 );
125             }
126 
127          }
128 
129          System.exit( 0 );
130 
131       } // end method quit
132 
```

close queue connection

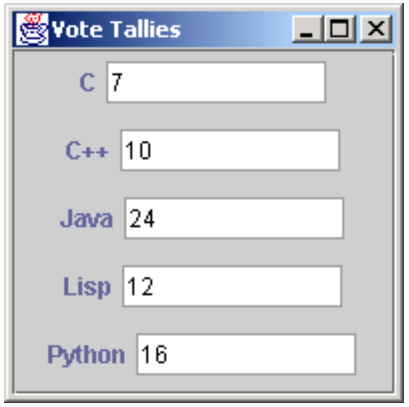
Fig. 16.6 **`VoteCollector`** **class retrieves and tallies votes.**

Line 118

```
    // launch VoteCollector
    public static void main( String args[] )
    {
       VoteCollector voteCollector = new VoteCollector();
       voteCollector.setSize( 200, 200 );
       voteCollector.setVisible( true );
    }
 }
```

Fig. 16.6 `VoteCollector` class retrieves and tallies votes.

16.3.3 Voter Application: Receiver Side (cont.)



C	7
C++	10
Java	24
Lisp	12
Python	16

Fig. 16.7 **`VoteCollector`** tallies and displays votes.

```
1     // VoteListener.java
2     // VoteListener is the message listener for the
3     // receiver of the "Votes" queue. It implements
4     // the specified onMessage method to update the
5     // GUI with the received vote.
6     package com.deitel.advjhtp1.jms.voter;
7
8     // Java extension packages
9     import javax.jms.*;
10
11    public class VoteListener implements MessageListener {
12
13       private VoteCollector voteCollector;
14
15       // VoteListener constructor
16       public VoteListener( VoteCollector collector )
17       {
18          voteCollector = collector;
19       }
20
21       // receive new message
22       public void onMessage( Message message )
23       {
24          TextMessage voteMessage;
25
26          // retrieve and process message
27          try {
28
29             if ( message instanceof TextMessage ) {
30                voteMessage = ( TextMessage ) message;
31                String vote = voteMessage.getText();
32                voteCollector.addVote( vote );
33
34                System.out.println( "Received vote: " + vote );
35             }
```

implements **`MessageListener`** interface

ensure message of type **`TextMessage`**

call back

Fig. 16.8 `VoteListener` class receives messages from the queue.

Line 11

29

32

```
            else {
               System.out.println( "Expecting " +
                  "TextMessage object, received " +
                  message.getClass().getName() );
            }
         }

         // process JMS exception from message
         catch ( JMSException jmsException ) {
            jmsException.printStackTrace();
         }

      } // end method onMessage
   }
```

Outline

Fig. 16.8 `VoteListener` class receives messages from the queue.

Outline

```java
// TallyPanel.java
// TallPanel is the GUI component which displays
// the name and tally for a vote candidate.
package com.deitel.advjhtp1.jms.voter;

// Java core packages
import java.awt.*;

// Java extension packages
import javax.swing.*;

public class TallyPanel extends JPanel {

   private JLabel nameLabel;
   private JTextField tallyField;
   private String name;
   private int tally;

   // TallyPanel constructor
   public TallyPanel( String voteName, int voteTally )
   {
      name = voteName;
      tally = voteTally;

      nameLabel = new JLabel( name );
      tallyField =
         new JTextField( Integer.toString( tally ), 10 );
      tallyField.setEditable( false );
      tallyField.setBackground( Color.white );

      add( nameLabel );
      add( tallyField );

   } // end TallyPanel constructor

```

Fig. 16.9
`TallyPanel` class displays candidate name and tally.

```
    // update tally by one vote
    public void updateTally()
    {
       tally++;
       tallyField.setText( Integer.toString( tally ) );
    }
 }
```

increments tally by one

Outline

Fig. 16.9 `TallyPanel` class displays candidate name and tally.

16.3.4 Voter Application: Configuring and Running

1. Start J2EE server

```
j2ee -verbose
```

2. Create Votes queue (in new window)

```
j2eeadmin -addJmsDestination Votes queue
```

3. Verify queue was created

```
j2eeadmin -listJmsDestination
```

4. Create connection factory

```
j2eeadmin -addJmsFactory VOTE_FACTORY queue
```

5. Start VoteCollector

```
java -classpath %J2EE_HOME%\lib\j2ee.jar;.
 -Djms.properties=%J2EE_HOME%\config\jms_client.properties
 com.deitel.advjhtp1.jms.voter.VoteCollector
```

6. Start Voter (in new window)

```
java -classpath %J2EE_HOME%\lib\j2ee.jar;.
 -Djms.properties=%J2EE_HOME%\config\jms_client.properties
 com.deitel.advjhtp1.jms.voter.Voter
```

16.3.4 Voter Application: Configuring and Running (cont.)

- Once application finished
  - remove connection factory

    ```
    j2eeadmin -removeJmsFactory VOTE_FACTORY
    ```
  - remote topic

    ```
    j2eeadmin -removeJmsDestination Votes
    ```
  - stop J2EE server

    ```
    j2ee -stop
    ```

16.4 Publish/Subscribe Messaging

- Allows multiple clients to
  - connect to topic on server
  - send messages
  - receive messages
- When client publishes message, message sent to all clients subscribed to topic.
- Two subscription types:
  1. *nondurable*
     - messages received while subscriptions active
  2. *durable*
     - server maintains messages while subscription inactive
       - server sends messages when client reactivates

16.4 Publish/Subscribe Messaging (cont.)

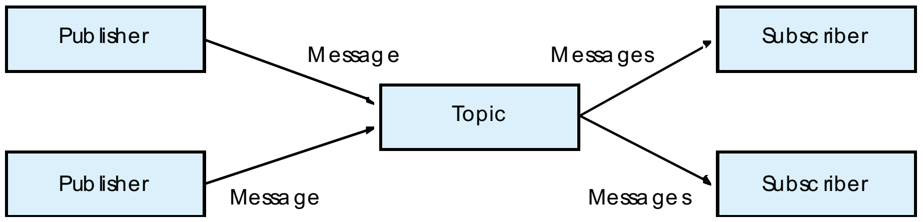


Fig. 16.10 Publish/subscribe messaging model.

16.4.1 Weather Application: Overview

- Class **`WeatherPublisher`**
  - retrieves weather updates from URL
  - publishes information as messages to topic
- Class **`WeatherSubscriber`**
  - provides GUI
    - enables user to select desired cities
  - subscribes to **`Weather`** topic
    - receives corresponding messages
    - uses message selector

16.4.1 Weather Application: Overview (cont.)

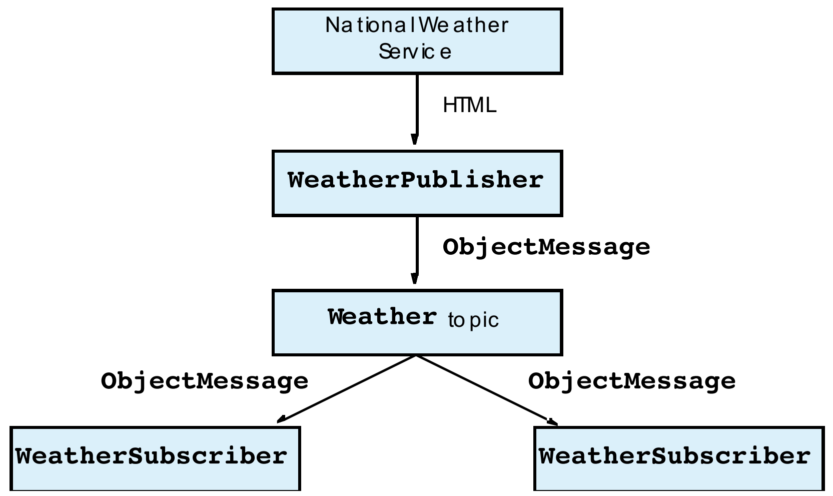


Fig. 16.11 Weather application overview.

16.4.2 Weather Application: Publisher Side

- Class **`WeatherPublisher`**
  - retrieves weather updates from National Weather Service
  - publishes weather updates to **`Weather`** topic
  - messages of type **`ObjectMessage`**
    - **`String`** property **`City`** specifies corresponding city

Outline

```
// WeatherPublisher.java
// WeatherPublisher retrieves weather conditions from the National
// Weather Service and publishes them to the Weather topic
// as ObjectMessages containing WeatherBeans. The city name is
// used in a String property "City" in the message header.
package com.deitel.advjhtp1.jms.weather;
\
// Java core packages
import java.io.*;
import java.net.*;
import java.util.*;

// Java extension packages
import javax.jms.*;
import javax.naming.*;

// Deitel packages
import com.deitel.advjhtp1.rmi.weather.WeatherBean;

public class WeatherPublisher extends TimerTask {

   private BufferedReader in;
   private TopicConnection topicConnection;

   // WeatherPublisher constructor
   public WeatherPublisher()
   {
      // update weather conditions every minute
      Timer timer = new Timer();
      timer.scheduleAtFixedRate( this, 0, 60000 );

      // allow user to quit
      InputStreamReader inputStreamReader =
         new InputStreamReader( System.in );
      char answer = '\0';
```

Fig. 16.12 `WeatherPublisher` class publishes messages to `Weather` topic.

Outline

```

      // loop until user enters q or Q
      while ( !( ( answer == 'q' ) || ( answer == 'Q' ) ) ) {

         // read in character
         try {
            answer = ( char ) inputStreamReader.read();
         }

         // process IO exception
         catch ( IOException ioException ) {
            ioException.printStackTrace();
            System.exit( 1 );
         }

      } // end while

      // close connections
      try {

         // close topicConnection if it exists
         if ( topicConnection != null ) {
            topicConnection.close();
         }

         in.close();  // close connection to NWS Web server
         timer.cancel();  // stop timer
      }

      // process JMS exception from closing topic connection
      catch ( JMSException jmsException ) {
         jmsException.printStackTrace();
         System.exit( 1 );
      }

```

Fig. 16.12 `WeatherPublisher` class publishes messages to `Weather` topic.

Outline

```
71          // process IO exception from closing connection
72          // to NWS Web server
73          catch ( IOException ioException ) {
74             ioException.printStackTrace();
75             System.exit( 1 );
76          }
77
78          System.exit( 0 );
79
80       } // end WeatherPublisher constructor
81
82       // get weather information from NWS
83       public void run()
84       {
85          // connect to topic "Weather"
86          try {
87             System.out.println( "Update weather information..." );
88
89             // create JNDI context
90             Context jndiContext = new InitialContext();
91             String topicName = "Weather";
92
93             // retrieve topic connection factory and topic
94             // from JNDI context
95             TopicConnectionFactory topicConnectionFactory =
96                ( TopicConnectionFactory )
97                jndiContext.lookup( "WEATHER_FACTORY" );
98
99             Topic topic =
100               ( Topic ) jndiContext.lookup( topicName );
101
102            // create connection, session, publisher and message
103            topicConnection =
104               topicConnectionFactory.createTopicConnection();
105
```

create JNDI context

look up **TopicConnectionFactory** and **Topic**

create **TopicConnection**

Fig. 16.12 WeatherPublisher class publishes messages to Weather topic.

Line 90

Line 95-97

Line 103-104

Outline

```
106        TopicSession topicSession =
107           topicConnection.createTopicSession( false,
108              Session.AUTO_ACKNOWLEDGE );
109
110        TopicPublisher topicPublisher =
111           topicSession.createPublisher( topic );
112
113        ObjectMessage message =
114           topicSession.createObjectMessage();
115
116        // connect to National Weather Service
117        // and publish conditions to topic
118
119        // National Weather Service Travelers Forecast page
120        URL url = new URL(
121           "http://iwin.nws.noaa.gov/iwin/us/traveler.html" );
122
123        // set up text input stream to read Web page contents
124        in = new BufferedReader(
125           new InputStreamReader( url.openStream() ) );
126
127        // helps determine starting point of data on Web page
128        String separator = "TAV12";
129
130        // locate separator string in Web page
131        while ( !in.readLine().startsWith( separator ) )
132           ;     // do nothing
133
134        // strings representing headers on Travelers Forecast
135        // Web page for daytime and nighttime weather
136        String dayHeader =
137           "CITY            WEA     HI/LO   WEA     HI/LO";
138
139        String nightHeader =
140           "CITY            WEA     LO/HI   WEA     LO/HI";
```

obtain **TopicSession**

obtain **TopicPublisher**

will contain **WeatherBean** objects

Fig. 16.12 WeatherPublisher cl bli h

Weather topic.

Lines 106-108

Lines 113-114

Outline

```
141
142         String inputLine = "";
143
144         // locate header that begins weather information
145         do {
146            inputLine = in.readLine();
147         }
148
149         while ( !inputLine.equals( dayHeader ) &&
150            !inputLine.equals( nightHeader ) );
151
152         // create WeatherBean objects for each city's data
153         // publish to Weather topic using city as message's type
154         inputLine = in.readLine();  // get first city's info
155
156         // the portion of inputLine containing relevant data is
157         // 28 characters long. If the line length is not at
158         // least 28 characters long, done processing data.
159         while ( inputLine.length() > 28 ) {
160
161            // create WeatherBean object for city
162            // first 16 characters are city name
163            // next six characters are weather description
164            // next six characters are HI/LO temperature
165            WeatherBean weather = new WeatherBean(
166               inputLine.substring( 0, 16 ).trim(),
167               inputLine.substring( 16, 22 ).trim(),
168               inputLine.substring( 23, 29 ).trim() );
169
170            // publish WeatherBean object with city name
171            // as a message property,
172            // used for selection by clients
173            message.setObject( weather );
174            message.setStringProperty( "City",
175               weather.getCityName() );
```

Fig. 16.12 `WeatherPublisher` class publishes messages to `Weather` topic.

Lines 165-168

Line 173

Lines 174-175

create **`WeatherBean`** object

store **`WeatherBean`** i

set **`City`** property

```
176             topicPublisher.publish( message );
177
178             System.out.println( "published message for city: "
179                + weather.getCityName() );
180
181             inputLine = in.readLine();  // get next city's info
182          }
183
184          System.out.println( "Weather information updated." );
185
186       } // end try
187
188       // process Naming exception from JNDI context
189       catch ( NamingException namingException ) {
190          namingException.printStackTrace();
191          System.exit( 1 );
192       }
193
194       // process JMS exception from connection,
195       // session, publisher or message
196       catch ( JMSException jmsException ) {
197          jmsException.printStackTrace();
198          System.exit( 1 );
199       }
200
201       // process failure to connect to National Weather Service
202       catch ( java.net.ConnectException connectException ) {
203          connectException.printStackTrace();
204          System.exit( 1 );
205       }
206
207       // process other exceptions
208       catch ( Exception exception ) {
209          exception.printStackTrace();
210          System.exit( 1 );
```

Fig. 16.12 WeatherPublisher class publishes

publish message to topic

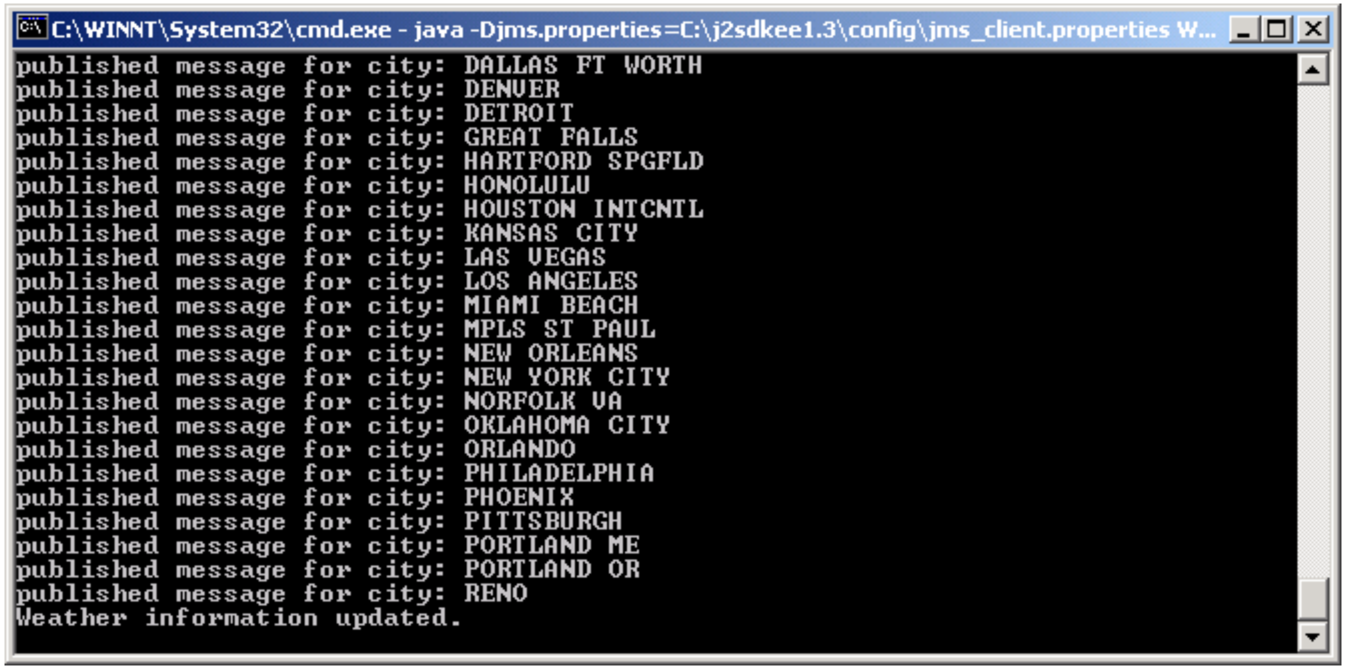
Weather topic.

Line 176

```
         }

      } // end method run

      // launch WeatherPublisher
      public static void main( String args[] )
      {
         System.err.println( "Initializing server...\n" +
            "Enter 'q' or 'Q' to quit" );

         WeatherPublisher publisher = new WeatherPublisher();
      }
   }
```

Fig. 16.12 `WeatherPublisher` class publishes messages to `Weather` topic.

16.4.2 Weather Application: Publisher Side (cont.)


C:\WINNT\System32\cmd.exe - java -Djms.properties=C:\j2sdkee1.3\config\jms_client.properties W...

```
published message for city: DALLAS FT WORTH
published message for city: DENVER
published message for city: DETROIT
published message for city: GREAT FALLS
published message for city: HARTFORD SPGFLD
published message for city: HONOLULU
published message for city: HOUSTON INTCNTL
published message for city: KANSAS CITY
published message for city: LAS VEGAS
published message for city: LOS ANGELES
published message for city: MIAMI BEACH
published message for city: MPLS ST PAUL
published message for city: NEW ORLEANS
published message for city: NEW YORK CITY
published message for city: NORFOLK VA
published message for city: OKLAHOMA CITY
published message for city: ORLANDO
published message for city: PHILADELPHIA
published message for city: PHOENIX
published message for city: PITTSBURGH
published message for city: PORTLAND ME
published message for city: PORTLAND OR
published message for city: RENO
Weather information updated.
```

Fig. 16.13 **WeatherPublisher** publishing weather update messages.

16.4.3 Weather Application: Subscriber Side

- Subscribes to Weather topic.
  - receives weather updates for selected cities
- Presents GUI for user.

Outline

```
// WeatherSubscriber.java
// WeatherSubscriber presents a GUI for the client to request
// weather conditions for various cities. The WeatherSubscriber
// retrieves the weather conditions from the Weather topic;
// each message body contains a WeatherBean object. The message
// header contains a String property "City," which allows
// the client to select the desired cities.
package com.deitel.advjhtp1.jms.weather;

// Java core packages
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;
import javax.naming.*;
import javax.jms.*;

public class WeatherSubscriber extends JFrame {

   // GUI variables
   private WeatherDisplay weatherDisplay;
   private JList citiesList;

```

Fig. 16.14 `WeatherSubscriber` class allows user to receive weather updates.

Outline

```
25      // cities contains cities for which weather
26      // updates are available on "Weather" topic
27      private String cities[] = { "ALBANY NY", "ANCHORAGE",
28         "ATLANTA", "ATLANTIC CITY", "BOSTON", "BUFFALO",
29         "BURLINGTON VT", "CHARLESTON WV", "CHARLOTTE", "CHICAGO",
30         "CLEVELAND", "DALLAS FT WORTH", "DENVER", "DETROIT",
31         "GREAT FALLS", "HARTFORD SPGFLD", "HONOLULU",
32         "HOUSTON INTCNTL", "KANSAS CITY", "LAS VEGAS",
33         "LOS ANGELES", "MIAMI BEACH", "MPLS ST PAUL", "NEW ORLEANS",
34         "NEW YORK CITY", "NORFOLK VA", "OKLAHOMA CITY", "ORLANDO",
35         "PHILADELPHIA", "PHOENIX", "PITTSBURGH", "PORTLAND ME",
36         "PORTLAND OR", "RENO" };
37
38      // JMS variables
39      private TopicConnection topicConnection;
40      private TopicSession topicSession;
41      private Topic topic;
42      private TopicSubscriber topicSubscriber;
43      private WeatherListener topicListener;
44
45      // WeatherSubscriber constructor
46      public WeatherSubscriber()
47      {
48         super( "JMS WeatherSubscriber..." );
49         weatherDisplay = new WeatherDisplay();
50
51         // set up JNDI context and JMS connections
52         try {
53
54            // create JNDI context
55            Context jndiContext = new InitialContext();
56
```

get JNDI context

Fig. 16.14 **`WeatherSubscriber`** class allows user to receive weather updates.

Line 55

Outline

Fig. 16.14
`WeatherSubscribe`
`r` class allows user to
eceive weather
updates.

```
57            // retrieve topic connection factory
58            // from JNDI context
59            TopicConnectionFactory topicConnectionFactory =
60               ( TopicConnectionFactory ) jndiContext.lookup(
61                  "WEATHER_FACTORY" );
62
63            // retrieve topic from JNDI context
64            String topicName = "We
65            topic = ( Topic ) jndi
66
67            // create topic connection
68            topicConnection =
69               topicConnectionFactory.createTopicConnection();
70
71            // create topic session
72            topicSession = topicConnection.createTopicSession( false,
73               Session.AUTO_ACKNOWLEDGE );
74
75            // initialize listener
76            topicListener = new WeatherListener( weatherDisplay );
77         }
78
79         // process Naming exception from JNDI context
80         catch ( NamingException namingException ) {
81            namingException.printStackTrace();
82         }
83
84         // process JMS exceptions from topic connection or session
85         catch ( JMSException jmsException ) {
86            jmsException.printStackTrace();
87         }
88
89         // lay out user interface
90         Container container = getContentPane();
91         container.setLayout( new BorderLayout() );
```

obtain **`TopicConnectionFactory`**

create **`Topic`**

create **`TopicConnection`**

Line 65

create **`TopicSession`**

Lines 72-73

initialize **`WeatherListener`**

```

      JPanel selectionPanel = new JPanel();
      selectionPanel.setLayout( new BorderLayout() );

      JLabel selectionLabel = new JLabel( "Select Cities" );
      selectionPanel.add( selectionLabel, BorderLayout.NORTH );

      // create list of cities for which users
      // can request weather updates
      citiesList = new JList( cities );
      selectionPanel.add( new JScrollPane( citiesList ),
         BorderLayout.CENTER );

      JButton getWeatherButton = new JButton( "Get Weather..." );
      selectionPanel.add( getWeatherButton, BorderLayout.SOUTH );

      // invoke method getWeather when getWeatherButton clicked
      getWeatherButton.addActionListener (

         new ActionListener() {

            public void actionPerformed ( ActionEvent event )
            {
               getWeather();
            }
         }

      ); // end call to addActionListener

      container.add( selectionPanel, BorderLayout.WEST );
      container.add( weatherDisplay, BorderLayout.CENTER );

      // invoke method quit when window closed
      addWindowListener(

```

Fig. 16.14 `WeatherSubscriber` class allows user to receive weather updates.

Outline

```
127            new WindowAdapter() {
128
129               public void windowClosing( WindowEvent event )
130               {
131                  quit();
132               }
133            }
134
135         ); // end call to addWindowListener
136
137      } // end WeatherSubscriber constructor
138
139      // get weather information for selected cities
140      public void getWeather()
141      {
142         // retrieve selected indices
143         int selectedIndices[] = citiesList.getSelectedIndices();
144
145         if ( selectedIndices.length > 0 ) {
146
147            // if topic subscriber exists, method has
148            // been called before
149            if ( topicSubscriber != null ) {
150
151               // close previous topic subscriber
152               try {
153                  topicSubscriber.close();
154               }
155
156               // process JMS exception
157               catch ( JMSException jmsException ) {
158                  jmsException.printStackTrace();
159               }
160
```

remove previous subscriber so that new subscriber can filter newly selected cities

Fig. 16.14 `WeatherSubscriber` class allows user to receive weather updates.

Line 153

Outline

```
161               // clear previous cities from display
162               weatherDisplay.clearCities();
163            }
164
165            // create message selector to retrieve specified cities
166            StringBuffer messageSelector = new StringBuffer();
167            messageSelector.append(
168               "City = '" + cities[ selectedIndices[ 0 ] ] + "'" );
169
170            for ( int i = 1; i < selectedIndices.length; i++ ) {
171               messageSelector.append( " OR City = '" +
172                  cities[ selectedIndices[ i ] ] + "'" );
173            }
174
175            // create topic subscriber and subscription
176            try {
177               topicSubscriber = topicSession.createSubscriber(
178                  topic, messageSelector.toString(), false );
179               topicSubscriber.setMessageListener( topicListener );
180               topicConnection.start();
181
182               JOptionPane.showMessageDialog( this,
183                  "A weather update should be arriving soon..." );
184            }
185
186            // process JMS exception
187            catch ( JMSException jmsException ) {
188               jmsException.printStackTrace();
189            }
190
191         } // end if
192
193      } // end method getWeather
194
```

create **MessageSelector**

create **TopicSubscriber**

begin receiving messages

Fig. 16.14 WeatherSubscriber class allows user to **updates.**

Lines 166-173

Lines 177-178

```
195    // quit WeatherSubscriber application
196    public void quit()
197    {
198       // close connection and subscription to topic
199       try {
200
201          // close topic subscriber
202          if ( topicSubscriber != null ) {
203             topicSubscriber.close();
204          }
205
206          // close topic connection
207          topicConnection.close();
208       }
209
210       // process JMS exception
211       catch ( JMSException jmsException ) {
212          jmsException.printStackTrace();
213          System.exit( 1 );
214       }
215
216       System.exit( 0 );
217
218    } // end method quit
219
220    // launch WeatherSubscriber application
221    public static void main( String args [] )
222    {
223       WeatherSubscriber subscriber = new WeatherSubscriber();
224       subscriber.pack();
225       subscriber.setVisible( true );
226    }
227 }
```

close subscriber

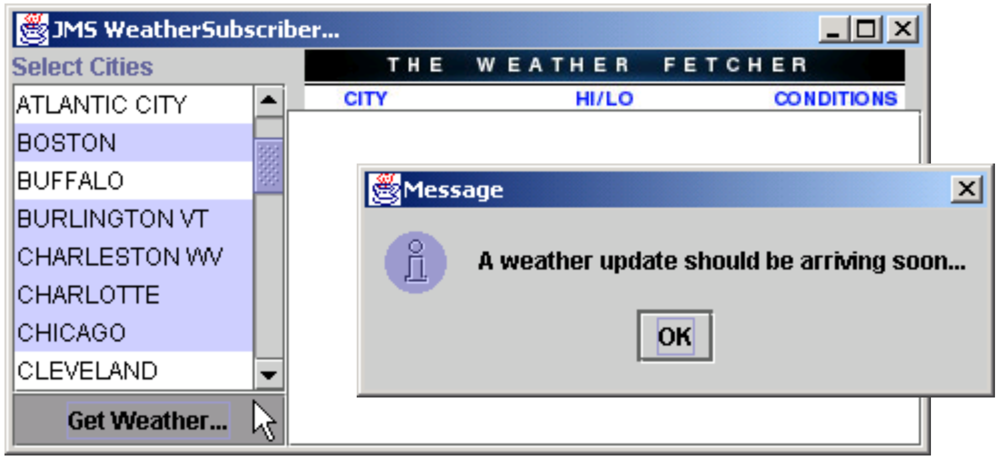
close connection

Fig. 16.14 `WeatherSubscriber` class allows user to receive weather updates.

Line 203

Line 207

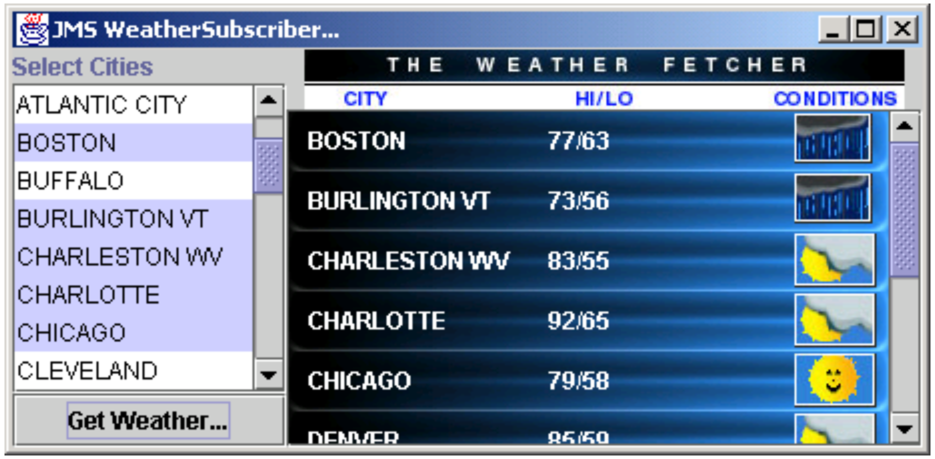
16.4.3 Weather Application: Subscriber Side (cont.)



A weather update should be arriving soon...

Fig. 16.15 **`WeatherSubscriber`** selecting cities for weather updates.

16.4.3 Weather Application: Subscriber Side (cont.)



ATLANTIC CITY
BOSTON
BUFFALO
BURLINGTON VT
CHARLESTON WV
CHARLOTTE
CHICAGO
CLEVELAND

CITY	HI/LO	CONDITIONS
BOSTON	77/63	
BURLINGTON VT	73/56	
CHARLESTON WV	83/55	
CHARLOTTE	92/65	
CHICAGO	79/58	
DENVER	85/59	

Fig. 16.16 **`WeatherSubscriber`** having received updated weather conditions.

```
1     // WeatherListener.java
2     // WeatherListener is the MessageListener for a subscription
3     // to the Weather topic. It implements the specified onMessage
4     // method to update the GUI with the corresponding city's
5     // weather.
6     package com.deitel.advjhtp1.jms.weather;
7
8     // Java extension packages
9     import javax.jms.*;
10    import javax.swing.*;
11
12    // Deitel packages
13    import com.deitel.advjhtp1.rmi.weather.WeatherBean;
14
15    public class WeatherListener implements MessageListener {
16
17       private WeatherDisplay weatherDisplay;
18
19       // WeatherListener constructor
20       public WeatherListener( WeatherDisplay display )
21       {
22          weatherDisplay = display;
23       }
24
25       // receive new message
26       public void onMessage( Message message )
27       {
28          // retrieve and process message
29          try {
30
31             // ensure Message is an ObjectMessage
32             if ( message instanceof ObjectMessage ) {
33
```

Fig. 16.17 `WeatherListener` class subscribes to Weather topic to receive weather forecasts.

Line 15

Line 32

implements **`MessageListener`**

ensures **`message`** of type **`ObjectMessage`**

Outline

Fig. 16.17 `WeatherListener` class subscribes to receive weather forecasts.

Lines 35-38

Line 41

```
34                 // get WeatherBean from ObjectMessage
35                 ObjectMessage objectMessage =
36                    ( ObjectMessage ) message;
37                 WeatherBean weatherBean =
38                    ( WeatherBean ) objectMessage.getObject();
39
40                 // add WeatherBean to display
41                 weatherDisplay.addItem( weatherBean );
42
43              } // end if
44
45              else {
46                 System.out.println( "Expected ObjectMessage," +
47                    " but received " + message.getClass().getName() );
48              }
49
50           } // end try
51
52           // process JMS exception from message
53           catch ( JMSException jmsException ) {
54              jmsException.printStackTrace();
55           }
56
57        } // end method onMessage
58     }
```

obtain **`WeatherBean`**

display contents

Outline

```
// WeatherDisplay.java
// WeatherDisplay extends JPanel to display results
// of client's request for weather conditions.
package com.deitel.advjhtp1.jms.weather;

// Java core packages
import java.awt.*;
import java.awt.event.*;
import java.util.*;

// Java extension packages
import javax.swing.*;

// Deitel packages
import com.deitel.advjhtp1.rmi.weather.*;

public class WeatherDisplay extends JPanel {

   // WeatherListModel and Map for storing WeatherBeans
   private WeatherListModel weatherListModel;
   private Map weatherItems;

   // WeatherDisplay constructor
   public WeatherDisplay()
   {
      setLayout( new BorderLayout() );

      ImageIcon headerImage = new ImageIcon(
         WeatherDisplay.class.getResource(
            "images/header.jpg" ) );
      add( new JLabel( headerImage ), BorderLayout.NORTH );

```

Fig. 16.18 **`WeatherDisplay`** **displays** **`WeatherBeans`** **in a** **`Jlist`** **using a** **`WeatherCellRenderer`**.

Outline

```
33       // use JList to display updated weather conditions
34       // for requested cities
35       weatherListModel = new WeatherListModel();
36       JList weatherJList = new JList( weatherListModel );
37       weatherJList.setCellRenderer( new WeatherCellRenderer() );
38
39       add( new JScrollPane( weatherJList ), BorderLayout.CENTER );
40
41       // maintain WeatherBean items in HashMap
42       weatherItems = new HashMap();
43
44    } // end WeatherDisplay constructor
45
46    // add WeatherBean item to display
47    public void addItem( WeatherBean weather )
48    {
49       String city = weather.getCityName();
50
51       // check whether city is already in display
52       if ( weatherItems.containsKey( city ) ) {
53
54          // if city is in Map, and therefore in display
55          // remove previous WeatherBean object
56          WeatherBean previousWeather =
57             ( WeatherBean ) weatherItems.remove( city );
58          weatherListModel.remove( previousWeather );
59       }
60
61       // add WeatherBean to Map and WeatherListModel
62       weatherListModel.add( weather );
63       weatherItems.put( city, weather );
64
65    } // end method addItem
```

remove if bean previously existing

add new bean to list

Fig. 16.18 `WeatherDisplay` displays `WeatherBeans` in a `Jlist` using a `WeatherCellRenderer`.

Lines 56-58

Lines 62-63

```

   // clear all cities from display
   public void clearCities()
   {
      weatherItems.clear();
      weatherListModel.clear();
   }
}
```

Outline

Fig. 16.18 `WeatherDisplay` displays `WeatherBeans` in a `Jlist` using a `WeatherCellRenderer`.

16.4.4 Weather Application: Configuring and Running

1. Start J2EE server

```
j2ee -verbose
```

2. Create **`Weather`** topic (new window)

```
j2eeadmin -addJmsDestination Weather topic
```

3. Verify topic selected

```
j2eeadmin -listJmsDestination
```

4. Create connection factory

```
j2eeadmin -addJmsFactory WEATHER_FACTORY topic
```

5. Start **`WeatherPublisher`**

```
java -classpath %J2EE_HOME%\lib\j2ee.jar;.
    -Djms.properties=%J2EE_HOME%\config\jms_client.properties
    com.deitel.advjhtp1.jms.weather.WeatherPublisher
```

6. Start **`WeatherSubscriber`**

```
java -classpath %J2EE_HOME%\lib\j2ee.jar;.
     -Djms.properties=%J2EE_HOME%\config\jms_client.properties
    com.deitel.advjhtp1.jmx.weather.WeatherSubscriber
```