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.ExecutorService;
10 import java.util.concurrent.Executors;
11 import com.deitel.messenger.MessageListener;
12 import static com.deitel.messenger.sockets.SocketMessengerConstants.*;
```

Fig. 24.18 | DeitelMessengerServer for managing a chat room. (Part 1 of 2.)
public class DeitelMessengerServer implements MessageListener {
  private ExecutorService serverExecutor; // executor for server

  // start chat server
  public void startServer() {
    try // create executor for server runnables
        serverExecutor = Executors.newCachedThreadPool();
    
    // create ServerSocket for incoming connections
    ServerSocket serverSocket =
        new ServerSocket( SERVER_PORT, 100 );

    System.out.printf( "%s%d%s", "Server listening on port ",
        SERVER_PORT, " ..." );

    // listen for clients constantly
    while ( true )
      {
        // accept new client connection
        Socket clientSocket = serverSocket.accept();

        // create MessageReceiver for receiving messages from client
        serverExecutor.execute( 
            new MessageReceiver( this, clientSocket ) );

        // print connection information
        System.out.println( "Connection received from: " +
            clientSocket.getInetAddress() );
      } // end while
  } // end method startServer

  // when new message is received, broadcast message to clients
  public void messageReceived( String from, String message ) {
    try
      {

      } // end try
  } // end method messageReceived

Fig. 24.18 | DeitelMessengerServer for managing a chat room. (Part 2 of 2.)
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.mes-
senger.sockets.server uses multicasting as an efficient mechanism for sending one mes-
sage 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.

---

```
// Fig. 24.19: DeitelMessengerServerTest.java
// Test the DeitelMessengerServer class.
package com.deitel.messenger.sockets.server;

public class DeitelMessengerServerTest
{
    public static void main ( String args[] )
    {
        DeitelMessengerServer application = new DeitelMessengerServer();
        application.startServer(); // start server
    } // end main
} // end class DeitelMessengerServerTest
```

Fig. 24.19 | Test class for DeitelMessengerServer.

```
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. 24.20: SocketMessengerConstants.java
// SocketMessengerConstants defines constants for the port numbers
// and multicast address in DeitelMessenger
package com.deitel.messenger.sockets;

public interface SocketMessengerConstants
{
    // address for multicast datagrams
    public static final String MULTICAST_ADDRESS = "239.0.0.1";

    // port for listening for multicast datagrams
    public static final int MULTICAST_LISTENING_PORT = 5555;

    // port for sending multicast datagrams
    public static final int MULTICAST_SENDING_PORT = 5554;

    // port for Socket connections to DeitelMessengerServer
    public static final int SERVER_PORT = 12345;
```

Fig. 24.20 | SocketMessengerConstants declares constants for use in the DeitelMessengerServer and DeitelMessenger. (Part 1 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
// Fig. 24.22: MessageReceiver.java
// MessageReceiver is a Runnable that listens for messages from a
// particular client and delivers messages to a MessageListener.
package com.deitel.messenger.sockets.server;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.Socket;
import java.net.SocketTimeoutException;
import java.util.StringTokenizer;
import com.deitel.messenger.MessageListener;
import static com.deitel.messenger.sockets.SocketMessengerConstants.*;

public class MessageReceiver implements Runnable
{
    private BufferedReader input; // input stream
    private MessageListener messageListener; // message listener
    private boolean keepListening = true; // when false, ends runnable

    // MessageReceiver constructor
    public MessageReceiver( MessageListener listener, Socket clientSocket )
    {
        // set listener to which new messages should be sent
        messageListener = listener;
        try
        {
            // set timeout for reading from client
            clientSocket.setSoTimeout( 5000 ); // five seconds
            // create BufferedReader for reading incoming messages
            input = new BufferedReader( new InputStreamReader( clientSocket.getInputStream() ) );
        }
        catch ( IOException ioException )
        {
            ioException.printStackTrace();
        }
    }

    // listen for new messages and deliver them to MessageListener
    public void run()
    {
        String message; // String for incoming messages
        // listen for messages until stopped
        while ( keepListening )
        {

        }
    }

    Fig. 24.22  |  MessageReceiver for listening for new messages from DeitelMessengerServer clients in separate threads. (Part 1 of 3.)
```java
try {
    message = input.readLine(); // read message from client
} // end try
catch ( SocketTimeoutException socketTimeoutException )
{
    continue; // continue to next iteration to keep listening
} // end catch
}
// ensure non-null message
if ( message != null )
{
    // ignore messages that do not contain a user
    // name and message body
    if ( tokenizer.countTokens() == 2 )
    {
        // send message to MessageListener
        messageListener.messageReceived( 
            tokenizer.nextToken(), // user name
            tokenizer.nextToken() ); // message body
    } // end if
    else
    {
        // if disconnect message received, stop listening
        if ( message.equalsIgnoreCase( MESSAGE_SEPARATOR + DISCONNECT_STRING ) )
            stopListening();
    } // end else
} // end while

try {
    input.close(); // close BufferedReader (also closes Socket)
} // end try
catch ( IOException ioException )
{
    ioException.printStackTrace();
} // end catch
} // end method run
```

**Fig. 24.22** MessageReceiver for listening for new messages from DeitelMessengerServer clients in separate threads. (Part 2 of 3.)
Case Study: DeitelMessenger Server and Client

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

---

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

```java
// deliver message to MULTICAST_ADDRESS over DatagramSocket
public void run()
{
  try // deliver message
  {
    // create DatagramSocket for sending message
    DatagramSocket socket = new DatagramSocket( MULTICAST_SENDING_PORT );
    // use InetAddress reserved for multicast group
    InetAddress group = InetAddress.getByName( MULTICAST_ADDRESS );
    // create DatagramPacket containing message
    DatagramPacket packet = new DatagramPacket( messageBytes, messageBytes.length, group, MULTICAST_LISTENING_PORT );
    socket.send( packet ); // send packet to multicast group
    socket.close(); // close socket
  } // end try
  catch ( IOException ioException )
  {
    ioException.printStackTrace();
  } // end catch
} // end method run
// end class MulticastSender
```

**Fig. 24.23** | MulticastSender for delivering outgoing messages to a multicast group via DatagramPackets. (Part 2 of 2.)
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.messengersockets.server resides (i.e., the directory in which com is located). Then type

```
java com.deitel.messengersockets.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

```java
1 // Fig. 24.24: MessageManager.java
2 // MessageManager 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

```java
// Fig. 24.25: SocketMessageManager.java
// SocketMessageManager communicates with a DeitelMessengerServer using
// Sockets and MulticastSockets.
package com.deitel.messenger.sockets.client;

import java.net.InetAddress;
import java.net.Socket;
import java.io.IOException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;

import com.deitel.messenger.MessageListener;
import com.deitel.messenger.MessageManager;
import static com.deitel.messenger.sockets.SocketMessengerConstants.*;

public class SocketMessageManager implements MessageManager
{
    private Socket clientSocket; // Socket for outgoing messages
    private String serverAddress; // DeitelMessengerServer address
    private PacketReceiver receiver; // receives multicast messages
    private boolean connected = false; // connection status
    private ExecutorService serverExecutor; // executor for server

    public SocketMessageManager( String address )
    {
        serverAddress = address; // store server address
        serverExecutor = Executors.newCachedThreadPool();
    } // end SocketMessageManager constructor
```

Fig. 24.25 | SocketMessageManager implementation of interface MessageManager for communicating via Sockets and multicast DatagramPackets. (Part 1 of 3.)
public void connect( MessageListener listener )
{
    if ( connected )
        return; // if already connected, return immediately

    try // open Socket connection to DeitelMessengerServer
    {
        clientSocket = new Socket( InetAddress.getByName( serverAddress ), SERVER_PORT );

        // create Runnable for receiving incoming messages
        receiver = new PacketReceiver( listener );
        serverExecutor.execute( receiver ); // execute Runnable
        connected = true; // update connected flag
    } // end try
    catch ( IOException ioException )
    {
        ioException.printStackTrace();
    } // end catch
} // end method connect

// disconnect from server and unregister given MessageListener
public void disconnect( MessageListener listener )
{
    try // stop listener and disconnect from server
    {
        Runnable disconnecter = new MessageSender( clientSocket, "", DISCONNECT_STRING );
        Future disconnecting = serverExecutor.submit( disconnecter );
        disconnecting.get(); // wait for disconnect message to be sent
        receiver.stopListening(); // stop receiver
        clientSocket.close(); // close outgoing Socket
    } // end try
    catch ( ExecutionException exception )
    {
        exception.printStackTrace();
    } // end catch
    catch ( InterruptedException exception )
    {
        exception.printStackTrace();
    } // end catch
    catch ( IOException ioException )
    {
        ioException.printStackTrace();
    } // end catch
}

Fig. 24.25 | SocketMessageManager implementation of interface MessageManager for communicating via Sockets and multicast DatagramPackets. (Part 2 of 3.)
24.10 Case Study: DeitelMessenger Server and Client

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

```java
// Fig. 24.26: MessageSender.java
// Sends a message to the chat server in a separate Runnable.
package com.deitel.messenger.sockets.client;
import java.io.IOException;
import java.util.Formatter;
import java.net.Socket;
import static com.deitel.messenger.sockets.SocketMessengerConstants.*;

public class MessageSender implements Runnable {
    private Socket clientSocket; // Socket over which to send message
    private String messageToSend; // message to send

    public MessageSender( Socket socket, String userName, String message )
    {
        clientSocket = socket; // store Socket for client
        // build message to be sent
        messageToSend = userName + MESSAGE_SEPARATOR + message;
    }

    // send message and end
    public void run()
    {
        try // send message and flush PrintWriter
        {
            Formatter output =
                new Formatter( clientSocket.getOutputStream() );
            output.format( "%s\n", messageToSend ); // send message
            output.flush(); // flush output
        } // end try
        catch ( IOException ioException )
        {
            ioException.printStackTrace();
        } // end catch
    }
}
```

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

© Copyright 1997–2007 by Pearson Education, Inc. All Rights Reserved.
the message. Line 32 invokes method \texttt{flush} of class \texttt{Formatter} to ensure that the message is sent immediately. Note that class \texttt{MessageSender} does not close the \texttt{clientSocket}. Class \texttt{SocketMessageManager} uses a new object of class \texttt{MessageSender} for each message the client sends, so the \texttt{clientSocket} must remain open until the user disconnects from \texttt{DeitelMessengerServer}.

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

The \texttt{PacketReceiver} constructor (lines 23–46) takes as an argument the \texttt{MessageListener} to which the \texttt{PacketReceiver} is to deliver incoming messages. Recall that interface \texttt{MessageListener} declares method \texttt{messageReceived}. When the \texttt{PacketReceiver}
// use InetAddress to get multicast group
multicastGroup = InetAddress.getByName( MULTICAST_ADDRESS );

// join multicast group to receive messages
multicastSocket.joinGroup( multicastGroup );

// set 5 second timeout when waiting for new packets
multicastSocket.setSoTimeout( 5000 );
}
} // end PacketReceiver constructor

// listen for messages from multicast group
public void run()
{
    // listen for messages until stopped
    while ( keepListening )
    {
        try // receive new DatagramPacket (blocking call)
        {
            multicastSocket.receive( packet );
        } // end try
        catch ( SocketTimeoutException socketTimeoutException )
        {  
            continue; // continue to next iteration to keep listening
        } // end catch
        catch ( IOException ioException )
        {
            ioException.printStackTrace();
        } // end catch
    } // end while

    // put message data in a String
    message = new String( packet.getData() );

    message = message.trim(); // trim whitespace from message

    // tokenize message to retrieve user name and message body
    StringTokenizer tokenizer = new StringTokenizer( message, MESSAGE_SEPARATOR );

    // DeitelMessengerServer in a separate thread. (Part 2 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 InterruptedIOException, 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 `IOException`, line 72 `break` 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 `DatagramPacket`s 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

---

```java
// Fig. 24.28: ClientGUI.java
// ClientGUI provides a user interface for sending and receiving messages to and from the DeitelMessengerServer.
package com.deitel.messenger;

import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.SwingUtilities;
import javax.swing.border.BevelBorder;

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

© Copyright 1997–2007 by Pearson Education, Inc. All Rights Reserved.
24.10  Case Study: DeitelMessenger Server and Client

public class ClientGUI extends JFrame {
    private JMenu serverMenu; // for connecting/disconnecting server
    private JTextArea messageArea; // displays messages
    private JTextArea inputArea; // inputs messages
    private JButton connectButton; // button for connecting
    private JMenuItem connectMenuItem; // menu item for connecting
    private JButton disconnectButton; // button for disconnecting
    private JMenuItem disconnectMenuItem; // menu item for disconnecting
    private JButton sendButton; // sends messages
    private JLabel statusBar; // label for connection status
    private String userName; // userName to add to outgoing messages
    private MessageManager messageManager; // communicates with server
    private MessageListener messageListener; // receives incoming messages

    // ClientGUI constructor
    public ClientGUI( MessageManager manager ) {
        super( "Deitel Messenger" );
        messageManager = manager; // set the MessageManager
        // create MyMessageListener for receiving messages
        messageListener = new MyMessageListener();
        serverMenu = new JMenu ( "Server" ); // create Server JMenu
        serverMenu.setMnemonic( 'S' ); // set mnemonic for server menu
        JMenuBar menuBar = new JMenuBar(); // create JMenuBar
        menuBar.add( serverMenu ); // add server menu to menu bar
        setJMenuBar( menuBar ); // add JMenuBar to application

        // create ImageIcon for connect buttons
        Icon connectIcon = new ImageIcon( lus().getResource( "images/Connect.gif" ) );

        // create connectButton and connectMenuItem
        connectButton = new JButton( "Connect", connectIcon );
        connectMenuItem = new JMenuItem( "Connect", connectIcon );
        connectMenuItem.setMnemonic( 'C' );

        // create ConnectListener for connect buttons
        ActionListener connectListener = new ConnectListener();
        connectButton.addActionListener( connectListener );
        connectMenuItem.addActionListener( connectListener );

        // create ImageIcon for disconnect buttons
        Icon disconnectIcon = new ImageIcon( getClass().getResource( "images/Disconnect.gif" ) );

        // create disconnectButton and disconnectMenuItem
        disconnectButton = new JButton( "Disconnect", disconnectIcon );

        // ClientGUI subclass of JFrame for presenting a GUI for viewing and sending chat messages. (Part 2 of 6.)
disconnectMenuItem = new JMenuItem( "Disconnect", disconnectIcon );
disconnectMenuItem.setMnemonic( 'D' );

// disable disconnect button and menu item
disconnectButton.setEnabled( false );
disconnectMenuItem.setEnabled( false );

// create DisconnectListener for disconnect buttons
ActionListener disconnectListener = new DisconnectListener();
disconnectButton.addActionListener( disconnectListener );
disconnectMenuItem.addActionListener( disconnectListener );

// add connect and disconnect JMenuItems to fileMenu
serverMenu.add( connectMenuItem );
serverMenu.add( disconnectMenuItem );

// add connect and disconnect JButtons to buttonPanel
JPanel buttonPanel = new JPanel();
buttonPanel.add( connectButton );
buttonPanel.add( disconnectButton );

messageArea = new JTextArea(); // displays messages
messageArea.setEditable( false ); // disable editing
messageArea.setWrapStyleWord( true ); // set wrap style to word
messageArea.setLineWrap( true ); // enable line wrapping

// put messageArea in JScrollPane to enable scrolling
JPanel messagePanel = new JPanel();
messagePanel.setLayout( new BorderLayout( 10, 10 ) );
messagePanel.add( new JScrollPane( messageArea ), BorderLayout.CENTER );

inputArea = new JTextArea( 4, 20 ); // for entering new messages
inputArea.setWrapStyleWord( true ); // set wrap style to word
inputArea.setLineWrap( true ); // enable line wrapping
inputArea.setEditable( false ); // disable editing

// create Icon for sendButton
Icon sendIcon = new ImageIcon( getClass().getResource( "images/Send.gif" ) );

sendButton = new JButton( "Send", sendIcon ); // create send button
sendButton.setEnabled( false ); // disable send button
sendButton.addActionListener( new ActionListener()
{
    public void actionPerformed( ActionEvent event )
    {
        messageManager.sendMessage( userName,
        inputArea.getText() ); // send message
    }
} );

Fig. 24.28 | ClientGUI subclass of JFrame for presenting a GUI for viewing and sending chat messages. (Part 3 of 6.)
Box box = new Box(BoxLayout.X_AXIS); // create new box for layout
box.add( new JScrollPane( inputArea ) ); // add input area to box
box.add( sendButton ); // add send button to box
messagePanel.add( box, BorderLayout.SOUTH ); // add box to panel

// create JLabel for statusBar with a recessed border
statusBar = new JLabel( "Not Connected" );
statusBar.setBorder( new BevelBorder( BevelBorder.LOWERED ) );

add( buttonPanel, BorderLayout.NORTH ); // add button panel
add( messagePanel, BorderLayout.CENTER ); // add message panel
add( statusBar, BorderLayout.SOUTH ); // add status bar

// add WindowListener to disconnect when user quits
addWindowListener( new WindowAdapter ()
{
   // disconnect from server and exit application
   public void windowClosing ( WindowEvent event )
   {
      messageManager.disconnect( messageListener );
      System.exit( 0 );
   } // end method windowClosing
} // end anonymous inner class
// end call to addWindowListener
// end ClientGUI constructor

// ConnectListener listens for user requests to connect to server
private class ConnectListener implements ActionListener
{
// connect to server and route messages to messageListener
   public void actionPerformed( ActionEvent event )
   {
      messageManager.connect( messageListener );

      // prompt for userName
      userName = JOptionPane.showInputDialog(
      ClientGUI.this, "Enter user name:" );

      messageArea.setText( "" ); // clear messageArea
      connectButton.setEnabled( false ); // disable connect
      connectMenuItem.setEnabled( false ); // disable connect
      disconnectButton.setEnabled( true ); // enable disconnect
      disconnectMenuItem.setEnabled( true ); // enable disconnect
      sendButton.setEnabled( true ); // enable send button

      // connect to server and route messages to messageListener
      messageManager.connect( messageListener );
   } // end method actionPerformed
} // end anonymous inner class
// end call to addActionListener
} // end method actionPerformed
// end anonymous inner class
} // end call to addActionListener

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

© Copyright 1997–2007 by Pearson Education, Inc. All Rights Reserved.
inputArea.setEditable( true ); // enable editing for input area
inputArea.requestFocus(); // set focus to input area
statusBar.setText( "Connected: " + userName ); // set text
} // end method actionPerformed
} // end ConnectListener inner class

// DisconnectListener listens for user requests to disconnect
// from DeitelMessengerServer
private class DisconnectListener implements ActionListener
{
    public void actionPerformed( ActionEvent event )
    {
        // disconnect from server and enable/disable GUI components
        messageManager.disconnect( messageListener );
        sendButton.setEnabled( false ); // disable send button
        disconnectButton.setEnabled( false ); // disable disconnect
        connectButton.setEnabled( true ); // enable connect
        inputArea.setEditable( false ); // disable editing
        connectMenuItem.setEnabled( true ); // enable connect
        disconnectMenuItem.setEnabled( false ); // disable disconnect
        disconnectButton.setEnabled( false ); // disable disconnect
        statusBar.setText( "Not Connected" ); // set status bar text
    } // end method actionPerformed
} // end DisconnectListener inner class

// MyMessageListener listens for new messages from MessageManager and
// displays messages in messageArea using MessageDisplayer.
private class MyMessageListener implements MessageListener
{
    public void messageReceived( String from, String message )
    {
        // when received, display new messages in messageArea
        SwingUtilities.invokeLater( new MessageDisplayer( from, message ) );
    } // end method messageReceived
} // end MyMessageListener inner class

// Displays new message by appending message to JTextArea. Should
// be executed only in Event thread; modifies live Swing component
private class MessageDisplayer implements Runnable
{
    private String fromUser; // user from which message came
    private String messageBody; // body of message

    // MessageDisplayer constructor
    public MessageDisplayer( String from, String body )
    {
        fromUser = from; // store originating user
        messageBody = body; // store message body
    } // end MessageDisplayer constructor

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

© Copyright 1997–2007 by Pearson Education, Inc. All Rights Reserved.
24.10 Case Study: DeitelMessenger Server and Client

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 textarea 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 userID 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
ceived, 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.

```java
package com.deitel.messenger.sockets.client;

import com.deitel.messenger.MessageManager;
import com.deitel.messenger.ClientGUI;

public class DeitelMessenger {
    public static void main(String args[]) {
        MessageManager messageManager; // declare MessageManager
        if (args.length == 0) {
            // connect to localhost
            messageManager = new SocketMessageManager("localhost");
        } else {
            // connect using command-line arg
            messageManager = new SocketMessageManager(args[0]);
        }
        // create GUI for SocketMessageManager
        ClientGUI clientGUI = new ClientGUI(messageManager);
        clientGUI.setSize(300, 400); // set window size
        clientGUI.setResizable(false); // disable resizing
        clientGUI.setVisible(true); // show window
    }
}
```

Fig. 24.29 | DeitelMessenger application for participating in a DeitelMessengerServer chat session. (Part 1 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

© Copyright 1997–2007 by Pearson Education, Inc. All Rights Reserved.
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.messengersockets.client.DeitelMessenger localhost
```
or

```
java com.deitel.messengersockets.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.