

## 24.10 Case Study: DeitelMessenger Server and Client

Chat rooms have become common on the Internet. They provide a central location where users can chat with each other via short text messages. Each participant can see all messages that the other users post, and each user can post messages. This section presents our capstone networking case study, which integrates many of the Java networking, multithreading and Swing GUI features we have learned thus far to build an online chat system. We also introduce **multicasting**, which enables an application to send `DatagramPackets` to groups of clients. After reading this section, you will be able to build more significant networking applications.

### 24.10.1 DeitelMessengerServer and Supporting Classes

`DeitelMessengerServer` (Fig. 24.18) is the heart of the online chat system. This class appears in package `com.deitel.messenger.sockets.server`. Chat clients can participate in a chat by connecting to the `DeitelMessengerServer`. Method `startServer` (lines 20–53) launches `DeitelMessengerServer`. Lines 28–29 create a `ServerSocket` to accept incoming network connections. Recall that the `ServerSocket` constructor takes as its first argument the port on which the server should listen for incoming connections. Interface `SocketMessengerConstants` (Fig. 24.20) declares the port number as the constant `SERVER_PORT` to ensure that the server and the clients use the correct port number.

Lines 35–47 listen continuously for new client connections. Line 38 invokes `ServerSocket` method `accept` to wait for and accept a new client connection. Lines 41–42 create and start a new `MessageReceiver` for the client. Class `MessageReceiver` (Fig. 24.22) of package `com.deitel.messenger.sockets.server` implements `Runnable` and listens for incoming messages from a client. The first argument to the `MessageReceiver` constructor is a `MessageListener` (Fig. 24.21), to which messages from the client should be delivered. Class `DeitelMessengerServer` implements interface `MessageListener` (line 15) of package `com.deitel.messenger` and therefore can pass the `this` reference to the `MessageReceiver` constructor.

When each `MessageReceiver` receives a new message from a client, the `MessageReceiver` passes the message to a `MessageListener` through method `messageReceived` (lines 56–64). Line 59 concatenates the `from` string with the separator `>>>` and the message

```

1 // Fig. 24.18: DeitelMessengerServer.java
2 // DeitelMessengerServer is a multithreaded, socket- and
3 // packet-based chat server.
4 package com.deitel.messenger.sockets.server;
5
6 import java.net.ServerSocket;
7 import java.net.Socket;
8 import java.io.IOException;
9 import java.util.concurrent.Executors;
10 import java.util.concurrent.ExecutorService;
11
12 import com.deitel.messenger.MessageListener;
13 import static com.deitel.messenger.sockets.SocketMessengerConstants.*;

```

**Fig. 24.18** | `DeitelMessengerServer` for managing a chat room. (Part I of 2.)

```

14
15 public class DeitelMessengerServer implements MessageListener
16 {
17     private ExecutorService serverExecutor; // executor for server
18
19     // start chat server
20     public void startServer()
21     {
22         // create executor for server runnables
23         serverExecutor = Executors.newCachedThreadPool();
24
25         try // create server and manage new clients
26         {
27             // create ServerSocket for incoming connections
28             ServerSocket serverSocket =
29                 new ServerSocket( SERVER_PORT, 100 );
30
31             System.out.printf( "%s%d%s", "Server listening on port ",
32                               SERVER_PORT, " ..." );
33
34             // listen for clients constantly
35             while ( true )
36             {
37                 // accept new client connection
38                 Socket clientSocket = serverSocket.accept();
39
40                 // create MessageReceiver for receiving messages from client
41                 serverExecutor.execute(
42                     new MessageReceiver( this, clientSocket ) );
43
44                 // print connection information
45                 System.out.println( "Connection received from: " +
46                                     clientSocket.getInetAddress() );
47             } // end while
48         } // end try
49         catch ( IOException ioException )
50         {
51             ioException.printStackTrace();
52         } // end catch
53     } // end method startServer
54
55     // when new message is received, broadcast message to clients
56     public void messageReceived( String from, String message )
57     {
58         // create String containing entire message
59         String completeMessage = from + MESSAGE_SEPARATOR + message;
60
61         // create and start MulticastSender to broadcast messages
62         serverExecutor.execute(
63             new MulticastSender( completeMessage.getBytes() ) );
64     } // end method messageReceived
65 } // end class DeitelMessengerServer

```

**Fig. 24.18** | DeitelMessengerServer for managing a chat room. (Part 2 of 2.)

```

1 // Fig. 24.19: DeitelMessengerServerTest.java
2 // Test the DeitelMessengerServer class.
3 package com.deitel.messenger.sockets.server;
4
5 public class DeitelMessengerServerTest
6 {
7     public static void main ( String args[] )
8     {
9         DeitelMessengerServer application = new DeitelMessengerServer();
10        application.startServer(); // start server
11    } // end main
12 } // end class DeitelMessengerServerTest

```

```

Server listening on port 12345 ...
Connection received from: /127.0.0.1
Connection received from: /127.0.0.1
Connection received from: /127.0.0.1

```

**Fig. 24.19** | Test class for DeitelMessengerServer.

body. Lines 62–63 create and start a new `MulticastSender` to deliver `completeMessage` to all clients. Class `MulticastSender` (Fig. 24.23) of package `com.deitel.messenger.sockets.server` uses multicasting as an efficient mechanism for sending one message to multiple clients. We discuss the details of multicasting shortly. Method `main` (lines 7–11 of Fig. 24.19) creates a new `DeitelMessengerServer` instance and starts the server.

Interface `SocketMessengerConstants` (Fig. 24.20) declares constants for use in the various classes that make up the Deitel messenger system. Classes can access these static constants by using a static import as shown in Fig. 24.22.

```

1 // Fig. 24.20: SocketMessengerConstants.java
2 // SocketMessengerConstants defines constants for the port numbers
3 // and multicast address in DeitelMessenger
4 package com.deitel.messenger.sockets;
5
6 public interface SocketMessengerConstants
7 {
8     // address for multicast datagrams
9     public static final String MULTICAST_ADDRESS = "239.0.0.1";
10
11     // port for listening for multicast datagrams
12     public static final int MULTICAST_LISTENING_PORT = 5555;
13
14     // port for sending multicast datagrams
15     public static final int MULTICAST_SENDING_PORT = 5554;
16
17     // port for Socket connections to DeitelMessengerServer
18     public static final int SERVER_PORT = 12345;
19

```

**Fig. 24.20** | `SocketMessengerConstants` declares constants for use in the `DeitelMessengerServer` and `DeitelMessenger`. (Part 1 of 2.)

```

20 // String that indicates disconnect
21 public static final String DISCONNECT_STRING = "DISCONNECT";
22
23 // String that separates the user name from the message body
24 public static final String MESSAGE_SEPARATOR = ">>>";
25
26 // message size (in bytes)
27 public static final int MESSAGE_SIZE = 512;
28 } // end interface SocketMessengerConstants

```

**Fig. 24.20** | SocketMessengerConstants declares constants for use in the DeitelMessengerServer and DeitelMessenger. (Part 2 of 2.)

Line 9 declares the String constant MULTICAST\_ADDRESS, which contains the address to which a MulticastSender (Fig. 24.23) should send messages. This address is one of the addresses reserved for multicast, which we describe in the discussion of Fig. 24.23. Line 12 declares the integer constant MULTICAST\_LISTENING\_PORT—the port on which clients should listen for new messages. Line 15 declares the integer constant MULTICAST\_SENDING\_PORT—the port to which a MulticastSender should post new messages at the MULTICAST\_ADDRESS. Line 18 declares the integer constant SERVER\_PORT—the port on which DeitelMessengerServer listens for incoming client connections. Line 21 declares String constant DISCONNECT\_STRING, which is the String that a client sends to DeitelMessengerServer when the user wishes to leave the chat room. Line 24 declares String constant MESSAGE\_SEPARATOR, which separates the user name from the message body. Line 27 specifies the maximum message size in bytes.

Many different classes in the Deitel messenger system receive messages. For example, DeitelMessengerServer receives messages from clients and delivers them to all chat room participants. As we will see, the user interface for each client also receives messages and displays them to the users. Each class that receives messages implements interface MessageListener (Fig. 24.21). The interface (from package com.deitel.messenger) declares method messageReceived, which allows an implementing class to receive chat messages. Method messageReceived takes two string arguments representing the name of the sender and the message body, respectively.

DeitelMessengerServer uses instances of class MessageReceiver (Fig. 24.22) from package com.deitel.messenger.sockets.server to listen for new messages from each

```

1 // Fig. 24.21: MessageListener.java
2 // MessageListener is an interface for classes that wish to
3 // receive new chat messages.
4 package com.deitel.messenger;
5
6 public interface MessageListener
7 {
8     // receive new chat message
9     public void messageReceived( String from, String message );
10 } // end interface MessageListener

```

**Fig. 24.21** | MessageListener interface that declares method messageReceived for receiving new chat messages.

```

1 // Fig. 24.22: MessageReceiver.java
2 // MessageReceiver is a Runnable that listens for messages from a
3 // particular client and delivers messages to a MessageListener.
4 package com.deitel.messenger.sockets.server;
5
6 import java.io.BufferedReader;
7 import java.io.IOException;
8 import java.io.InputStreamReader;
9 import java.net.Socket;
10 import java.net.SocketTimeoutException;
11 import java.util.StringTokenizer;
12
13 import com.deitel.messenger.MessageListener;
14 import static com.deitel.messenger.sockets.SocketMessengerConstants.*;
15
16 public class MessageReceiver implements Runnable
17 {
18     private BufferedReader input; // input stream
19     private MessageListener messageListener; // message listener
20     private boolean keepListening = true; // when false, ends runnable
21
22     // MessageReceiver constructor
23     public MessageReceiver( MessageListener listener, Socket clientSocket )
24     {
25         // set listener to which new messages should be sent
26         messageListener = listener;
27
28         try
29         {
30             // set timeout for reading from client
31             clientSocket.setSoTimeout( 5000 ); // five seconds
32
33             // create BufferedReader for reading incoming messages
34             input = new BufferedReader( new InputStreamReader(
35                 clientSocket.getInputStream() ) );
36         } // end try
37         catch ( IOException ioException )
38         {
39             ioException.printStackTrace();
40         } // end catch
41     } // end MessageReceiver constructor
42
43     // listen for new messages and deliver them to MessageListener
44     public void run()
45     {
46         String message; // String for incoming messages
47
48         // listen for messages until stopped
49         while ( keepListening )
50     {

```

**Fig. 24.22** | MessageReceiver for listening for new messages from DeitelMessengerServer clients in separate threads. (Part I of 3.)

```

51     try
52     {
53         message = input.readLine(); // read message from client
54     } // end try
55     catch ( SocketTimeoutException socketTimeoutException )
56     {
57         continue; // continue to next iteration to keep listening
58     } // end catch
59     catch ( IOException ioException )
60     {
61         ioException.printStackTrace();
62         break;
63     } // end catch
64
65     // ensure non-null message
66     if ( message != null )
67     {
68         // tokenize message to retrieve user name and message body
69         StringTokenizer tokenizer = new StringTokenizer(
70             message, MESSAGE_SEPARATOR );
71
72         // ignore messages that do not contain a user
73         // name and message body
74         if ( tokenizer.countTokens() == 2 )
75         {
76             // send message to MessageListener
77             messageListener.messageReceived(
78                 tokenizer.nextToken(), // user name
79                 tokenizer.nextToken() ); // message body
80         } // end if
81         else
82         {
83             // if disconnect message received, stop listening
84             if ( message.equalsIgnoreCase(
85                 MESSAGE_SEPARATOR + DISCONNECT_STRING ) )
86                 stopListening();
87         } // end else
88     } // end if
89 } // end while
90
91     try
92     {
93         input.close(); // close BufferedReader (also closes Socket)
94     } // end try
95     catch ( IOException ioException )
96     {
97         ioException.printStackTrace();
98     } // end catch
99 } // end method run
100

```

**Fig. 24.22** | MessageReceiver for listening for new messages from DeitelMessengerServer clients in separate threads. (Part 2 of 3.)

```

101 // stop listening for incoming messages
102 public void stopListening()
103 {
104     keepListening = false;
105 } // end method stopListening
106 } // end class MessageReceiver

```

**Fig. 24.22** | MessageReceiver for listening for new messages from DeitelMessengerServer clients in separate threads. (Part 3 of 3.)

client. Class MessageReceiver implements interface Runnable. This enables DeitelMessengerServer to create an object of class MessageReceiver to run in a separate thread for each client, so that messages from multiple clients can be handled concurrently. When DeitelMessengerServer receives a new client connection, DeitelMessengerServer creates a new MessageReceiver for the client, then continues listening for new client connections. The MessageReceiver listens for messages from a single client and passes them back to the DeitelMessengerServer through method messageReceived.

The MessageReceiver constructor (lines 23–41) takes a MessageListener as its first argument. The MessageReceiver will deliver new messages to this listener by invoking its messageReceived method. The MessageReceiver constructor's Socket argument is the connection to a particular client. Line 26 sets the MessageListener to which the MessageReceiver should deliver new messages. Line 31 invokes Socket method `setSoTimeout` with an integer argument of 5000 milliseconds. Reading data from a Socket is a **blocking call**—the current thread does not execute until the read operation completes. Method `setSoTimeout` specifies that if no data is received in the given number of milliseconds, the Socket should issue a `SocketTimeoutException`, which the current thread can catch, then continue executing. This technique prevents the current thread from deadlocking if no more data is available from the Socket. Lines 34–35 create a new `BufferedReader` for the `clientSocket`'s `InputStream`. The MessageReceiver uses this `BufferedReader` to read new messages from the client.

Method `run` (lines 44–99) listens continuously for new messages from the client. Lines 49–89 loop as long as the boolean variable `keepListening` is true. Line 53 invokes `BufferedReader` method `readLine` to read a line of text from the client. If more than 5000 milliseconds pass without any data being read, method `readLine` throws an `InterruptedException`, which indicates that the timeout set on line 31 has expired. Line 57 uses a `continue` statement to go to the next iteration of the `while` loop to continue listening for messages. Lines 59–63 catch an `IOException`, which indicates a more severe problem from method `readLine`. In this case, line 61 prints a stack trace to aid in debugging the application, and line 62 uses keyword `break` to terminate the loop.

When sending a message to the server, the client separates the user's name from the message body with `MESSAGE_SEPARATOR` declared in interface `SocketMessengerConstants`. If no exceptions are thrown when reading data from the client and the message is not null (line 66), lines 69–70 create a new `StringTokenizer` that uses delimiter `MESSAGE_SEPARATOR` to separate each message into two tokens—the sender's user name and the message. Line 74 checks for the proper number of tokens (using `StringTokenizer` method `countTokens`), and lines 77–79 invoke method `messageReceived` of interface `MessageListener` to deliver the new message to the registered `MessageListener`. If the

`StringTokenizer` does not produce two tokens, lines 84–85 check the message to see whether it matches the constant `DISCONNECT_STRING`, which would indicate that the user wishes to leave the chat room. Line 84 uses `String` method `equalsIgnoreCase` to test whether the input `String` equals the disconnect string. This method is equivalent to `String` method `equals`, but it does not consider the case of the letters. This allows the user to type `DISCONNECT`, `disconnect` or even `dISCONNECT` to terminate the connection. If the strings match, line 86 invokes `MessageReceiver` method `stopListening` to terminate the `MessageReceiver`.

Method `stopListening` (lines 102–105) sets `boolean` variable `keepListening` to `false`. This causes the `while` loop condition that starts at line 49 to fail and causes the `MessageReceiver` to close the client `Socket` (line 93). Then method `run` returns, which terminates the `MessageReceiver`'s execution.

`MulticastSender` (Fig. 24.23) delivers `DatagramPackets` containing chat messages to a group of clients. Multicast is an efficient way to send data to many clients without the overhead of broadcasting it to every host on the Internet. To understand multicast, let us look at a real-world analogy—the relationship between a magazine's publisher and its subscribers. The publisher produces a magazine and provides it to a distributor. Customers obtain a subscription and begin receiving the magazine in the mail from the distributor. This communication is quite different from a television broadcast. When a television station produces a television show, the station broadcasts the show throughout a geographical region or perhaps throughout the world by using satellites. Broadcasting a show for 1,000,000 viewers costs no more than broadcasting one for 100 viewers—the signal carrying the broadcast reaches a wide area. However, printing and delivering a magazine to 1,000,000 readers would be much more expensive than for 100 readers. Most publishers could not stay in business if they had to broadcast their magazines to everyone, so they multicast them to a group of subscribers instead.

```

1 // Fig. 24.23: MulticastSender.java
2 // MulticastSender broadcasts a chat message using a multicast datagram.
3 package com.deitel.messenger.sockets.server;
4
5 import java.io.IOException;
6 import java.net.DatagramPacket;
7 import java.net.DatagramSocket;
8 import java.net.InetAddress;
9
10 import static com.deitel.messenger.sockets.SocketMessengerConstants.*;
11
12 public class MulticastSender implements Runnable
13 {
14     private byte[] messageBytes; // message data
15
16     public MulticastSender( byte[] bytes )
17     {
18         messageBytes = bytes; // create the message
19     } // end MulticastSender constructor

```

**Fig. 24.23** | `MulticastSender` for delivering outgoing messages to a multicast group via `DatagramPackets`. (Part 1 of 2.)



```

20
21 // deliver message to MULTICAST_ADDRESS over DatagramSocket
22 public void run()
23 {
24     try // deliver message
25     {
26         // create DatagramSocket for sending message
27         DatagramSocket socket =
28             new DatagramSocket( MULTICAST_SENDING_PORT );
29
30         // use InetAddress reserved for multicast group
31         InetAddress group = InetAddress.getByName( MULTICAST_ADDRESS );
32
33         // create DatagramPacket containing message
34         DatagramPacket packet = new DatagramPacket( messageBytes,
35             messageBytes.length, group, MULTICAST_LISTENING_PORT );
36
37         socket.send( packet ); // send packet to multicast group
38         socket.close(); // close socket
39     } // end try
40     catch ( IOException ioException )
41     {
42         ioException.printStackTrace();
43     } // end catch
44 } // end method run
45 } // end class MulticastSender

```

**Fig. 24.23** | MulticastSender for delivering outgoing messages to a multicast group via DatagramPackets. (Part 2 of 2.)

Using multicast, an application can “publish” DatagramPackets to “subscriber” applications by sending them to a **multicast address**, which is an IP address reserved for multicast. Multicast addresses are in the range from 224.0.0.0 to 239.255.255.255. Addresses starting with 239 are reserved for intranets, so we use one of these (239.0.0.1) in our case study. Clients that wish to receive these DatagramPackets can connect to the appropriate multicast address to join the group of subscribers—the **multicast group**. When an application sends a DatagramPacket to the multicast address, each client in the group receives it. Multicast DatagramPackets, like unicast DatagramPackets (Fig. 24.7), are not reliable—packets are not guaranteed to reach any destination or arrive in any particular order.

Class MulticastSender implements interface Runnable to enable DeitelMessengerServer to send multicast messages in a separate thread. The DeitelMessengerServer creates a MulticastSender with the contents of the message and starts the thread. The MulticastSender constructor (lines 16–19) takes as an argument an array of bytes containing the message.

Method run (lines 22–44) delivers the message to the multicast address. Lines 27–28 create a new DatagramSocket. Recall from Section 24.7 that we use DatagramSockets to send **unicast** DatagramPackets—packets sent from one host directly to another host. Multicast DatagramPackets are sent the same way, except that the address to which they are sent is a multicast address. Line 31 create an InetAddress object for the multicast address, which is declared as a constant in interface SocketMessengerConstants. Lines 34–35

create the `DatagramPacket` containing the message. The first argument to the `DatagramPacket` constructor is the byte array containing the message. The second argument is the length of the byte array. The third argument specifies the `InetAddress` to which the packet should be sent, and the last specifies the port number at which the packet should be delivered to the multicast address. Line 37 sends the packet with `DatagramSocket` method `send`. All clients listening to the multicast address on the proper port will receive this `DatagramPacket`. Line 38 closes the `DatagramSocket`, and the `run` method returns, terminating the `MulticastSender`.

### Executing the `DeitelMessengerServerTest`

To execute the `DeitelMessengerServerTest`, open a **Command Prompt** window and change directories to the location in which package `com.deitel.messenger.sockets.server` resides (i.e., the directory in which `com` is located). Then type

```
java com.deitel.messenger.sockets.server.DeitelMessengerServerTest
```

to execute the server.

## 24.10.2 DeitelMessenger Client and Supporting Classes

The client for the `DeitelMessengerServer` has several components. A class that implements interface `MessageManager` (Fig. 24.24) manages communication with the server. A `Runnable` subclass listens for messages at `DeitelMessengerServer`'s multicast address. Another `Runnable` subclass sends messages from the client to the server. A `JFrame` subclass provides the client's GUI.

Interface `MessageManager` (Fig. 24.24) declares methods for managing communication with `DeitelMessengerServer`. We declare this interface to abstract the base functionality a client needs to interact with a chat server from the underlying communication

```

1 // Fig. 24.24: MessageManager.java
2 // MessageManger is an interface for objects capable of managing
3 // communications with a message server.
4 package com.deitel.messenger;
5
6 public interface MessageManager
7 {
8     // connect to message server and route incoming messages
9     // to given MessageListener
10    public void connect( MessageListener listener );
11
12    // disconnect from message server and stop routing
13    // incoming messages to given MessageListener
14    public void disconnect( MessageListener listener );
15
16    // send message to message server
17    public void sendMessage( String from, String message );
18 } // end interface MessageManager

```

**Fig. 24.24** | `MessageManager` interface that declares methods for communicating with a `DeitelMessengerServer`.

mechanism. This abstraction enables us to provide `MessageManager` implementations that use other network protocols to implement the communication details. For example, if we wanted to connect to a different chat server that did not use multicast `DatagramPackets`, we could implement the `MessageManager` interface with the appropriate network protocols for this alternative messaging server. We would not need to modify any other code in the client, because the client's other components refer only to interface `MessageManager`, not a particular `MessageManager` implementation. Similarly, `MessageManager` methods refer to other components of the client only through interface `MessageListener`, so other client components can change without requiring changes in the `MessageManager` or its implementations. Method `connect` (line 10) connects a `MessageManager` to `DeitelMessengerServer` and routes incoming messages to the appropriate `MessageListener`. Method `disconnect` (line 14) disconnects a `MessageManager` from the `DeitelMessengerServer` and stops delivering messages to the given `MessageListener`. Method `sendMessage` (line 17) sends a new message to `DeitelMessengerServer`.

Class `SocketMessageManager` (Fig. 24.25) implements `MessageManager` (line 18), using `Sockets` and `MulticastSockets` to communicate with `DeitelMessengerServer` and

```

1 // Fig. 24.25: SocketMessageManager.java
2 // SocketMessageManager communicates with a DeitelMessengerServer using
3 // Sockets and MulticastSockets.
4 package com.deitel.messenger.sockets.client;
5
6 import java.net.InetAddress;
7 import java.net.Socket;
8 import java.io.IOException;
9 import java.util.concurrent.Executors;
10 import java.util.concurrent.ExecutorService;
11 import java.util.concurrent.ExecutionException;
12 import java.util.concurrent.Future;
13
14 import com.deitel.messenger.MessageListener;
15 import com.deitel.messenger.MessageManager;
16 import static com.deitel.messenger.sockets.SocketMessengerConstants.*;
17
18 public class SocketMessageManager implements MessageManager
19 {
20     private Socket clientSocket; // Socket for outgoing messages
21     private String serverAddress; // DeitelMessengerServer address
22     private PacketReceiver receiver; // receives multicast messages
23     private boolean connected = false; // connection status
24     private ExecutorService serverExecutor; // executor for server
25
26     public SocketMessageManager( String address )
27     {
28         serverAddress = address; // store server address
29         serverExecutor = Executors.newCachedThreadPool();
30     } // end SocketMessageManager constructor
31

```

**Fig. 24.25** | `SocketMessageManager` implementation of interface `MessageManager` for communicating via `Sockets` and multicast `DatagramPackets`. (Part 1 of 3.)

```

32 // connect to server and send messages to given MessageListener
33 public void connect( MessageListener listener )
34 {
35     if ( connected )
36         return; // if already connected, return immediately
37
38     try // open Socket connection to DeitelMessengerServer
39     {
40         clientSocket = new Socket(
41             InetAddress.getByName( serverAddress ), SERVER_PORT );
42
43         // create Runnable for receiving incoming messages
44         receiver = new PacketReceiver( listener );
45         serverExecutor.execute( receiver ); // execute Runnable
46         connected = true; // update connected flag
47     } // end try
48     catch ( IOException ioException )
49     {
50         ioException.printStackTrace();
51     } // end catch
52 } // end method connect
53
54 // disconnect from server and unregister given MessageListener
55 public void disconnect( MessageListener listener )
56 {
57     if ( !connected )
58         return; // if not connected, return immediately
59
60     try // stop listener and disconnect from server
61     {
62         // notify server that client is disconnecting
63         Runnable disconnecter = new MessageSender( clientSocket, "",
64             DISCONNECT_STRING );
65         Future disconnecting = serverExecutor.submit( disconnecter );
66         disconnecting.get(); // wait for disconnect message to be sent
67         receiver.stopListening(); // stop receiver
68         clientSocket.close(); // close outgoing Socket
69     } // end try
70     catch ( ExecutionException exception )
71     {
72         exception.printStackTrace();
73     } // end catch
74     catch ( InterruptedException exception )
75     {
76         exception.printStackTrace();
77     } // end catch
78     catch ( IOException ioException )
79     {
80         ioException.printStackTrace();
81     } // end catch
82

```

**Fig. 24.25** | SocketMessageManager implementation of interface MessageManager for communicating via Sockets and multicast DatagramPackets. (Part 2 of 3.)

```

83     connected = false; // update connected flag
84 } // end method disconnect
85
86 // send message to server
87 public void sendMessage( String from, String message )
88 {
89     if ( !connected )
90         return; // if not connected, return immediately
91
92     // create and start new MessageSender to deliver message
93     serverExecutor.execute(
94         new MessageSender( clientSocket, from, message ) );
95 } // end method sendMessage
96 } // end method SocketMessageManager

```

**Fig. 24.25** | SocketMessageManager implementation of interface MessageManager for communicating via Sockets and multicast DatagramPackets. (Part 3 of 3.)

receive incoming messages. Line 20 declares the Socket used to connect to and send messages to DeitelMessengerServer. Line 22 declares a PacketReceiver (Fig. 24.27) that listens for new incoming messages. The connected flag (line 23) indicates whether the SocketMessageManager is currently connected to DeitelMessengerServer.

The SocketMessageManager constructor (lines 26–30) receives the address of the DeitelMessengerServer to which SocketMessageManager should connect. Method connect (lines 33–52) connects SocketMessageManager to DeitelMessengerServer. If it was connected previously, line 36 returns from method connect. Lines 40–41 create a new Socket to communicate with the server. Line 41 creates an InetAddress object for the server's address and uses the constant SERVER\_PORT to specify the port on which the client should connect. Line 44 creates a new PacketReceiver, which listens for incoming multicast messages from the server, and line 45 executes the Runnable. Line 46 updates boolean variable connected to indicate that SocketMessageManager is connected to the server.

Method disconnect (lines 55–84) terminates the SocketMessageManager's connection to the server. If SocketMessageManager is not connected, line 58 returns from method disconnect. Lines 63–64 create a new MessageSender (Fig. 24.26) to send DISCONNECT\_STRING to DeitelMessengerServer. Class MessageSender delivers a message to DeitelMessengerServer over the SocketMessageManager's Socket connection. Line 65 starts the MessageSender to deliver the message using method submit of the ExecutorService. This method returns a Future which represents the executing Runnable. Line 66 invokes Future method get to wait for the disconnect message to be delivered and the Runnable to terminate. Once the disconnect message has been delivered, line 67 invokes PacketReceiver method stopListening to stop receiving incoming chat messages. Line 68 closes the Socket connection to DeitelMessengerServer.

Method sendMessage (lines 87–95) sends an outgoing message to the server. If SocketMessageManager is not connected, line 90 returns from method sendMessage. Lines 93–94 create and start a new MessageSender (Fig. 24.26) to deliver the new message in a separate thread of execution.

Class `MessageSender` (Fig. 24.26), which implements `Runnable`, delivers outgoing messages to the server in a separate thread of execution. `MessageSender`'s constructor (lines 16–22) takes as arguments the `Socket` over which to send the message, the `userName` from whom the message came and the message. Line 21 concatenates these arguments to build `messageToSend`. Constant `MESSAGE_SEPARATOR` enables the message recipient to parse the message into two parts—the sending user's name and the message body—by using a `StringTokenizer`.

Method `run` (lines 25–38) delivers the complete message to the server, using the `Socket` provided to the `MessageSender` constructor. Lines 29–30 create a new `Formatter` for the `clientSocket`'s `OutputStream`. Line 31 invokes `Formatter` method `format` to send

```

1 // Fig. 24.26: MessageSender.java
2 // Sends a message to the chat server in a separate Runnable.
3 package com.deitel.messenger.sockets.client;
4
5 import java.io.IOException;
6 import java.util.Formatter;
7 import java.net.Socket;
8
9 import static com.deitel.messenger.sockets.SocketMessengerConstants.*;
10
11 public class MessageSender implements Runnable
12 {
13     private Socket clientSocket; // Socket over which to send message
14     private String messageToSend; // message to send
15
16     public MessageSender( Socket socket, String userName, String message )
17     {
18         clientSocket = socket; // store Socket for client
19
20         // build message to be sent
21         messageToSend = userName + MESSAGE_SEPARATOR + message;
22     } // end MessageSender constructor
23
24     // send message and end
25     public void run()
26     {
27         try // send message and flush PrintWriter
28         {
29             Formatter output =
30                 new Formatter( clientSocket.getOutputStream() );
31             output.format( "%s\n", messageToSend ); // send message
32             output.flush(); // flush output
33         } // end try
34         catch ( IOException ioException )
35         {
36             ioException.printStackTrace();
37         } // end catch
38     } // end method run
39 } // end class MessageSender

```

**Fig. 24.26** | `MessageSender` for delivering outgoing messages to `DeitelMessengerServer`.

the message. Line 32 invokes method `flush` of class `Formatter` to ensure that the message is sent immediately. Note that class `MessageSender` does not close the `clientSocket`. Class `SocketMessageManager` uses a new object of class `MessageSender` for each message the client sends, so the `clientSocket` must remain open until the user disconnects from `DeitelMessengerServer`.

Class `PacketReceiver` (Fig. 24.27) implements interface `Runnable` to enable `SocketMessageManager` to listen for incoming messages in a separate thread of execution. Line 18 declares the `MessageListener` to which `PacketReceiver` will deliver incoming messages. Line 19 declares a `MulticastSocket` for receiving multicast `DatagramPackets`. Line 20 declares an `InetAddress` reference for the multicast address to which `DeitelMessengerServer` posts new chat messages. The `MulticastSocket` connects to this `InetAddress` to listen for incoming chat messages.

The `PacketReceiver` constructor (lines 23–46) takes as an argument the `MessageListener` to which the `PacketReceiver` is to deliver incoming messages. Recall that interface `MessageListener` declares method `messageReceived`. When the `PacketReceiver`

```

1 // Fig. 24.27: PacketReceiver.java
2 // PacketReceiver listens for DatagramPackets containing
3 // messages from a DeitelMessengerServer.
4 package com.deitel.messenger.sockets.client;
5
6 import java.io.IOException;
7 import java.net.InetAddress;
8 import java.net.MulticastSocket;
9 import java.net.DatagramPacket;
10 import java.net.SocketTimeoutException;
11 import java.util.StringTokenizer;
12
13 import com.deitel.messenger.MessageListener;
14 import static com.deitel.messenger.sockets.SocketMessengerConstants.*;
15
16 public class PacketReceiver implements Runnable
17 {
18     private MessageListener messageListener; // receives messages
19     private MulticastSocket multicastSocket; // receive broadcast messages
20     private InetAddress multicastGroup; // InetAddress of multicast group
21     private boolean keepListening = true; // terminates PacketReceiver
22
23     public PacketReceiver( MessageListener listener )
24     {
25         messageListener = listener; // set MessageListener
26
27         try // connect MulticastSocket to multicast address and port
28         {
29             // create new MulticastSocket
30             multicastSocket = new MulticastSocket(
31                 MULTICAST_LISTENING_PORT );
32

```

**Fig. 24.27** | `PacketReceiver` for listening for new multicast messages from `DeitelMessengerServer` in a separate thread. (Part 1 of 3.)

```

33 // use InetAddress to get multicast group
34 multicastGroup = InetAddress.getByName( MULTICAST_ADDRESS );
35
36 // join multicast group to receive messages
37 multicastSocket.joinGroup( multicastGroup );
38
39 // set 5 second timeout when waiting for new packets
40 multicastSocket.setSoTimeout( 5000 );
41 } // end try
42 catch ( IOException ioException )
43 {
44     ioException.printStackTrace();
45 } // end catch
46 } // end PacketReceiver constructor
47
48 // listen for messages from multicast group
49 public void run()
50 {
51     // listen for messages until stopped
52     while ( keepListening )
53     {
54         // create buffer for incoming message
55         byte[] buffer = new byte[ MESSAGE_SIZE ];
56
57         // create DatagramPacket for incoming message
58         DatagramPacket packet = new DatagramPacket( buffer,
59             MESSAGE_SIZE );
60
61         try // receive new DatagramPacket (blocking call)
62         {
63             multicastSocket.receive( packet );
64         } // end try
65         catch ( SocketTimeoutException socketTimeoutException )
66         {
67             continue; // continue to next iteration to keep listening
68         } // end catch
69         catch ( IOException ioException )
70         {
71             ioException.printStackTrace();
72             break;
73         } // end catch
74
75         // put message data in a String
76         String message = new String( packet.getData() );
77
78         message = message.trim(); // trim whitespace from message
79
80         // tokenize message to retrieve user name and message body
81         StringTokenizer tokenizer = new StringTokenizer(
82             message, MESSAGE_SEPARATOR );
83

```

**Fig. 24.27** | PacketReceiver for listening for new multicast messages from DeitelMessengerServer in a separate thread. (Part 2 of 3.)



```

84     // ignore messages that do not contain a user
85     // name and message body
86     if ( tokenizer.countTokens() == 2 )
87     {
88         // send message to MessageListener
89         messageListener.messageReceived(
90             tokenizer.nextToken(), // user name
91             tokenizer.nextToken() ); // message body
92     } // end if
93 } // end while
94
95 try
96 {
97     multicastSocket.leaveGroup( multicastGroup ); // leave group
98     multicastSocket.close(); // close MulticastSocket
99 } // end try
100 catch ( IOException ioException )
101 {
102     ioException.printStackTrace();
103 } // end catch
104 } // end method run
105
106 // stop listening for new messages
107 public void stopListening()
108 {
109     keepListening = false;
110 } // end method stopListening
111 } // end class PacketReceiver

```

**Fig. 24.27** | PacketReceiver for listening for new multicast messages from DeitelMessengerServer in a separate thread. (Part 3 of 3.)

receives a new chat message over the MulticastSocket, PacketReceiver invokes messageReceived to deliver the new message to the MessageListener.

Lines 30–31 create a new MulticastSocket and pass to the MulticastSocket constructor the constant MULTICAST\_LISTENING\_PORT from interface SocketMessengerConstants. This argument specifies the port on which the MulticastSocket will listen for incoming chat messages. Line 34 creates an InetAddress object for the MULTICAST\_ADDRESS, to which DeitelMessengerServer multicasts new chat messages. Line 37 invokes MulticastSocket method **joinGroup** to register the MulticastSocket to receive messages sent to MULTICAST\_ADDRESS. Line 40 invokes MulticastSocket method **setSoTimeout** to specify that if no data is received in 5000 milliseconds, the MulticastSocket should issue an InterruptedException, which the current thread can catch, then continue executing. This approach prevents PacketReceiver from blocking indefinitely when waiting for incoming data. Also, if the MulticastSocket never timed out, the while loop would not be able to check the keepListening variable and would therefore prevent PacketReceiver from stopping if keepListening were set to false.

Method run (lines 49–104) listens for incoming multicast messages. Lines 58–59 create a DatagramPacket to store the incoming message. Line 63 invokes MulticastSocket method receive to read an incoming packet from the multicast address. If 5000

milliseconds pass without receipt of a packet, method `receive` throws an `InterruptedException`, because we previously set a 5000-millisecond timeout (line 40). Line 67 uses `continue` to proceed to the next loop iteration to listen for incoming messages. For other `IOExceptions`, line 72 breaks the `while` loop to terminate the `PacketReceiver`.

Line 76 invokes `DatagramPacket` method `getData` to retrieve the message data. Line 78 invokes method `trim` of class `String` to remove extra white space from the end of the message. Recall that `DatagramPackets` are of a fixed size—512 bytes in this example—so, if the message is shorter than 512 bytes, there will be extra white space after it. Lines 81–82 create a `StringTokenizer` to separate the message body from the name of the user who sent the message. Line 86 checks for the correct number of tokens. Lines 89–91 invoke method `messageReceived` of interface `MessageListener` to deliver the incoming message to the `PacketReceiver`'s `MessageListener`.

If the program invokes method `stopListening` (lines 107–110), the `while` loop in method `run` (lines 49–104) terminates. Line 97 invokes `MulticastSocket` method `leaveGroup` to stop receiving messages from the multicast address. Line 98 invokes `MulticastSocket` method `close` to close the `MulticastSocket`. When method `run` completes execution, the `PacketReceiver` terminates.

Class `ClientGUI` (Fig. 24.28) extends `JFrame` to create a GUI for a user to send and receive chat messages. The GUI consists of a `JTextArea` for displaying incoming messages

```

1 // Fig. 24.28: ClientGUI.java
2 // ClientGUI provides a user interface for sending and receiving
3 // messages to and from the DeitelMessengerServer.
4 package com.deitel.messenger;
5
6 import java.awt.BorderLayout;
7 import java.awt.event.ActionEvent;
8 import java.awt.event.ActionListener;
9 import java.awt.event.WindowAdapter;
10 import java.awt.event.WindowEvent;
11 import javax.swing.Box;
12 import javax.swing.BoxLayout;
13 import javax.swing.Icon;
14 import javax.swing.ImageIcon;
15 import javax.swing.JButton;
16 import javax.swing.JFrame;
17 import javax.swing.JLabel;
18 import javax.swing.JMenuBar;
19 import javax.swing.JMenu;
20 import javax.swing.JMenuItem;
21 import javax.swing.JOptionPane;
22 import javax.swing.JPanel;
23 import javax.swing.JScrollPane;
24 import javax.swing.JTextArea;
25 import javax.swing.SwingUtilities;
26 import javax.swing.border.BevelBorder;
27

```

**Fig. 24.28** | `ClientGUI` subclass of `JFrame` for presenting a GUI for viewing and sending chat messages. (Part 1 of 6.)

```

28 public class ClientGUI extends JFrame
29 {
30     private JMenu serverMenu; // for connecting/disconnecting server
31     private JTextArea messageArea; // displays messages
32     private JTextArea inputArea; // inputs messages
33     private JButton connectButton; // button for connecting
34     private JMenuItem connectMenuItem; // menu item for connecting
35     private JButton disconnectButton; // button for disconnecting
36     private JMenuItem disconnectMenuItem; // menu item for disconnecting
37     private JButton sendButton; // sends messages
38     private JLabel statusBar; // label for connection status
39     private String userName; // userName to add to outgoing messages
40     private MessageManager messageManager; // communicates with server
41     private MessageListener messageListener; // receives incoming messages
42
43     // ClientGUI constructor
44     public ClientGUI( MessageManager manager )
45     {
46         super( "Deitel Messenger" );
47
48         messageManager = manager; // set the MessageManager
49
50         // create MyMessageListener for receiving messages
51         messageListener = new MyMessageListener();
52
53         serverMenu = new JMenu ( "Server" ); // create Server JMenu
54         serverMenu.setMnemonic( 'S' ); // set mnemonic for server menu
55         JMenuBar menuBar = new JMenuBar(); // create JMenuBar
56         menuBar.add( serverMenu ); // add server menu to menu bar
57         setJMenuBar( menuBar ); // add JMenuBar to application
58
59         // create ImageIcon for connect buttons
60         Icon connectIcon = new ImageIcon(
61             Tus().getResource( "images/Connect.gif" ) );
62
63         // create connectButton and connectMenuItem
64         connectButton = new JButton( "Connect", connectIcon );
65         connectMenuItem = new JMenuItem( "Connect", connectIcon );
66         connectMenuItem.setMnemonic( 'C' );
67
68         // create ConnectListener for connect buttons
69         ActionListener connectListener = new ConnectListener();
70         connectButton.addActionListener( connectListener );
71         connectMenuItem.addActionListener( connectListener );
72
73         // create ImageIcon for disconnect buttons
74         Icon disconnectIcon = new ImageIcon(
75             getClass().getResource( "images/Disconnect.gif" ) );
76
77         // create disconnectButton and disconnectMenuItem
78         disconnectButton = new JButton( "Disconnect", disconnectIcon );

```

**Fig. 24.28** | ClientGUI subclass of JFrame for presenting a GUI for viewing and sending chat messages. (Part 2 of 6.)

```

79     disconnectMenuItem = new JMenuItem( "Disconnect", disconnectIcon );
80     disconnectMenuItem.setMnemonic( 'D' );
81
82     // disable disconnect button and menu item
83     disconnectButton.setEnabled( false );
84     disconnectMenuItem.setEnabled( false );
85
86     // create DisconnectListener for disconnect buttons
87     ActionListener disconnectListener = new DisconnectListener();
88     disconnectButton.addActionListener( disconnectListener );
89     disconnectMenuItem.addActionListener( disconnectListener );
90
91     // add connect and disconnect JMenuItem to fileMenu
92     serverMenu.add( connectMenuItem );
93     serverMenu.add( disconnectMenuItem );
94
95     // add connect and disconnect JButtons to buttonPanel
96     JPanel buttonPanel = new JPanel();
97     buttonPanel.add( connectButton );
98     buttonPanel.add( disconnectButton );
99
100    messageArea = new JTextArea(); // displays messages
101    messageArea.setEditable( false ); // disable editing
102    messageArea.setWrapStyleWord( true ); // set wrap style to word
103    messageArea.setLineWrap( true ); // enable line wrapping
104
105    // put messageArea in JScrollPane to enable scrolling
106    JPanel messagePanel = new JPanel();
107    messagePanel.setLayout( new BorderLayout( 10, 10 ) );
108    messagePanel.add( new JScrollPane( messageArea ),
109                    BorderLayout.CENTER );
110
111    inputArea = new JTextArea( 4, 20 ); // for entering new messages
112    inputArea.setWrapStyleWord( true ); // set wrap style to word
113    inputArea.setLineWrap( true ); // enable line wrapping
114    inputArea.setEditable( false ); // disable editing
115
116    // create Icon for sendButton
117    Icon sendIcon = new ImageIcon(
118        getClass().getResource( "images/Send.gif" ) );
119
120    sendButton = new JButton( "Send", sendIcon ); // create send button
121    sendButton.setEnabled( false ); // disable send button
122    sendButton.addActionListener(
123        new ActionListener()
124        {
125            // send new message when user activates sendButton
126            public void actionPerformed((ActionEvent event) )
127            {
128                messageManager.sendMessage( userName,
129                    inputArea.getText() ); // send message

```

**Fig. 24.28** | CClientGUI subclass of JFrame for presenting a GUI for viewing and sending chat messages. (Part 3 of 6.)

```

130         inputArea.setText( "" ); // clear inputArea
131     } // end method actionPerformed
132 } // end anonymous inner class
133 ); // end call to addActionListener
134
135 Box box = new Box( BorderLayout.X_AXIS ); // create new box for layout
136 box.add( new JScrollPane( inputArea ) ); // add input area to box
137 box.add( sendButton ); // add send button to box
138 messagePanel.add( box, BorderLayout.SOUTH ); // add box to panel
139
140 // create JLabel for statusBar with a recessed border
141 statusBar = new JLabel( "Not Connected" );
142 statusBar.setBorder( new BevelBorder( BevelBorder.LOWERED ) );
143
144 add( buttonPanel, BorderLayout.NORTH ); // add button panel
145 add( messagePanel, BorderLayout.CENTER ); // add message panel
146 add( statusBar, BorderLayout.SOUTH ); // add status bar
147
148 // add WindowListener to disconnect when user quits
149 addWindowListener (
150     new WindowAdapter ()
151     {
152         // disconnect from server and exit application
153         public void windowClosing ( WindowEvent event )
154         {
155             messageManager.disconnect( messageListener );
156             System.exit( 0 );
157         } // end method windowClosing
158     } // end anonymous inner class
159 ); // end call to addWindowListener
160 } // end ClientGUI constructor
161
162 // ConnectListener listens for user requests to connect to server
163 private class ConnectListener implements ActionListener
164 {
165     // connect to server and enable/disable GUI components
166     public void actionPerformed( ActionEvent event )
167     {
168         // connect to server and route messages to messageListener
169         messageManager.connect( messageListener );
170
171         // prompt for userName
172         userName = JOptionPane.showInputDialog(
173             ClientGUI.this, "Enter user name:" );
174
175         messageArea.setText( "" ); // clear messageArea
176         connectButton.setEnabled( false ); // disable connect
177         connectMenuItem.setEnabled( false ); // disable connect
178         disconnectButton.setEnabled( true ); // enable disconnect
179         disconnectMenuItem.setEnabled( true ); // enable disconnect
180         sendButton.setEnabled( true ); // enable send button

```

**Fig. 24.28** | ClientGUI subclass of JFrame for presenting a GUI for viewing and sending chat messages. (Part 4 of 6.)

```

181     inputArea.setEditable( true ); // enable editing for input area
182     inputArea.requestFocus(); // set focus to input area
183     statusBar.setText( "Connected: " + userName ); // set text
184 } // end method actionPerformed
185 } // end ConnectListener inner class
186
187 // DisconnectListener listens for user requests to disconnect
188 // from DeitelMessengerServer
189 private class DisconnectListener implements ActionListener
190 {
191     // disconnect from server and enable/disable GUI components
192     public void actionPerformed( ActionEvent event )
193     {
194         // disconnect from server and stop routing messages
195         messageManager.disconnect( messageListener );
196         sendButton.setEnabled( false ); // disable send button
197         disconnectButton.setEnabled( false ); // disable disconnect
198         disconnectMenuItem.setEnabled( false ); // disable disconnect
199         inputArea.setEditable( false ); // disable editing
200         connectButton.setEnabled( true ); // enable connect
201         connectMenuItem.setEnabled( true ); // enable connect
202         statusBar.setText( "Not Connected" ); // set status bar text
203     } // end method actionPerformed
204 } // end DisconnectListener inner class
205
206 // MyMessageListener listens for new messages from MessageManager and
207 // displays messages in messageArea using MessageDisplayer.
208 private class MyMessageListener implements MessageListener
209 {
210     // when received, display new messages in messageArea
211     public void messageReceived( String from, String message )
212     {
213         // append message using MessageDisplayer
214         SwingUtilities.invokeLater(
215             new MessageDisplayer( from, message ) );
216     } // end method messageReceived
217 } // end MyMessageListener inner class
218
219 // Displays new message by appending message to JTextArea. Should
220 // be executed only in Event thread; modifies live Swing component
221 private class MessageDisplayer implements Runnable
222 {
223     private String fromUser; // user from which message came
224     private String messageBody; // body of message
225
226     // MessageDisplayer constructor
227     public MessageDisplayer( String from, String body )
228     {
229         fromUser = from; // store originating user
230         messageBody = body; // store message body
231     } // end MessageDisplayer constructor

```

**Fig. 24.28** | ClientGUI subclass of JFrame for presenting a GUI for viewing and sending chat messages. (Part 5 of 6.)

```

232
233     // display new message in messageArea
234     public void run()
235     {
236         // append new message
237         messageArea.append( "\n" + fromUser + "> " + messageBody );
238     } // end method run
239 } // end MessageDisplayer inner class
240 } // end class ClientGUI

```

**Fig. 24.28** | ClientGUI subclass of JFrame for presenting a GUI for viewing and sending chat messages. (Part 6 of 6.)

(line 31), a JTextArea for entering new messages (line 32), JButtons and JMenuItem for connecting to and disconnecting from the server (lines 33–36) and a JButton for sending messages (line 37). The GUI also contains a JLabel that displays whether the client is connected or disconnected (line 38).

ClientGUI uses a MessageManager (line 40) to handle all communication with the chat server. Recall that MessageManager is an interface that enables ClientGUI to use any MessageManager implementation. Class ClientGUI also uses a MessageListener (line 41) to receive incoming messages from the MessageManager.

The ClientGUI constructor (lines 44–160) takes as an argument the MessageManager for communicating with DeitelMessengerServer. Line 48 sets the ClientGUI's MessageManager. Line 51 creates an instance of MyMessageListener, which implements interface MessageListener. Lines 53–57 create a **Server** menu that contains JMenuItem for connecting to and disconnecting from the chat server. Lines 60–61 create an ImageIcon for connectButton and connectMenuItem.

Lines 64–65 create connectButton and connectMenuItem, each with the label "Connect" and the Icon connectIcon. Line 66 invokes method setMnemonic to set the mnemonic character for keyboard access to connectMenuItem. Line 69 creates an instance of inner class ConnectListener (declared at lines 163–185), which implements interface ActionListener to handle ActionEvents from connectButton and connectMenuItem. Lines 70–71 add connectListener as an ActionListener for connectButton and connectMenuItem.

Lines 74–75 create an ImageIcon for the disconnectButton and disconnectMenuItem components. Lines 78–79 create disconnectButton and disconnectMenuItem, each with the label "Disconnect" and the Icon disconnectIcon. Line 80 invokes method setMnemonic to enable keyboard access to disconnectMenuItem. Lines 83–84 invoke method setEnabled with a false argument on disconnectButton and disconnectMenuItem to disable these components. This prevents the user from attempting to disconnect from the server because the client is not yet connected. Line 87 creates an instance of inner class DisconnectListener (declared at lines 189–204), which implements interface ActionListener to handle ActionEvents from disconnectButton and disconnectMenuItem. Lines 88–89 add disconnectListener as an ActionListener for disconnectButton and disconnectMenuItem.

Lines 92–93 add connectMenuItem and disconnectMenuItem to menu **Server**. Lines 96–98 create a JPanel and add connectButton and disconnectButton to it. Line 100 cre-

ates the text area `messageArea`, in which the client displays incoming messages. Line 101 invokes method `setEditable` with a `false` argument, to disable editing. Lines 102–103 invoke `JTextArea` methods `setWrapStyleWord` and `setLineWrap` to enable word wrapping in `messageArea`. If a message is longer than `messageArea`'s width, the `messageArea` will wrap the text after the last word that fits on each line, making longer messages easier to read. Lines 106–109 create a `JPanel` for the `messageArea` and add the `messageArea` to the `JPanel` in a `JScrollPane`.

Line 111 creates the `inputArea` `JTextArea` for entering new messages. Lines 112–113 enable word and line wrapping, and line 114 disables editing the `inputArea`. When the client connects to the chat server, `ConnectListener` enables the `inputArea` to allow the user to type new messages.

Lines 117–118 create an `ImageIcon` for `sendButton`. Line 120 creates `sendButton`, which the user can click to send a message. Line 121 disables `sendButton`—the `ConnectListener` enables the `sendButton` when the client connects to the chat server. Lines 122–133 add an `ActionListener` to `sendButton`. Lines 128–129 invoke method `sendMessage` of interface `MessageManager` with the `userName` and `inputArea` text as arguments. This statement sends the user's name and message as a new chat message to `DeitelMessengerServer`. Line 130 clears the `inputArea` for the next message.

Lines 135–138 use a horizontal `Box` container to arrange components `inputArea` and `sendButton`. Line 136 places `inputArea` in a `JScrollPane` to enable scrolling of long messages. Line 138 adds the `Box` containing `inputArea` and `sendButton` to the `SOUTH` region of `messagePanel`. Line 141 creates the `statusBar` `JLabel`. This label displays whether the client is connected to or disconnected from the chat server. Line 142 invokes method `setBorder` of class `JLabel` and creates a new `BevelBorder` of type `BevelBorder.LOWERED`. This `border` makes the label appear recessed, as is common with status bars in many applications. Lines 144–146 add `buttonPanel`, `messagePanel` and `statusBar` to the `ClientGUI`.

Lines 149–159 add a `WindowListener` to the `ClientGUI`. Line 155 invokes method `disconnect` of interface `MessageManager` to disconnect from the chat server in case the user quits while still connected. Then line 156 terminates the application.

Inner class `ConnectListener` (lines 163–185) handles events from `connectButton` and `connectMenuItem`. Line 169 invokes `MessageManager` method `connect` to connect to the chat server. Line 169 passes as an argument to method `connect` the `MessageListener` to which new messages should be delivered. Lines 172–173 prompt the user for a user name, and line 175 clears the `messageArea`. Lines 176–181 enable the components for disconnecting from the server and for sending messages and disable the components for connecting to the server. Line 182 invokes `inputArea`'s `requestFocus` method (inherited from class `Component`) to place the text-input cursor in the `inputArea` so that the user can immediately begin typing a message.

Inner class `DisconnectListener` (lines 189–204) handles events from `disconnectButton` and `disconnectMenuItem`. Line 195 invokes `MessageManager` method `disconnect` to disconnect from the chat server. Lines 196–201 disable the components for sending messages and the components for disconnecting, then enable the components for connecting to the chat server.

Inner class `MyMessageListener` (lines 208–217) implements interface `MessageListener` to receive incoming messages from the `MessageManager`. When a new message is



received, the `MessageManager` invokes method `messageReceived` (lines 211–216) with the user name of the sender and the message body. Lines 214–215 invoke `SwingUtilities` method `invokeLater` with a `MessageDisplayer` object that appends the new message to `messageArea`. Recall, from Chapter 23, that Swing components should be accessed only from the event dispatch thread. Method `messageReceived` is invoked by the `PacketReceiver` in class `SocketMessageManager` and therefore cannot append the message text to `messageArea` directly, as this would occur in `PacketReceiver`, not the event dispatch thread.

Inner class `MessageDisplayer` (lines 221–239) implements interface `Runnable` to provide a thread-safe way to append text to the `messageArea`. The `MessageDisplayer` constructor (lines 227–231) takes as arguments the user name and the message to send. Method `run` (lines 234–238) appends the user name, "> " and `messageBody` to `messageArea`.

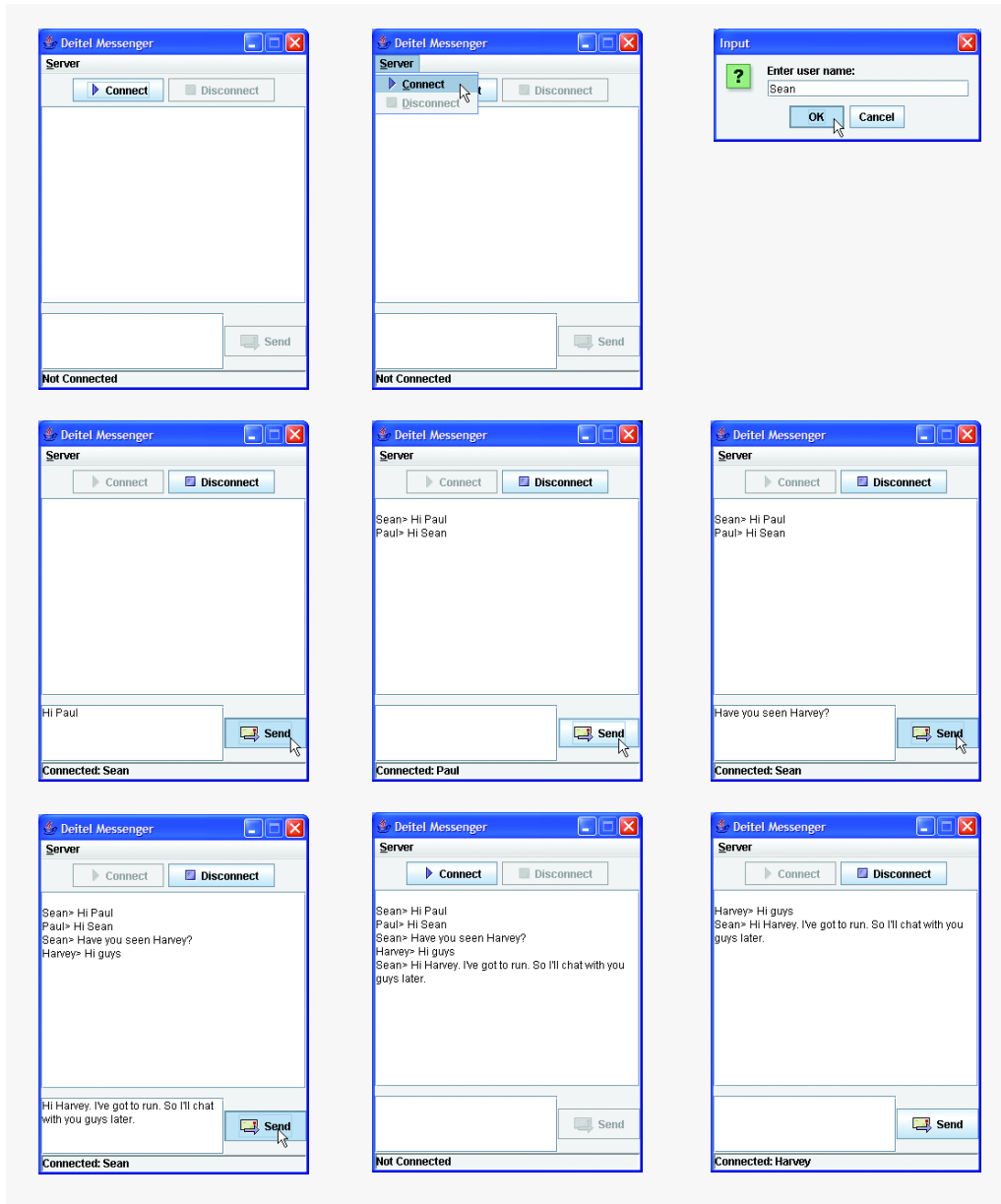
Class `DeitelMessenger` (Fig. 24.29) launches the client for the `DeitelMessengerServer`. Lines 15–20 create a new `SocketMessageManager` to connect to the `DeitelMessengerServer` with the IP address specified as a command-line argument to the application (or `localhost`, if no address is provided). Lines 23–26 create a `ClientGUI` for the `MessageManager`, set the `ClientGUI` size and make the `ClientGUI` visible.

```

1 // Fig. 24.29: DeitelMessenger.java
2 // DeitelMessenger is a chat application that uses a ClientGUI
3 // and SocketMessageManager to communicate with DeitelMessengerServer.
4 package com.deitel.messenger.sockets.client;
5
6 import com.deitel.messenger.MessageManager;
7 import com.deitel.messenger.ClientGUI;
8
9 public class DeitelMessenger
10 {
11     public static void main( String args[] )
12     {
13         MessageManager messageManager; // declare MessageManager
14
15         if ( args.length == 0 )
16             // connect to localhost
17             messageManager = new SocketMessageManager( "localhost" );
18         else
19             // connect using command-line arg
20             messageManager = new SocketMessageManager( args[ 0 ] );
21
22         // create GUI for SocketMessageManager
23         ClientGUI clientGUI = new ClientGUI( messageManager );
24         clientGUI.setSize( 300, 400 ); // set window size
25         clientGUI.setResizable( false ); // disable resizing
26         clientGUI.setVisible( true ); // show window
27     } // end main
28 } // end class DeitelMessenger

```

**Fig. 24.29** | `DeitelMessenger` application for participating in a `DeitelMessengerServer` chat session. (Part 1 of 2.)



**Fig. 24.29** | DeitelMessenger application for participating in a DeitelMessengerServer chat session. (Part 2 of 2.)

*Executing the DeitelMessenger Client Application*

To execute the DeitelMessenger client, open a command window and change directories to the location in which package `com.deitel.messenger.sockets.client` resides (i.e., the directory in which `com` is located). Then type

```
java com.deitel.messenger.sockets.client.DeitelMessenger
```

to execute the client and connect to the `DeitelMessengerServer` running on your local computer. If the server resides on another computer, follow the preceding command with the hostname or IP address of that computer. The preceding command is equivalent to

```
java com.deitel.messenger.sockets.client.DeitelMessenger localhost
```

or

```
java com.deitel.messenger.sockets.client.DeitelMessenger 127.0.0.1
```

### *Deitel Messenger Case Study Summary*

The Deitel messenger case study is a significant application that uses many intermediate Java features, such as networking with `Sockets`, `DatagramPackets` and `MulticastSockets`, multithreading and Swing GUI. The case study also demonstrates good software engineering practices by separating interface from implementation, enabling developers to build `MessageManagers` for different network protocols and `MessageListeners` that provide different user interfaces. You should now be able to apply these techniques to your own, more complex, Java projects.