Lecture 15a

Networking: Streams-Based Sockets and Datagrams

OBJECTIVES

In this lecture you will learn:

- To implement networking applications that use sockets and datagrams.
- To implement clients and servers that communicate with one another.
- To implement network-based collaborative applications.
- To construct a multithreaded server.
- To use the `WebBrowser` control to add Web browsing capabilities to any application.
- To use .NET remoting to enable an application executing on one computer to invoke methods from an application executing on a different computer.

Outline

23.1 Introduction

The .NET FCL provides a number of built-in networking capabilities

- **Makes it easy to develop Internet- and Web-based applications**

Provide overview of the communication techniques and technologies used to transmit data over the Internet

Discuss basic concepts of establishing a connection between two applications using streams of data

- **Enables programs to communicate with one another as easily as writing to and reading from files on disk**

Discuss connectionless techniques for transmitting data between applications that is less reliable but much more efficient

23.2 Connection-Oriented vs. Connectionless Communication

Two primary approaches to communicating between applications

- **TCP is a representative of connection-oriented communications**
  - **Similar to the telephone system**
    - **A connection is established and held for the length of the session**
  - **Handshaking**
    - **Computers send each other control information**
      - **Through a technique called initiate an end-to-end connection**
  - **Data is sent in packets**
    - **Contain pieces of the data along with information that helps the Internet route the packets to the proper destination**
  - **Ensures reliable communications on unreliable networks**
    - **Guarantee that sent packets will arrive at the intended receiver undamaged and be reassembled in the correct sequence**
  - **The Internet does not guarantee anything about the packets sent**

23.2 Connection-Oriented vs. Connectionless Communication (Cont.)

- **UDP is a representative of connectionless communications**
  - **Similar to the postal service:**
    - **Two letters mailed from the same place and to the same destination may take two dramatically different paths through the system and even arrive at different times, or not at all**
  - **The two computers do not handshake before transmission**
    - **Reliability is not guaranteed**
      - **Data sent may never reach the intended recipient**
  - **Avoids the overhead associated with handshaking and enforcing reliability**
    - **Less information often needs to be passed between the hosts**

23.3 Protocols for Transporting Data

Protocols

- **Sets of rules that govern how two entities should interact**
- **Networking capabilities are defined in the `System.Net.Sockets` namespace**
  - **Transmission Control Protocol (TCP)**
  - **User Datagram Protocol (UDP)**
- **TCP**
  - **A connection-oriented communication protocol which guarantees that sent packets will arrive at the intended receiver undamaged and in the correct sequence**
    - **If packets are lost, TCP ensures that the packets are sent again**
    - **If the packets arrive out of order, TCP reassembles them in the correct order transparently to the receiving application**
    - **If duplicate packets arrive, TCP discards them**
  - **Send information across a network as simply and reliably as writing to a file on a local computer**

23.3 Protocols for Transporting Data (Cont.)

- **UDP**
  - **Incurs the minimum overhead necessary to communicate between applications**
    - **Do not need to carry the information that TCP packets carry to ensure reliability**
    - **Reduces network traffic relative to TCP due to the absence of handshaking, retransmissions, etc**
  - **Makes no guarantees that packets, called datagrams, will reach their destination or arrive in their original order**

23.4 Establishing a Simple TCP Client (Using Stream Sockets)

Typically with TCP, a server waits for a connection request

- **Contains a control statement or block of code that executes until the server receives the request**
- **On receiving a request, the server establishes a connection to the client**

`Socket` **object**

- **Manages the connection between server and client**

23.4 Establishing a Simple TCP Client (Using Stream Sockets) (Cont.)

Establish a simple server with TCP and stream sockets requires 5 steps (Uses namespace `System.Net.Sockets`)

- **Step 1: Create an object of class `TcpListener` of namespace**
  - **Using an IP Address and port number**
- **Step 2: Call `TcpListener`'s `Start` method**
  - **Causes the `TcpListener` object to begin listening for connection requests**
  - **An object of class `Socket` manages a connection to client**
  - **Method `AcceptSocket` of class `TcpListener` accepts a connection request**
- **Step 3: Establishes the streams used for communication with the client**
  - **Create a `NetworkStream` object that uses the `Socket` object**
    - **Represent the connection to perform the sending and receiving of data**
- **Step 4: The server and client communicate using the connection established in the third step**
- **Step 5: Terminate connections**
  - **Server calls method `Close` of the `BinaryReader`, `BinaryWriter`, `NetworkStream` and `Socket` to terminate the connection (in reverse order of their creation)**

23.5 Establishing a Simple TCP Client

`BinaryReader` – Reads primitive data types as binary values in a specific encoding

```
using (BinaryReader reader =
        new BinaryReader(File.Open(fileName, FileMode.Open)))
            {
                aspectRatio = reader.ReadSingle();
                tempDirectory = reader.ReadString();
                autoSaveTime = reader.ReadInt32();
                showStatusBar = reader.ReadBoolean();
            }
```

`BinaryWriter`- Writes primitive types in binary to a stream and supports writing strings in a specific encoding

```
using (BinaryWriter writer =
            new BinaryWriter(File.Open(fileName, FileMode.Create)))
        {
            writer.Write(1.250F);
            writer.Write(@"c:\Temp");
            writer.Write(10);
            writer.Write(true);
        }
```

Software Engineering Observation 23.1

Port numbers can have values between 0 and 65535. Many operating systems reserve port numbers below 1024 for system services (such as e-mail and Web servers). Applications must be granted special privileges to use these reserved port numbers.

Software Engineering Observation 23.2

Multithreaded servers can efficiently manage simultaneous connections with multiple clients. This architecture is precisely what popular UNIX and Windows network servers use.

Software Engineering Observation 23.3

A multithreaded server can be implemented to create a thread that manages network I/O across a `Socket` object returned by method `AcceptSocket`. A multithreaded server also can be implemented to maintain a pool of threads that manage network I/O across newly created `Sockets`.

Performance Tip 23.1

In high-performance systems with abundant memory, a multithreaded server can be implemented to create a pool of threads. These threads can be assigned quickly to handle network I/O across multiple `Sockets`. Thus, when a connection is received, the server does not incur the overhead of thread creation.

23.5 Establishing a Simple TCP Client (Using Stream Sockets)

There are 4 steps to creating a simple TCP client:

- **Step 1: Create an object of class `TcpClient` and connect to the server using method `Connect`**
  - **If the connection is successful, returns a positive integer (a `Socket` is created)**
    - **Otherwise, it returns 0**
- **Step 2: The `TcpClient` uses its `GetStream` method to get a `NetworkStream` so that it can write to and read from the server**
  - **Use the `NetworkStream` object to create a `BinaryWriter` and a `BinaryReader` that will be used to send information to and receive information from the server**
- **Step 3: The client and the server communicate**
- **Step 4: Require the client to close the connection by calling method `Close`**
  - **For `BinaryReader`, `BinaryWriter`, `NetworkStream` and `TcpClient` (in reverse order of their creation)**

23.6 Client/Server Interaction with Stream-Socket Connections

WPF controls are not thread safe

- **A control that is modified from multiple threads is not guaranteed to be modified correctly**

The *Visual Studio Documentation* recommends that only the thread which created the GUI should modify the controls

- **Class `Control`'s `Dispatcher` object method `Invoke` or `BeginInvoke` to help ensure this**

Localhost

- **A.K.A. the loopback IP address**
- **Equivalent to the IP address 127.0.0.1**

```
<Window x:Class="WPFChatServer.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WPFChatServer" mc:Ignorable="d"
        Title="Chat server" Height="492.483" Width="519.675"
        Closing="Window_Closing">
    <Grid  >
        <Grid.RowDefinitions>
            <RowDefinition Height="49*"/>
            <RowDefinition Height="412*"/>
        </Grid.RowDefinitions>
        <TextBox x:Name="TxtInput" HorizontalAlignment="Left" Height="27"
                 Margin="8" TextWrapping="NoWrap" VerticalAlignment="Top"
                 Width="488" VerticalScrollBarVisibility="Disabled"
                 FontWeight="Bold" MaxLines="1"
                 IsEnabled="False"   KeyDown="TxtInput_KeyDown"/>
        <TextBox x:Name="TxtDisplay" HorizontalAlignment="Left" Height="394"
                 Margin="8,4,0,0" Grid.Row="1" TextWrapping="Wrap"
                 VerticalAlignment="Top"
                 Width="488" VerticalScrollBarVisibility="Auto" FontWeight="Bold"
                 AcceptsReturn="True" AcceptsTab="True"/>
    </Grid>
</Window>
```

Outline

ChatServer.cs

(1 of 7)

```
// ChatServer.cs
// Set up a server that will receive a connection from a client, send a
// string to the client, chat with the client and close the connection.
using System;
using System.Windows;
using System.Threading;
using System.Net;
using System.Net.Sockets;
using System.IO;
using System.Windows.Input;
public partial class MainWindow : Window
{

   private Socket connection; // Socket for accepting a connection
   private Thread readThread; // Thread for processing incoming messages
   private NetworkStream socketStream; // network data stream
   private BinaryWriter writer; // facilitates writing to the stream
   private BinaryReader reader; // facilitates reading from the stream




   // initialize thread for reading
  private void MainWindow ( )
   {
      InitializeComponent();
      readThread = new Thread( new ThreadStart( RunServer ) );
      readThread.Start();
   } // end method CharServerForm_Load
```

Using namespaces for networking capabilities

Create `private` instance variables for networking purposes

Create a `Thread` that will accept connections from clients

Starts the `Thread`

Outline

```
30
31    // close all threads associated with this application
32    private void Window_Closing( object sender,
33                                System.ComponentModel.CancelEventArgs e
34    {
35       System.Environment.Exit( System.Environment.ExitCode );
36    } // end method CharServerForm_FormClosing
37
38
39
40
41
42    // method DisplayMessage sets displayTextBox's Text property
43    // in a thread-safe manner
44    private void DisplayMessage( string message )
45    {
46       // if modifying displayTextBox is not thread safe
47       if (!TxtDisplay.Dispatcher.CheckAccess() )
48       {
49          // use inherited method Invoke to execute DisplayMessage
50          // via a delegate
51          TxtDisplay.Dispatcher.Invoke(new Action(()
52                                        => TxtDisplay.Text += message));
53       } // end if
54       else // OK to modify displayTextBox in current thread
55          TxtDisplay.Text += message;
56    } // end method DisplayMessage
```

Terminate program and close its threads

ChatServer.cs

(2 of 7)

Returns true if the current thread is not allowed to modify this control directly

Execute `Action` on the UI Thread

Appends message to `TxtDisplay`

Outline

ChatServer.cs

```
57
58
59
60
61
62    // method DisableInput sets inputTextBox's ReadOnly property
63    // in a thread-safe manner
64    private void EnableInput( bool value )
65    {
66       // if modifying inputTextBox is not thread safe
67       if ( !TxtInput.Dispatcher.CheckAccess() )
68       {
69          // use inherited method Invoke to execute DisableInput
70          // via a delegate
71          TxtInput.Dispatcher.Invoke(new Action(()
72                => TxtInput.IsEnabled = value));
73       } // end if
74       else // OK to modify inputTextBox in current thread
75          TxtInput.IsEnabled = value;
76    } // end method DisableInput
```

Returns `true` if the current thread is not allowed to modify this control directly

Pass a new `Action Delegate` to update the `IsEnabled` property by means the UI thread

Set `TxtInput.IsEnabled` to the value of the `boolean` argument

Outline

ChatServer.cs

```
77
78    // send the text typed at the server to the client
79    private void TxtInput_KeyDown( object sender, KeyEventArgs e )
80    {
81       // send the text to the client
82       try
83       {
84          if ( e.Key == Key.Enter && TxtInput.IsEnabled == true )
85          {
86             writer.Write( "SERVER>>> " + inputTextBox.Text );
87             TxtDisplay.Text += "\r\nSERVER>>> " + TxtInput.Text;
88
89             // if the user at the server signaled termination
90             // sever the connection to the client
91             if ( TxtInput.Text == "TERMINATE" )
92                connection?.Close();
93
94             TxtInput.Clear(); // clear the user's input
95          } // end if
96       } // end try
97       catch ( SocketException )
98       {
99          TxtDisplay.Text += "\nError writing object";
100      } // end catch
101   } // end method inputTextBox_KeyDown
```

Sends message via method `Write` of class `BinaryWriter`

Calls method `Close` of the `Socket` object to close the connection

Outline

ChatServer.cs

(5 of 7)

```
102
103   // allows a client to connect; displays text the client sends
104   public void RunServer()
105   {
106      TcpListener listener;
107      int counter = 1;
108
109      // wait for a client connection and display the text
110      // that the client sends
111      try
112      {
113         // Step 1: create TcpListener
114         IPAddress local = IPAddress.Parse( "127.0.0.1" );
115         listener = new TcpListener( local, 50000 );
116
117         // Step 2: TcpListener waits for connection request
118         listener.Start();
119
120         // Step 3: establish connection upon client request
121         while ( true )
122         {
123            DisplayMessage( "Waiting for connection\r\n" );
124
125            // accept an incoming connection
126            connection = listener.AcceptSocket();
127
128            // create NetworkStream object associated with socket
129            socketStream = new NetworkStream( connection );
```

Create instance of `TcpListener`

Instantiate the `TcpListener` object to listen for a connection request from a client at port 50000

Causes the `TcpListener` to begin waiting for request

Calls method `AcceptSocket` of the `TcpListener` object, which returns a `Socket` upon successful connection

Passes the `Socket` object as an argument to the constructor of a `NetworkStream` object

```
130
131         // create objects for transferring data across str
132         writer = new BinaryWriter( socketStream );
133         reader = new BinaryReader( socketStream );
134
135         DisplayMessage( "Connection " + counter + " received.\r\n" );
136
137         // inform client that connection was successfull
138         writer.Write( "SERVER>>> Connection successful" );
139
140         EnableInput( true ); // enable inputTextBox
141
142         string theReply = "";
143
144         // Step 4: read string data sent from client
145         do
146     {
147       try
148             {
149                // read the string sent to the server
150                theReply = reader.ReadString();
151
152                // display the message
153                DisplayMessage( "\r\n" + theReply );
154             } // end try
```

Create instances of `BinaryWriter` and `BinaryReader` classes for writing and reading data

ChatServer.cs

Send to the client a string notifying the user of a successful connection

Read a `string` from the stream

Outline

ChatServer.cs

(7 of 7)

```
155                catch ( Exception )
156                {
157                   // handle exception if error reading data
158                   break;
159                } // end catch
160             } while ( theReply != "CLIENT>>> TERMINATE"  &&
161                connection.Connected );
162
163             DisplayMessage( "\r\nUser terminated connection\r\n" );
164
165             // Step 5: close connection
166             writer?.Close();
167             reader?.Close();
168             socketStream?.Close();
169             connection?.Close();
170
171             EnableInput( false ); // disable InputTextBox
172             counter++;
173          } // end while
174       } // end try
175       catch ( Exception error )
176       {
177          MessageBox.Show( error.ToString() );
178       } // end catch
179    } // end method RunServer
180 } // end class ChatServerForm
```

Release program's resources by closing its connections

Outline

ChatClient.cs

(1 of 9)

```
// Fig. 23.2: ChatClient.cs
// Set up a client that will send information to and
// read information from a server.
using System;
using System.Windows.Forms;
using System.Threading;
using System.Net.Sockets;
using System.IO;

public partial class ChatClientForm : Form
{
   public ChatClientForm()
   {
      InitializeComponent();
   } // end constructor

   private NetworkStream output; // stream for receiving data
   private BinaryWriter writer; // facilitates writing to the stream
   private BinaryReader reader; // facilitates reading from the stream
   private Thread readThread; // Thread for processing incoming messages
   private string message = "";

   // initialize thread for reading
   private void ChatClientForm_Load( object sender, EventArgs e )
   {
      readThread = new Thread( new ThreadStart( RunClient ) );
      readThread.Start();
   } // end method ChatClientForm_Load
```

Using namespaces for networking capabilities

Create `private` instance variables for networking purposes

Create a `Thread` to handle all incoming messages

Starts the `Thread`

Outline

```
29
30    // close all threads associated with this application
31    private void ChatClientForm_FormClosing( object sender,
32       FormClosingEventArgs e )
33    {
34       System.Environment.Exit( System.Environment.ExitCode );
35    } // end method ChatClientForm_FormClosing
36
37    // delegate that allows method DisplayMessage to be called
38    // in the thread that creates and maintains the GUI
39    private delegate void DisplayDelegate( string message );
40
41    // method DisplayMessage sets displayTextBox's Text property
42    // in a thread-safe manner
43    private void DisplayMessage( string message )
44    {
45       // if modifying displayTextBox is not thread safe
46       if ( displayTextBox.InvokeRequired )
47       {
48          // use inherited method Invoke to execute DisplayMessage
49          // via a delegate
50          Invoke( new DisplayDelegate( DisplayMessage ),
51             new object[] { message } );
52       } // end if
53       else // OK to modify displayTextBox in current thread
54          displayTextBox.Text += message;
55    } // end method DisplayMessage
```

Terminate program and close its threads

ChatClient.cs

(2 of 9)

Represents methods that take a `string` argument and do not return a value

Returns `true` if the current thread is not allowed to modify this control directly

Pass a new `DisplayDelegate` representing the method `DisplayMessage` and a new object array consisting of the argument `message`

Appends message to `displayTextBox`

Outline

```
56
57    // delegate that allows method DisableInput to be called
58    // in the thread that creates and maintains the GUI
59    private delegate void DisableInputDelegate( bool value );
60
61    // method DisableInput sets inputTextBox's ReadOnly property
62    // in a thread-safe manner
63    private void DisableInput( bool value )
64  {
65       // if modifying inputTextBox is not thread safe
66       if ( inputTextBox.InvokeRequired )
67       {
68          // use inherited method Invoke to execute DisableInput
69          // via a delegate
70          Invoke( new DisableInputDelegate( DisableInput ),
71             new object[] { value } );
72       } // end if
73       else // OK to modify inputTextBox in current thread
74          inputTextBox.ReadOnly = value;
75    } // end method DisableInput
```

Delegate for `DisableInput`

ChatClient.cs

Returns `true` if the current thread is not allowed to modify this control directly

Pass a new `DisableInputDelegate` representing the method `DisableInput` and a new object array consisting of the argument `value`

Set `inputTextBox.ReadOnly` to the value of the `boolean` argument

Outline

```
76
77    // sends text the user typed to server
78    private void inputTextBox_KeyDown( object sender, KeyEventArgs e )
79    {
80       try
81       {
82          if ( e.KeyCode == Keys.Enter && inputTextBox.ReadOnly == false )
83          {
84             writer.Write( "CLIENT>>> " + inputTextBox.Text );
85             displayTextBox.Text += "\r\nCLIENT>>> " + inputTextBox.Text;
86             inputTextBox.Clear();
87          } // end if
88       } // end try
89       catch ( SocketException )
90       {
91          displayTextBox.Text += "\nError writing object";
92       } // end catch
93    } // end method inputTextBox_KeyDown
```

ChatClient.cs

Sends message via method `Write` of class `BinaryWriter`

Outline

ChatClient.cs

(5 of 9)

```
// connect to server and display server-generated text
public void RunClient()
{
   TcpClient client;

   // instantiate TcpClient for sending data to server
   try
   {
      DisplayMessage( "Attempting connection\r\n" );

      // Step 1: create TcpClient and connect to server
      client = new TcpClient();
      client.Connect( "127.0.0.1", 50000 );

      // Step 2: get NetworkStream associated with TcpClient
      output = client.GetStream();

      // create objects for writing and reading across stream
      writer = new BinaryWriter( output );
      reader = new BinaryReader( output );

      DisplayMessage( "\r\nGot I/O streams\r\n" );
      DisableInput( false ); // enable inputTextBox
```

Create instance of `TcpClient`

Instantiate the `TcpClient`, then call its `Connect` method to establish a connection

Retrieve the `NetworkStream` object that send data to and receive data from the server

Create instances of `BinaryWriter` and `BinaryReader` classes for writing and reading data

Outline

ChatClient.cs

```
118
119         // loop until server signals termination
120         do
121         {
122            // Step 3: processing phase
123            try
124            {
125               // read message from server
126               message = reader.ReadString();
127               DisplayMessage( "\r\n" + message );
128            } // end try
129            catch ( Exception )
130            {
131               // handle exception if error in reading server data
132               System.Environment.Exit( System.Environment.ExitCode );
133            } // end catch
134         } while ( message != "SERVER>>> TERMINATE" );
135
136         // Step 4: close connection
137         writer.Close();
138         reader.Close();
139         output.Close();
140         client.Close();
141
142         Application.Exit();
143      } // end try
```

Read and display the message

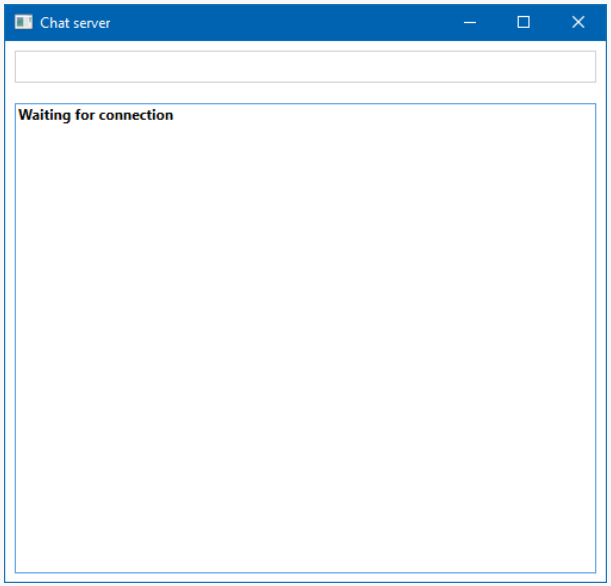
Release program's resources by closing its connections

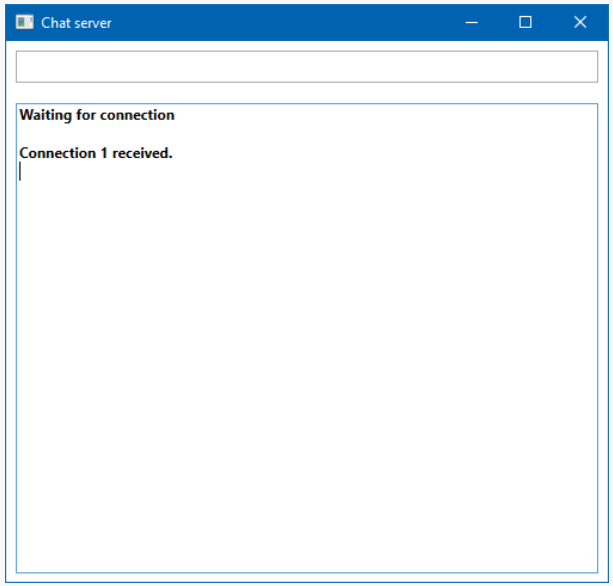
Outline

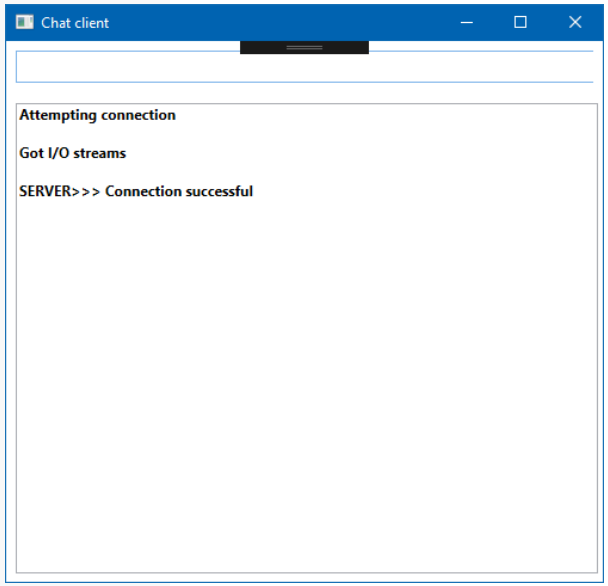
```
      catch ( Exception error )
      {
         // handle exception if error in establishing connection
         MessageBox.Show( error.ToString(), "Connection Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error );
         System.Environment.Exit( System.Environment.ExitCode );
      } // end catch
   } // end method RunClient
} // end class ChatClientForm
```

ChatClient.cs

(7 of 9)



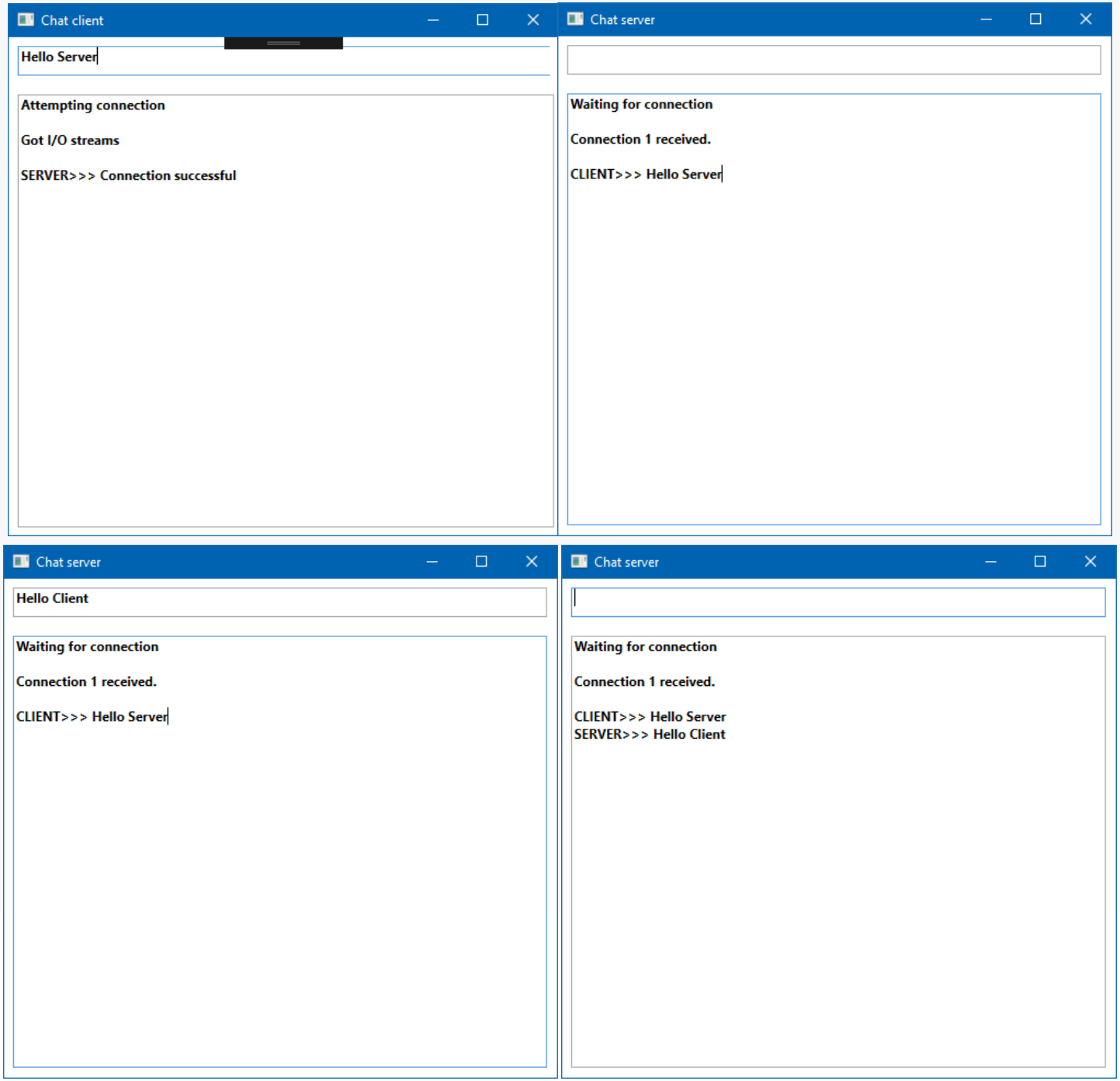




Outline

ChatClient.cs

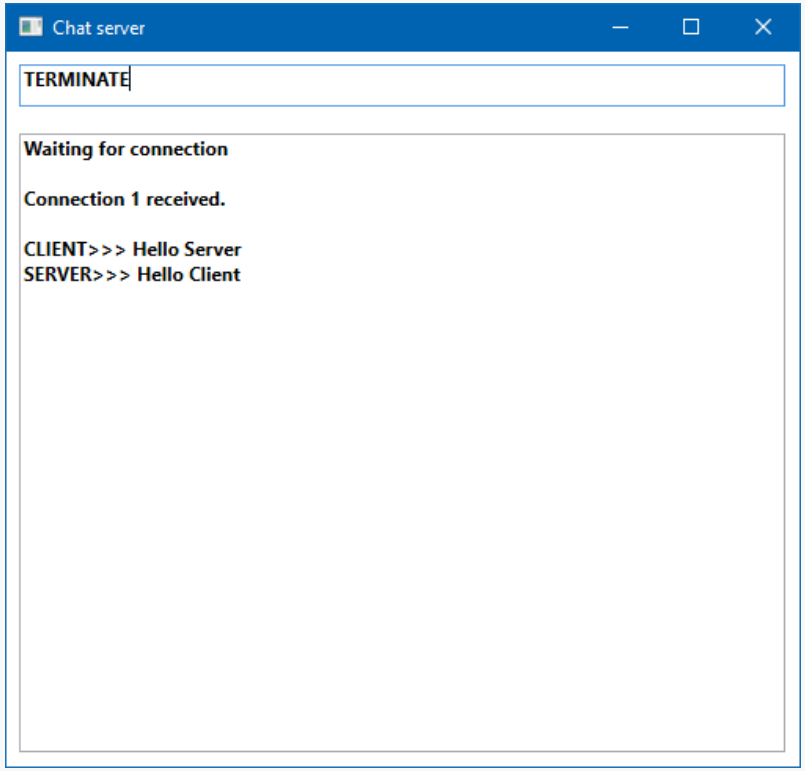
(8 of 9)

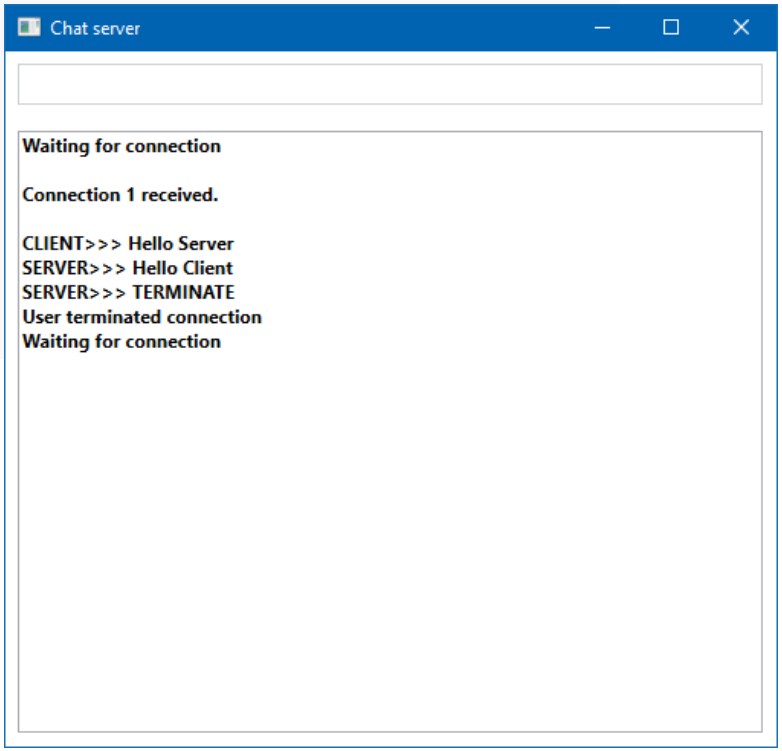


Outline

ChatClient.cs

(9 of 9)





23.7 Connectionless Client/Server Interaction with Datagrams

Connectionless transmission

- **Bundles and sends information in packets called datagrams**
- **Class** `UdpClient`
  - **For connectionless transmission**
  - **Methods** `Send` **and** `Receive`
    - **Transmit data with** `Socket`**'s** `SendTo` **method**
    - **Read data with** `Socket`**'s** `ReceiveFrom` **method**
- **Class** `IPEndPoint`
  - **Hold the IP address and port number of the client(s)**

Outline

PacketServer.cs

(1 of 3)

```
// Fig. 23.3: PacketServer.cs
// Set up a server that will receive packets from a
// client and send the packets back to the client.
using System;
using System.Windows.Forms;
using System.Net;
using System.Net.Sockets;
using System.Threading;

public partial class PacketServerForm : Form
{
   public PacketServerForm()
   {
      InitializeComponent();
   } // end constructor

   private UdpClient client;
   private IPEndPoint receivePoint;

   // initialize variables and thread for receiving packets
   private void PacketServerForm_Load( object sender, EventArgs e )
   {
      client = new UdpClient( 50000 );
      receivePoint = new IPEndPoint( new IPAddress( 0 ), 0 );
      Thread readThread =
         new Thread( new ThreadStart( WaitForPackets ) );
      readThread.Start();
   } // end method PacketServerForm_Load
```

Using namespaces for networking capabilities

Create `private` instance variables of class `UdpClient` and `IPEndPoint`

Receives data at port 50000

Hold the IP address and port number of the clients that transit to `PacketServerForm`

Create a `Thread` to handle packages

Starts the `Thread`

Outline

Terminate program and close its threads

PacketServer.cs

(2 of 3)

```
   // shut down the server
   private void PacketServerForm_FormClosing( object sender,
      FormClosingEventArgs e )
   {
      System.Environment.Exit( System.Environment.ExitCode );
   } // end method PacketServerForm_FormClosing

   // delegate that allows method DisplayMessage to be called
   // in the thread that creates and maintains the GUI
   private delegate void DisplayDelegate( string message );

   // method DisplayMessage sets displayTextBox's Text property
   // in a thread-safe manner
   private void DisplayMessage( string message )
{
      // if modifying displayTextBox is not thread safe
      if ( displayTextBox.InvokeRequired )
      {
         // use inherited method Invoke to execute DisplayMessage
         // via a delegate
         Invoke( new DisplayDelegate( DisplayMessage ),
            new object[] { message } );
      } // end if
      else // OK to modify displayTextBox in current thread
         displayTextBox.Text += message;
   } // end method DisplayMessage
```

Outline

PacketServer.cs

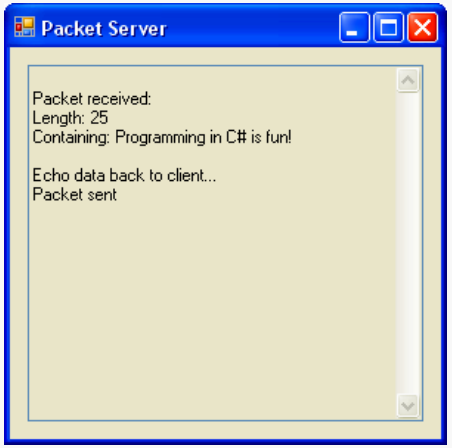
(3 of 3)

```
   // wait for a packet to arrive
   public void WaitForPackets()
   {
      while ( true )
      {
         // set up packet
         byte[] data = client.Receive( ref receivePoint );
         DisplayMessage( "\r\nPacket received:" +
            "\r\nLength: " + data.Length +
            "\r\nContaining: " +
            System.Text.Encoding.ASCII.GetString( data ) );

         // echo information from packet back to client
         DisplayMessage( "\r\n\r\nEcho data back to client..." );
         client.Send( data, data.Length, receivePoint );
         DisplayMessage( "\r\nPacket sent\r\n" );
      } // end while
   } // end method WaitForPackets
} // end class PacketServerForm
```

Receives a byte array from the client

Return the `string` of the byte array

Echos the data back to the client using `UdpClient`'s method `Send`



Outline

PacketClient.cs

(1 of 4)

```
// Fig. 23.4: PacketClient.cs
// Set up a client that sends packets to a server and receives
// packets from a server.
using System;
using System.Windows.Forms;
using System.Net;
using System.Net.Sockets;
using System.Threading;

public partial class PacketClientForm : Form
{
   public PacketClientForm()
   {
      InitializeComponent();
   } // end constructor

   private UdpClient client;
   private IPEndPoint receivePoint;

   // initialize variables and thread for receiving packets
   private void PacketClientForm_Load( object sender, EventArgs e )
   {
      receivePoint = new IPEndPoint( new IPAddress( 0 ), 0 );
      client = new UdpClient( 50001 );
      Thread thread =
         new Thread( new ThreadStart( WaitForPackets ) );
      thread.Start();
   } // end method PacketClientForm_Load
```

Using namespaces for networking capabilities

Create `private` instance variables of class `UdpClient` and `IPEndPoint`

Hold the IP address and port number of the clients that transit to `PacketClientForm`

Receives data at port 50001

Create a `Thread` to handle packages

Starts the `Thread`

Outline

```
29
30    // shut down the client
31    private void PacketClientForm_FormClosing( object sender,
32       FormClosingEventArgs e )
33    {
34       System.Environment.Exit( System.Environment.ExitCode );
35    } // end method PacketClientForm_FormClosing
36
37    // delegate that allows method DisplayMessage to be called
38    // in the thread that creates and maintains the GUI
39    private delegate void DisplayDelegate( string message );
40
41    // method DisplayMessage sets displayTextBox's Text property
42    // in a thread-safe manner
43    private void DisplayMessage( string message )
44    {
45       // if modifying displayTextBox is not thread safe
46       if ( displayTextBox.InvokeRequired )
47       {
48          // use inherited method Invoke to execute DisplayMessage
49          // via a delegate
50          Invoke( new DisplayDelegate( DisplayMessage ),
51             new object[] { message } );
52       } // end if
53       else // OK to modify displayTextBox in current thread
54          displayTextBox.Text += message;
55    } // end method DisplayMessage
```

Terminate program and close its threads

PacketClient.cs

(2 of 4)

Outline

PacketClient.cs

(3 of 4)

```
56
57    // send a packet
58    private void inputTextBox_KeyDown( object sender, KeyEventArgs e )
59    {
60       if ( e.KeyCode == Keys.Enter )
61       {
62          // create packet (datagram) as string
63          string packet = inputTextBox.Text;
64          displayTextBox.Text +=
65             "\r\nSending packet containing: " + packet;
66
67          // convert packet to byte array
68          byte[] data = System.Text.Encoding.ASCII.GetBytes( packet );
69
70          // send packet to server on port 50000
71          client.Send( data, data.Length, "127.0.0.1", 50000 );
72          displayTextBox.Text += "\r\nPacket sent\r\n";
73          inputTextBox.Clear();
74       } // end if
75    } // end method inputTextBox_KeyDown
```

Converts the `string` that the user entered to a byte array

Send the byte array to the `PacketServerForm` that is located on the localhost

Outline

PacketClient.cs

(4 of 4)

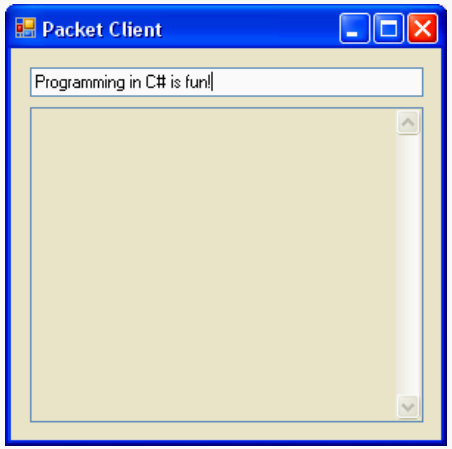
```
   // wait for packets to arrive
   public void WaitForPackets()
   {
      while ( true )
      {
         // receive byte array from server
         byte[] data = client.Receive( ref receivePoint );

         // output packet data to TextBox
         DisplayMessage( "\r\nPacket received:" +
            "\r\nLength: " + data.Length + "\r\nContaining: " +
            System.Text.Encoding.ASCII.GetString( data ) + "\r\n" );
      } // end while
   } // end method WaitForPackets
}// end class PacketClientForm
```

Receives a byte array from the server

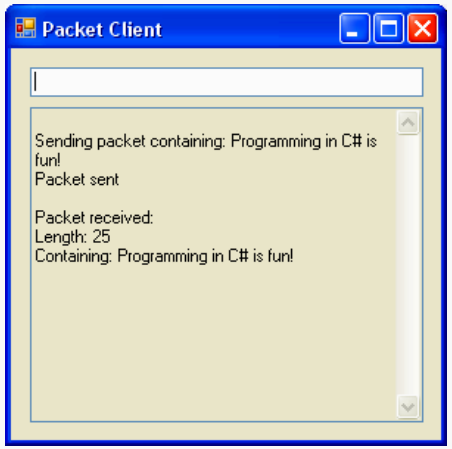
Return the `string` of the byte array

Display its contents in the `TextBox`

(a) `Packet Client` window before sending a packet to the server



(b) `Packet Client` window after sending a packet to the server and receiving it back



23.8 Client/Server Tic-Tac-Toe Using a Multithreaded Server

Multithreaded servers can manage many simultaneous connections with multiple clients

Outline

TicTacToeServer.cs

(1 of 10)

```
// Fig. 23.5: TicTacToeServer.cs
// This class maintains a game of Tic-Tac-Toe for two
// client applications.
using System;
using System.Windows.Forms;
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.IO;

public partial class TicTacToeServerForm : Form
{
   public TicTacToeServerForm()
   {
      InitializeComponent();
   } // end constructor

   private byte[] board; // the local representation of the game board
   private Player[] players; // two Player objects
   private Thread[] playerThreads; // Threads for client interaction
   private TcpListener listener; // listen for client connection
   private int currentPlayer; // keep track of whose turn it is
   private Thread getPlayers; // Thread for acquiring client connections
   internal bool disconnected = false; // true if the server closes
```

Using namespaces for networking capabilities

Create instance variables for implementing the server side of the Tic-Tac-Toe networking game

Outline

```
25
26    // initialize variables and thread for receiving clients
27    private void TicTacToeServerForm_Load( object sender, EventArgs e )
28    {
29       board = new byte[ 9 ];
30       players = new Player[ 2 ];
31       playerThreads = new Thread[ 2 ];
32       currentPlayer = 0;
33
34       // accept connections on a different thread
35       getPlayers = new Thread( new ThreadStart( SetUp ) );
36       getPlayers.Start();
37    } // end method TicTacToeServerForm_Load
38
39    // notify Players to stop Running
40    private void TicTacToeServerForm_FormClosing( object sender,
41       FormClosingEventArgs e )
42    {
43       disconnected = true;
44       System.Environment.Exit( System.Environment.ExitCode );
45    } // end method TicTacToeServerForm_FormClosing
46
47    // delegate that allows method DisplayMessage to be called
48    // in the thread that creates and maintains the GUI
49    private delegate void DisplayDelegate( string message );
```

Store the moves the players have made

Arrays referencing `Player` and `Thread` objects

Keep track of the current player

Create and start `Thread` to accept connections so that the current thread does not block while awaiting players

Notify and terminate program and close its associated threads

Outline

TicTacToeServer.cs

(3 of 10)

```
   // method DisplayMessage sets displayTextBox's Text property
   // in a thread-safe manner
   internal void DisplayMessage( string message )
   {
      // if modifying displayTextBox is not thread safe
      if ( displayTextBox.InvokeRequired )
      {
         // use inherited method Invoke to execute DisplayMessage
         // via a delegate
         Invoke( new DisplayDelegate( DisplayMessage ),
            new object[] { message } );
      } // end if
      else // OK to modify displayTextBox in current thread
         displayTextBox.Text += message;
   } // end method DisplayMessage

   // accepts connections from 2 players
   public void SetUp()
   {
      DisplayMessage( "Waiting for players...\r\n" );

      // set up Socket
      listener =
         new TcpListener( IPAddress.Parse( "127.0.0.1" ), 50000 );
            listener.Start();
```

Creates a `TcpListerner` object to listen for requests on port 50000

Outline

```
76
77      // accept first player and start a player thread
78      players[ 0 ] = new Player( listener.AcceptSocket(), this, 0 );
79      playerThreads[ 0 ] =
80         new Thread( new ThreadStart( players[ 0 ].Run ) );
81      playerThreads[ 0 ].Start();
82
83      // accept second player and start another player thread
84      players[ 1 ] = new Player( listener.AcceptSocket(), this, 1 );
85      playerThreads[ 1 ] =
86         new Thread( new ThreadStart( players[ 1 ].Run ) );
87      playerThreads[ 1 ].Start();
88
89      // let the first player know that the other player has connected
90      lock ( players[ 0 ] )
91      {
92         players[ 0 ].threadSuspended = false;
93         Monitor.Pulse( players[ 0 ] );
94      } // end lock
95   } // end method SetUp
96
97   // determine if a move is valid
98   public bool ValidMove( int location, int player )
99   {
100      // prevent another thread from making a move
101      lock ( this )
102      {
103         // while it is not the current player's turn, wait
104         while ( player != currentPlayer )
105            Monitor.Wait( this );
```

Instantiate `Player` objects representing players

`TicTacToeServer.cs`

Creates two `Threads` that execute the `Run` methods of each `Player` object

Notify and unsuspend the first `Player` since other `Player` has connected

Allow only one move to be attempted at a time

Wait if it's not the player's turn

```
106
107         // if the desired square is not occupied
108         if ( !IsOccupied( location ) )
109         {
110            // set the board to contain the current player's mark
111            board[ location ] = ( byte ) ( currentPlayer == 0 ?
112               'X' : 'O' );
113
114            // set the currentPlayer to be the other player
115            currentPlayer = ( currentPlayer + 1 ) % 2;
116
117            // notify the other player of the move
118            players[ currentPlayer ].OtherPlayerMoved( location );
119
120            // alert the other player that it's time to move
121            Monitor.Pulse( this );
122            return true;
123         } // end if
124         else
125            return false;
126      } // end lock
127   } // end method ValidMove
```

Checks to see if the user have selected an unoccupied space

TicTacToeServer.cs

Place the mark on the local representation of the board

Determine and set the `currentPlayer` to the appropriate value

Notifies the other player that a move has been made

Invokes the `Pulse` method so that the waiting `Player` can validate a move

```
   // determines whether the specified square is occupied
   public bool IsOccupied( int location )
   {
      if ( board[ location ] == 'X' || board[ location ] == 'O' )
         return true;
      else
         return false;
   } // end method IsOccupied

   // determines if the game is over
   public bool GameOver()
   {
      // place code here to test for a winner of the game
      return false;
   } // end method GameOver
} // end class TicTacToeServerForm
```

Outline

TicTacToeServer.cs

(6 of 10)

```
// class Player represents a tic-tac-toe player
public class Player
{
   internal Socket connection; // Socket for accepting a connection
   private NetworkStream socketStream; // network data stream
   private TicTacToeServerForm server; // reference to server
   private BinaryWriter writer; // facilitates writing to the stream
   private BinaryReader reader; // facilitates reading from the stream
   private int number; // player number
   private char mark; // player's mark on the board
   internal bool threadSuspended = true; // if waiting for other player

   // constructor requiring Socket, TicTacToeServerForm and int
   // objects as arguments
   public Player( Socket socket, TicTacToeServerForm serverValue,
      int newNumber )
   {
      mark = (newNumber == 0 ? 'X' : 'O');
      connection = socket;
      server = serverValue;
      number = newNumber;

      // create NetworkStream object for Socket
      socketStream = new NetworkStream( connection );

      // create Streams for reading/writing bytes
      writer = new BinaryWriter( socketStream );
      reader = new BinaryReader( socketStream );
   } // end constructor
```

Create instance variables for implementing the player of the Tic-Tac-Toe networking game

TicTacToeServer.cs

(7 of 10)

Instantiate the `NetworkStream` object by passing the `Socket` object into the constructor

Instantiate the `BinaryWriter` and `BinaryReader` objects by passing the `NetworkStream` object into the constructor

```
175
176   // signal other player of move
177   public void OtherPlayerMoved( int location )
178   {
179      // signal that opponent moved
180            writer.Write( "Opponent moved." );
181            writer.Write( location ); // send location of move
182   } // end method OtherPlayerMoved
183
184   // allows the players to make moves and receive moves
185   // from the other player
186   public void Run()
187   {
188      bool done = false;
189
190      // display on the server that a connection was made
191      server.DisplayMessage( "Player " + ( number == 0 ? 'X' : 'O' )
192         + " connected\r\n" );
193
194      // send the current player's mark to the client
195      writer.Write( mark );
196
197      // if number equals 0 then this player is X,
198      // otherwise O must wait for X's first move
199      writer.Write( "Player " + ( number == 0 ?
200         "X connected.\r\n" : "O connected, please wait.\r\n" ) );
```

Notify player that the opponent has moved via method `Write` of `BinaryWriter`

TicTacToeServer.cs

(8 of 10)

Notify the server of a successful connection and send to the client the `char` that the client will place on the board when making a move

Outline

```
201
202     // X must wait for another player to arrive
203     if ( mark == 'X' )
204     {
205        writer.Write( "Waiting for another player." );
206
207        // wait for notification from server that another
208        // player has connected
209        lock ( this )
210        {
211           while ( threadSuspended )
212              Monitor.Wait( this );
213        } // end lock
214
215        writer.Write( "Other player connected. Your move." );
216     } // end if
217
218     // play game
219     while ( !done )
220     {
221        // wait for data to become available
222        while ( connection.Available == 0 )
223        {
224           Thread.Sleep( 1000 );
225
226           if ( server.disconnected )
227              return;
228        } // end while
```

Suspends the Player "X" thread until the server signals that Player O has connected

TicTacToeServer.cs

(9 of 10)

Notify player X of the current situation

Loops until `Socket` property `Available` indicates that there is information to receive from the `Socket`

If there is no information, `Thread` goes to sleep for one second

Checks whether server is disconnected

```
229
230         // receive data
231         int location = reader.ReadInt32();
232
233         // if the move is valid, display the move on the
234         // server and signal that the move is valid
235         if ( server.ValidMove( location, number ) )
236         {
237            server.DisplayMessage( "loc: " + location + "\r\n" );
238            writer.Write( "Valid move." );
239         } // end if
240         else // signal that the move is invalid
241            writer.Write( "Invalid move, try again." );
242
243         // if game is over, set done to true to exit while loop
244         if ( server.GameOver() )
245            done = true;
246      } // end while loop
247
248      // close the socket connection
249      writer.Close();
250      reader.Close();
251      socketStream.Close();
252      connection.Close();
253   } // end method Run
254} // end class Player
```

Read in an `int` representing the location in which the client wants to place a mark

TicTacToeServer.cs

Validates the player's move

Determine if the game is over

Release program's resources by closing its connections

Outline

TicTacToeClient.cs

(1 of 13)

```
// Fig. 23.6: TicTacToeClient.cs
// Client for the TicTacToe program.
using System;
using System.Drawing;
using System.Windows.Forms;
using System.Net.Sockets;
using System.Threading;
using System.IO;

public partial class TicTacToeClientForm : Form
{
   public TicTacToeClientForm()
   {
      InitializeComponent();
   } // end constructor

   private Square[ , ] board; // local representation of the game board
   private Square currentSquare; // the Square that this player chose
   private Thread outputThread; // Thread for receiving data from server
   private TcpClient connection; // client to establish connection
   private NetworkStream stream; // network data stream
   private BinaryWriter writer; // facilitates writing to the stream
   private BinaryReader reader; // facilitates reading from the stream
   private char myMark; // player's mark on the board
   private bool myTurn; // is it this player's turn?
   private SolidBrush brush; // brush for drawing X's and O's
   private bool done = false; // true when game is over
```

Using namespaces for networking capabilities

Create instance variables for implementing the client side of the Tic-Tac-Toe networking game

Outline

TicTacToeClient.cs

(2 of 13)

```
// initialize variables and thread for connecting to server
private void TicTacToeClientForm_Load( object sender, EventArgs e )
{
   board = new Square[ 3, 3 ];

   // create 9 Square objects and place them on the board
   board[ 0, 0 ] = new Square( board0Panel, ' ', 0 );
   board[ 0, 1 ] = new Square( board1Panel, ' ', 1 );
   board[ 0, 2 ] = new Square( board2Panel, ' ', 2 );
   board[ 1, 0 ] = new Square( board3Panel, ' ', 3 );
   board[ 1, 1 ] = new Square( board4Panel, ' ', 4 );
   board[ 1, 2 ] = new Square( board5Panel, ' ', 5 );
   board[ 2, 0 ] = new Square( board6Panel, ' ', 6 );
   board[ 2, 1 ] = new Square( board7Panel, ' ', 7 );
   board[ 2, 2 ] = new Square( board8Panel, ' ', 8 );

   // create a SolidBrush for writing on the Squares
   brush = new SolidBrush( Color.Black );

   // make connection to server and get the associated
   // network stream
   connection = new TcpClient( "127.0.0.1", 50000 );
   stream = connection.GetStream();
   writer = new BinaryWriter( stream );
   reader = new BinaryReader( stream );
```

Initialize the Tic-Tac-Toe board

Create a black `SolidBrush` object for coloring the “X”s and “O”s

Instantiate the instance variables

Outline

```
54
55      // start a new thread for sending and receiving messages
56      outputThread = new Thread( new ThreadStart( Run ) );
57      outputThread.Start();
58   } // end method TicTacToeClientForm_Load
59
60   // repaint the Squares
61   private void TicTacToeClientForm_Paint( object sender,
62      PaintEventArgs e )
63   {
64      PaintSquares();
65   } // end method TicTacToeClientForm_Load
66
67   // game is over
68   private void TicTacToeClientForm_FormClosing( object sender,
69      FormClosingEventArgs e )
70   {
71      done = true;
72      System.Environment.Exit( System.Environment.ExitCode );
73   } // end TicTacToeClientForm_FormClosing
74
75   // delegate that allows method DisplayMessage to be called
76   // in the thread that creates and maintains the GUI
77   private delegate void DisplayDelegate( string message );
```

Create and start `Thread` to read messages sent from the server to the client

Repaint the squares by calling `PaintSquare`

Notify and terminate program and close its associated threads

TicTacToeClient.cs

(3 of 13)

```
   // method DisplayMessage sets displayTextBox's Text property
   // in a thread-safe manner
   private void DisplayMessage( string message )
   {
      // if modifying displayTextBox is not thread safe
      if ( displayTextBox.InvokeRequired )
      {
         // use inherited method Invoke to execute DisplayMessage
         // via a delegate
         Invoke( new DisplayDelegate( DisplayMessage ),
            new object[] { message } );
      } // end if
      else // OK to modify displayTextBox in current thread
         displayTextBox.Text += message;
   } // end method DisplayMessage

   // delegate that allows method ChangeIdLabel to be called
   // in the thread that creates and maintains the GUI
   private delegate void ChangeIdLabelDelegate( string message );
```

```
98
99    // method ChangeIdLabel sets displayTextBox's Text property
100   // in a thread-safe manner
101   private void ChangeIdLabel( string label )
102   {
103      // if modifying idLabel is not thread safe
104      if ( idLabel.InvokeRequired )
105   {
106         // use inherited method Invoke to execute ChangeIdLabel
107         // via a delegate
108         Invoke( new ChangeIdLabelDelegate( ChangeIdLabel ),
109            new object[] { label } );
110      } // end if
111      else // OK to modify idLabel in current thread
112         idLabel.Text = label;
113   } // end method ChangeIdLabel
```

Outline

TicTacToeClient.cs

(5 of 13)

Outline

TicTacToeClient.cs

```
114
115   // draws the mark of each square
116   public void PaintSquares()
117   {
118      Graphics g;
119
120      // draw the appropriate mark on each panel
121      for ( int row = 0; row < 3; row++ )
122      {
123         for ( int column = 0; column < 3; column++ )
124         {
125            // get the Graphics for each Panel
126            g = board[ row, column ].SquarePanel.CreateGraphics();
127
128            // draw the appropriate letter on the panel
129            g.DrawString( board[ row, column ].Mark.ToString(),
130               board0Panel.Font, brush, 10, 8 );
131         } // end for
132      } // end for
133   } // end method PaintSquares
```

Create an instance of `Graphics` for repainting

Retrieve and reference the `Graphics` object for each `Panel`

Mark the appropriate square

Outline

TicTacToeClient.cs

(7 of 13)

```
134
135   // send location of the clicked square to server
136   private void square_MouseUp( object sender,
137      System.Windows.Forms.MouseEventArgs e )
138   {
139      // for each square check if that square was clicked
140      for ( int row = 0; row < 3; row++ )
141      {
142         for ( int column = 0; column < 3; column++ )
143         {
144            if ( board[ row, column ].SquarePanel == sender )
145            {
146               CurrentSquare = board[ row, column ];
147
148               // send the move to the server
149               SendClickedSquare( board[ row, column ].Location );
150            } // end if
151         } // end for
152      } // end for
153   } // end method square_MouseUp
154
155   // control thread that allows continuous update of the
156   // TextBox display
157   public void Run()
158 {
159      // first get players's mark (X or O)
160      myMark = reader.ReadChar();
161      ChangeIdLabel( "You are player \"" + myMark + "\"" );
162      myTurn = ( myMark == 'X' ? true : false );
```

Determine and update which square was clicked

Notify the server of the clicked square

Get and output the player's mark

```
163
164     // process incoming messages
165     try
166     {
167        // receive messages sent to client
168        while ( !done )
169           ProcessMessage( reader.ReadString() );
170     } // end try
171     catch ( IOException )
172     {
173        MessageBox.Show( "Server is down, game over", "Error",
174           MessageBoxButtons.OK, MessageBoxIcon.Error );
175     } // end catch
176  } // end method Run
177
178  // process messages sent to client
179  public void ProcessMessage( string message )
180  {
181     // if the move the player sent to the server is valid
182     // update the display, set that square's mark to be
183     // the mark of the current player and repaint the board
184     if ( message == "Valid move." )
185     {
186        DisplayMessage( "Valid move, please wait.\r\n" );
187        currentSquare.Mark = myMark;
188        PaintSquares();
189     } // end if
```

Invoke `ProcessMessage` with the return value of `ReadString` of the `BinaryReader` object as the argument

If the message indicates that a move is valid, the client sets its Mark to the current square and repaints the board

TicTacToeClient.cs

(8 of 13)

```
    else if ( message == "Invalid move, try again." )
    {
       // if the move is invalid, display that and it is now
       // this player's turn again
       DisplayMessage( message + "\r\n" );
       myTurn = true;
    } // end else if
    else if ( message == "Opponent moved." )
    {
       // if opponent moved, find location of their move
       int location = reader.ReadInt32();

       // set that square to have the opponents mark and
       // repaint the board
       board[ location / 3, location % 3 ].Mark =
          ( myMark == 'X' ? 'O' : 'X' );
       PaintSquares();

       DisplayMessage( "Opponent moved.  Your turn.\r\n" );

       // it is now this player's turn
       myTurn = true;
    } // end else if
    else
       DisplayMessage( message + "\r\n" ); // display message
 } // end method ProcessMessage
```

If the message indicates that a move is invalid, the client notifies the user to click a different square

Update the player's turn

TicTacToeClient.cs

(9 of 13)

If the message indicates that the opponent made a move, read an `int` from the server specifying where on the board the client should place the opponent's Mark

Update the player's turn

Display the read message

Outline

TicTacToeClient.cs

```
216
217   // sends the server the number of the clicked square
218   public void SendClickedSquare( int location )
219   {
220      // if it is the current player's move right now
221      if ( myTurn )
222      {
223         // send the location of the move to the server
224         writer.Write( location );
225
226         // it is now the other player's turn
227         myTurn = false;
228      } // end if
229   } // end method SendClickedSquare
230
231   // write-only property for the current square
232   public Square CurrentSquare
233   {
234      set
235      {
236         currentSquare = value;
237      } // end set
238   } // end property CurrentSquare
239} // end class TicTacToeClientForm
```

Determine which player's turn it is

Notify the location of the move via `BinaryWriter`'s method `Write`

Update the player's turn

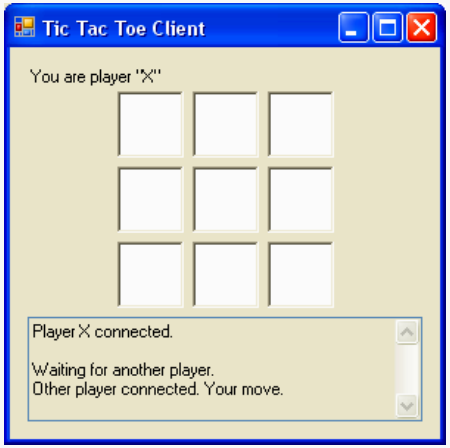
Outline

`TicTacToeClient.cs`

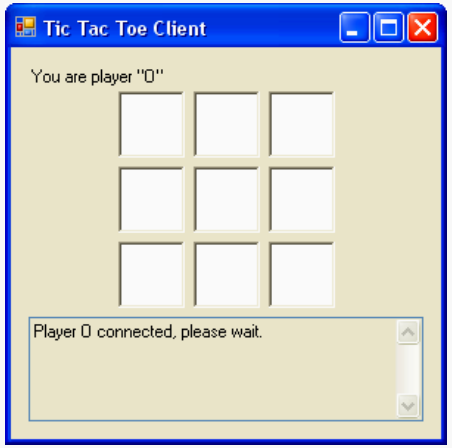
(11 of 13)

At the start of the game.

(a)

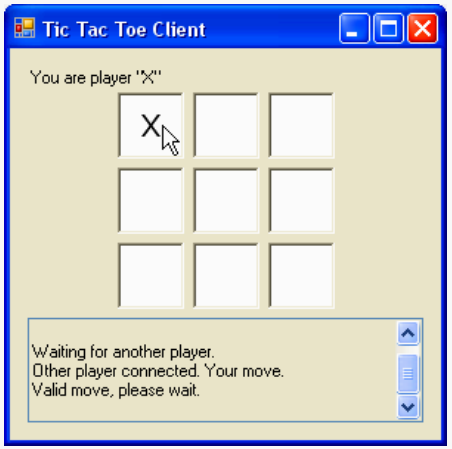


(b)

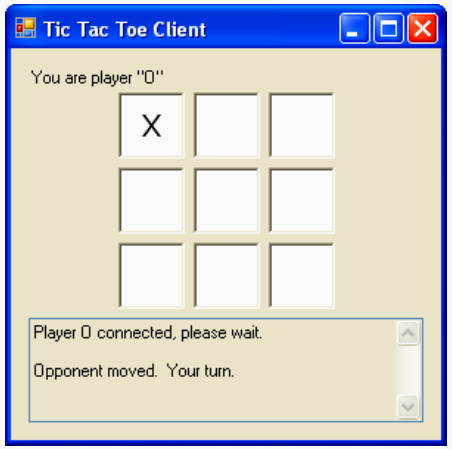


After Player X makes the first move.

(c)



(d)



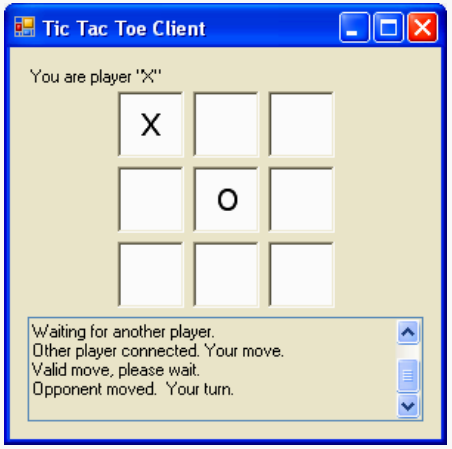
Outline

TicTacToeClient.cs

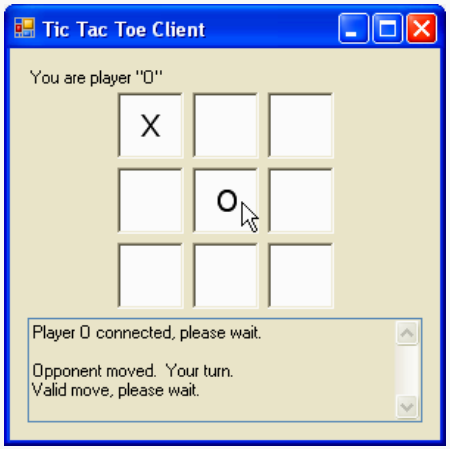
(12 of 13)

After Player O makes the second move.

(e)

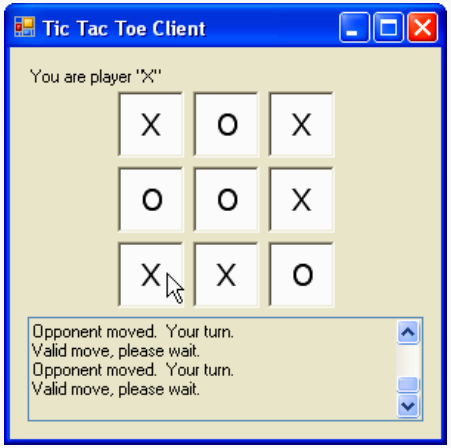


(f)

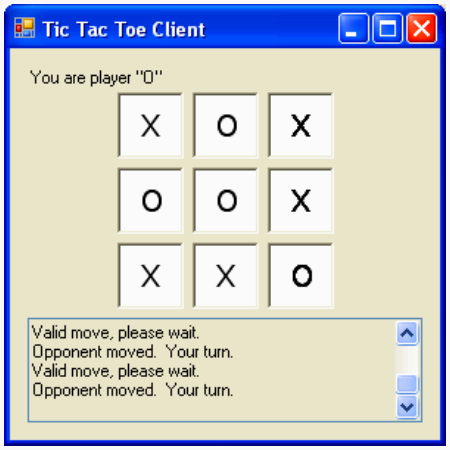


After Player X makes the final move.

(g)

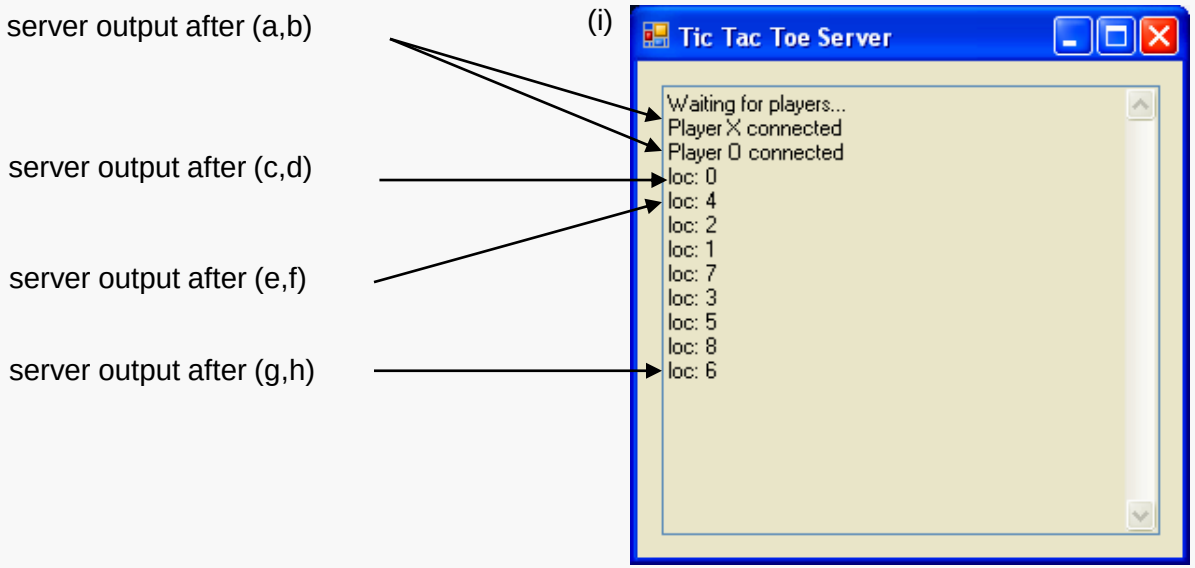


(h)



Outline

The Tie Tac Toe Server's output from the client interactions



`TicTacToeClient.cs`

(13 of 13)

Outline

Square.cs

(1 of 2)

```
// Fig. 23.7: Square.cs
// A Square on the TicTacToe board.
using System.Windows.Forms;

// the representation of a square in a tic-tac-toe grid
public class Square
{
   private Panel panel; // GUI Panel that represents this Square
   private char mark; // player's mark on this Square (if any)
   private int location; // location on the board of this Square

   // constructor
   public Square( Panel newPanel, char newMark, int newLocation )
   {
      panel = newPanel;
      mark = newMark;
      location = newLocation;
   } // end constructor

   // property SquarePanel; the panel which the square represents
   public Panel SquarePanel
   {
      get
      {
         return panel;
      } // end get
   } // end property SquarePanel
```

Represents a square in the Tic-Tac-Toe game

```
28
29    // property Mark; the mark on the square
30    public char Mark
31   {
32    get
33       {
34          return mark;
35       } // end get
36       set
37       {
38          mark = value;
39       } // end set
40    } // end property Mark
41
42    // property Location; the square's location on the board
43    public int Location
44    {
45       get
46       {
47          return location;
48       } // end get
49    } // end property Location
50 }// end class Square
```

Outline

Square.cs

(2 of 2)

23.9 `WebBrowser` Control

`WebBrowser` control

- **Enables applications to incorporate Web browsing capabilities**
- **Generates events as the user interacts with the content displayed in the control**
  - **Applications can respond to these events**
- **Properties `CanGoBack` and `CanGoForward`**
  - **Determine whether the back and forward buttons are enabled**

23.9 WebBrowser Control (Cont.)

`WebBrowser` **methods**

- **Method** `GoBack`
  - **Causes the control to navigate back to the previous page in the navigation history**
- **Method** `GoForward`
  - **Causes the control to navigate forward to the next page in the navigation history**
- **Method** `Stop`
  - **Causes the control to stop loading the current page**
- **Method** `Refresh`
  - **Causes the control to reload the current page**
- **Method** `GoHome`
  - **Causes the control to navigate to the user's home page, as defined under Internet Explorer's settings**
- **Method** `Navigate`
  - **Retrieve the document at the specified URL**

23.9 WebBrowser Control (Cont.)

WebBrowser's Events

- Navigating **event**
  - **Occurs when the** WebBrowser **starts loading a new page**
- StatusTextChanged **event**
  - **Occurs when the** WebBrowser's StatusText **property changes**
- ProgressChanged **event**
  - **Occurs when the** WebBrowser **control's page-loading progress is updated**
- DocumentCompleted **event**
  - **Occurs when the** WebBrowser **finishes loading a document**
- DocumentTitleChanged **event**
  - **Occurs when a new document is loaded in the** WebBrowser **control**

Outline

Browser.cs

(1 of 4)

```
// Fig. 23.8: Browser.cs
// WebBrowser control example.
using System;
using System.Windows.Forms;

public partial class BrowserForm : Form
{
   public BrowserForm()
   {
      InitializeComponent();
   } // end constructor

   // navigate back one page
   private void backButton_Click( object sender, EventArgs e )
   {
      webBrowser.GoBack();
   } // end method backButton_Click

   // navigate forward one page
   private void forwardButton_Click( object sender, EventArgs e )
   {
      webBrowser.GoForward();
   } // end method forwardButton_Click

   // stop loading the current page
   private void stopButton_Click( object sender, EventArgs e )
   {
      webBrowser.Stop();
   } // end method stopButton_Click
```

Navigate back to the previous page in the navigation history

Navigate forward to the next page in the navigation history

Stop loading the current page

Outline

Browser.cs

(2 of 4)

```
30
31    // reload the current page
32    private void reloadButton_Click( object sender, EventArgs e )
33    {
34       webBrowser.Refresh();
35    } // end method reloadButton_Click
36
37    // navigate to the user's home page
38    private void homeButton_Click( object sender, EventArgs e )
39    {
40       webBrowser.GoHome();
41    } // end method homeButton_Click
42
43    // if the user pressed enter, navigate to the specified URL
44    private void navigationTextBox_KeyDown( object sender,
45       KeyEventArgs e )
46    {
47       if ( e.KeyCode == Keys.Enter )
48          webBrowser.Navigate( navigationTextBox.Text );
49    } // end method navigationTextBox_KeyDown
50
51    // enable stopButton while the current page is loading
52    private void webBrowser_Navigating( object sender,
53       WebBrowserNavigatingEventArgs e )
54    {
55       stopButton.Enabled = true;
56    } // end method webBrowser_Navigating
```

Reload the current page

Navigate to the user's homepage

Navigate to the user specified URL

Event occurs when loading a new page

Enable the stop button

```
57
58  // update the status text
59  private void webBrowser_StatusTextChanged( object sender,
60     EventArgs e )
61  {
62     statusTextBox.Text = webBrowser.StatusText;
63  } // end method webBrowser_StatusTextChanged
64
65  // update the ProgressBar for how much of the page has been loaded
66  private void webBrowser_ProgressChanged( object sender,
67     WebBrowserProgressChangedEventArgs e )
68  {
69     pageProgressBar.Value =
70        ( int ) ( ( 100 * e.CurrentProgress ) / e.MaximumProgress );
71  } // end method webBrowser_ProgressChanged
72
73  // update the web browser's controls appropr
74  private void webBrowser_DocumentCompleted( object sender,
75     WebBrowserDocumentCompletedEventArgs e )
76  {
77     // set the text in navigationTextBox to the current page's URL
78     navigationTextBox.Text = webBrowser.Url.ToString();
79
80     // enable or disable backButton and forwardButton
81     backButton.Enabled = webBrowser.CanGoBack;
82     forwardButton.Enabled = webBrowser.CanGoForward;
83
84     // disable stopButton
85     stopButton.Enabled = false;
```

Event occurs when the `StatusText` property changes

Assigns the new contents of the control's `StatusText` property to `statusTextBox`'s `Text` property

(3 of 4)

Event occurs when the page-loading progress is updated

Update `pageProgressBar`'s `Value` to reflect how much of the current document has been loaded

Event occurs when finish loading a document

Update to show the URL of currently loaded page

Determine whether the back and forward buttons should be enabled or disabled

Disable the stop button

```
86
87          // clear the pageProgressBar
88          pageProgressBar.Value = 0;
89       } // end method webBrowser_DocumentCompleted
90
91       // update the title of the Browser
92       private void webBrowser_DocumentTitleChanged( object sender,
93          EventArgs e )
94       {
95          this.Text = webBrowser.DocumentTitle + " - Browser";
96       } // end method webBrowser_DocumentTitleChanged
97 }// end class BrowserForm
```

Indicates that no content is currently being loaded

Browser.cs

Event occurs when a new document is loaded

Sets the `BrowserForm`'s `Text` property to the `WebBrowser`'s current `DocumentTitle`

23.10 .NET Remoting

.NET Remoting is a mechanism for communicating between objects which are not in the same process. It is a generic system for *different applications to communicate with one another*.

.NET objects are exposed to remote processes, thus allowing inter process communication. The applications can be located on the same computer, different computers on the same network, or on computers across separate networks

23.10 .NET Remoting

Microsoft .NET Remoting provides a framework that allows objects to interact with each other across application domains. Remoting was designed in such a way that it hides the most difficult aspects like *managing connections*, *marshaling data*, and *reading* and *writing XML* and *SOAP*. The framework provides a number of services, including *object activation* and *object lifetime* support, as well as *communication channels* which are responsible for transporting messages to and from remote applications

23.10 .NET Remoting

.NET Remoting

- `System.Runtime.Remoting`
- **A distributed computing technology**
- **Allows a program to access objects on another machine over a network**
- **For .NET remoting, both must be written in .NET languages**
- **Marshaling the objects**
  - **A client and a server can communicate via method calls and objects can be transmitted between applications**

Distributed applications should nowadays be developed using the Windows Communications Foundation (WCF)

23.10 .NET Remoting

Remote Objects

- **Any object outside the application domain of the caller application should be considered remote, where the object will be reconstructed. Local objects that cannot be serialized cannot be passed to a different application domain, and are therefore non-remotable.**
- **Any object can be changed into a remote object by deriving it from `MarshalByRefObject`, or by making it serializable either by adding the [`Serializable`] tag or by implementing the `ISerializable` interface**

23.10 .NET Remoting

Remote Objects

- **When a client activates a remote object, it receives a proxy to the remote object. All operations on this proxy are appropriately indirected to enable the Remoting infrastructure to intercept and forward the calls appropriately. In cases where the proxy and remote objects are in different application domains, all *method call parameters on the stack are converted into messages and transported to the remote application domain*, where the *messages are turned back into a stack frame and the method call is invoked*. The same procedure is used for returning results from the method call**

23.10 .NET Remoting

Types of .NET Remotable Objects

There are three types of objects that can be configured to serve as .NET remote objects.

You can choose the type of object depending on the requirement of your application.

23.10 .NET Remoting

Types of .NET Remotable Objects- Single Call

- **Single Call objects service one and only one request coming in. Single Call objects are useful in scenarios where the *objects are required to do a finite amount of work*. Single Call objects are usually not required to store state information, and they cannot hold state information between method calls.**

23.10 .NET Remoting

Types of .NET Remotable Objects- Singleton

- **Singleton objects are those objects that service multiple clients, and hence share data by storing state information between client invocations. They are useful in cases in which *data needs to be shared explicitly between clients*, and also in which the *overhead of creating and maintaining objects is substantial***

23.10 .NET Remoting

Types of .NET Remotable Objects- Client-Activated

- **Client-activated objects (CAO) are server-side objects that are *activated upon request from the client*. When the client submits a request for a server object using a "`new`" operator, an activation request message is sent to the remote application. The server then creates an instance of the requested class, and returns an `ObjRef` back to the client application that invoked it. A `proxy` is then created on the client side using the `ObjRef`. The client's method calls will be executed on the proxy. Client-activated objects can store state information between method calls for its specific client, and not across different client objects. Each invocation of "`new`" returns a proxy to an independent instance of the server type**

23.10 .NET Remoting

Domains

✓**In .NET, when an application is loaded in memory, a *process is created*, and within this process, an application domain is created.**

✓**The application is actually loaded in the application domain. If this application communicates with another application, it has to use `Remoting` because the other application will have its own domain, and across domains, object cannot communicate directly. Different application domains may exist in same process, or they may exist in different processes.**

23.10 .NET Remoting

Proxies

✓**When a call is made between objects in the same Application Domain, only a normal local call is required. A call across Application Domains requires a remote call. In order to facilitate a remote call, a proxy is introduced by the .NET framework at the client side. This proxy is an instance of the `TransparentProxy` class, directly available to the client to communicate with the remote object. Generally, a proxy object is an object that acts in place of some other object. The proxy object *ensures that all calls made on the proxy are forwarded to the correct remote object instance*. In .NET Remoting, the proxy manages the marshaling process and the other tasks required to make cross-boundary calls. The .NET Remoting infrastructure automatically handles the creation and management of proxies**

23.10 .NET Remoting

`RealProxy` and `TransparentProxy`

The .NET Remoting Framework uses two proxy objects to accomplish its work of making a remote call from a client object to a remote server object:

- ✓ a `RealProxy` object and
- ✓ a `TransparentProxy` object.

The `RealProxy` object does the work of actually sending messages to the remote object and receiving response messages from the remote object. The `TransparentProxy` interacts with the client, and does the work of intercepting the remote method call made by the client

23.10 .NET Remoting

Marshaling

Object Marshalling specifies how a remote object is exposed to the client application.

It is the *process of packaging an object access request in one application domain and passing that request to another domain.* The .NET Remoting infrastructure manages the entire marshaling process. There are two methods by which a remote object can be made available to a local client object:

Marshal by value, and

Marshal by reference

23.10 .NET Remoting

Marshalling objects *by value*

Marshaling by value is analogous *to having a copy of the server object at the client*. Objects that are marshaled by value are created on the remote server, serialized into a stream, and transmitted to the client where an exact copy is reconstructed. Once copied to the caller's application domain (by the marshaling process), all method calls and property accesses are executed entirely within that domain.

23.10 .NET Remoting

Marshalling objects *by value*

Marshall by value has several implications- first, the entire remote object is transmitted on the network. Second, some or the entire remote object may have no relevance outside of its local context. For example, the remote object may have a connection to a database, or a handle to a window, or a file handle etc. Third, parts of the remote object may not be serialiazable. In addition, when the client invokes a method on an MBV object, the local machine does the execution, which means that the compiled code (remote class) has to be available to the client. ***Because the object exists entirely in the caller's application domain, no state changes to the object are communicated to the originating application domain, or from the originator back to the caller***

23.10 .NET Remoting

Marshalling objects *by value*

MBV objects are, however, very efficient if they are small, and provide a repeated function that does not consume bandwidth. The entire object exists in the caller's domain, so there is no need to marshal accesses across domain boundaries. Using marshal-by-value objects can increase performance and reduce network traffic, when used for small objects or objects to which you will be making many accesses.

Marshal by value classes must either be marked with the [Serilaizable] attribute in order to use the default serialization, or must implement the ISerializable interface

23.10 .NET Remoting

Marshalling objects *by reference*

Marshaling by reference is analogous to having a pointer to the object. Marshal by reference passes a reference to the remote object back to the client. This reference is an `ObjRef` class that contains all the information required to generate the proxy object that does the communication with the actual remote object. On the network, only parameters and return values are passed. A remote method invocation requires the remote object to call its method on the remote host (server).

Marshal by reference classes must inherit from `System.MarshalByRefObject`

23.10 .NET Remoting (Cont.)

Marshalling

- Marshaling by value
  - Requires that the object be serializable
- Marshaling by reference
  - Requires that the object's class extend class `MarshalByRefObject` of namespace `System`
  - The object itself is not transmitted
    - Instead, two proxy objects are created
      - Transparent proxy
        - Provides all the public services of a remote object
        - Calls the `Invoke` method of the real proxy
      - Real proxy

23.10 .NET Remoting

Channels

The .NET Remoting infrastructure provides a mechanism by which a stream of bytes is sent from one point to the other (client to server etc.). This is achieved via a channel. Strictly speaking, it is a class that implements the `IChannel` interface. There are two pre-defined .NET Remoting channels existing in `System.Runtime.Remoting.Channels`, the `TcpChannel` and the `HttpChannel`. To use the `TcpChannel`, the server must instantiate and register the `TcpServerChannel` class, and the client, the `TcpClientChannel` class.

23.10 .NET Remoting (Cont.)

Channels

- The client and the server are able to communicate with one another through channels
- The client and the server each create a channel
  - Both channels must use the same protocol to communicate with one another
- HTTP channel
  - Firewall usually permit HTTP connections by default
- TCP channel
  - Better performance than an HTTP channel
  - Firewalls normally block unfamiliar TCP connections
- `System.Runtime.Remoting.Channels`
- `System.Runtime.Remoting.Channels.Http`

23.10 .NET Remoting

Channels

At least one channel must be registered with the remoting framework before a remote object can be called. Channels must be registered before objects are registered.

Channels are registered per application domain. There can be multiple application domains in a single process. When a process dies, all channels that it registers are automatically destroyed.

23.10 .NET Remoting

Channels

It is illegal to register the same channel that listens on the same port more than once. Even though channels are registered per application domain, different application domains on the same machine cannot register the same channel listening on the same port. You can register the same channel listening on two different ports for an application domain.

Clients can communicate with a remote object using any registered channel. The remoting framework ensures that the remote object is connected to the right channel when a client attempts to connect to it. The client is responsible for calling the RegisterChannel on the ChannelService class before attempting to communicate with a remote object

23.10 .Summary

`interface IArea`

- **Declares the methods(services) offered by the remote object**

`class CircleClientForm`

- **Consumes services implemented as methods of a remote object**

Remote `class` CircleServer

- **Creates a `HTTP` channel and registers the remote object as a `Singleton`**

Remote class registered as `Singleton`

- **One remote object will be created when the first client requests that remote class**
  - **The remote object will service all clients**

Remote object represented by `class` Circle

- **Implements `interface IArea` and `MarshalByRefObject`**

23.10 .NET Remoting (Cont.)

```
public interface IArea
{// define services offered by the remote object
   double Area();
   double R { get; set; }
  // compute Circle area
} // end interface IArea
```

23.10 .NET Remoting (Cont.)

```
public class Circle : MarshalByRefObject, IArea
{


  public Circle()
  {
      this.R = 0;


  } // end constructor
  // IArea methods - services offered by the remote object
  public double Area(  )
  {
      return Math.PI * R * R;
  } // end method Area()
  public double R { get; set; } // sample property usage

} // end class Circle
```

23.10 .NET Remoting (Cont.)

```
class CircleServer
{
  static void Main( string[] args )
  {
     // establish HTTP channel
     HttpChannel channel = new HttpChannel( 50001 );
     ChannelServices.RegisterChannel( channel, false );

     // register ReportInfo class
     RemotingConfiguration.RegisterWellKnownServiceType(
                                                 typeof( Circle ), "Circle",
                                             WellKnownObjectMode.Singleton );

     Console.WriteLine( "Press Enter to terminate server." );
     Console.ReadLine();
  } // end Main
} // end class CircleServer
```

23.10 .NET Remoting (Cont.)

```
public partial class CircleClientForm : Form
{
   private IArea info;
  public CircleClientForm()
  {
     InitializeComponent();
  } // end constructor

  // consume IArea services
  private void CircleClientForm_Load( object sender, EventArgs e )
  {
     // setup HTTP channel, does not need to provide a port number
     HttpChannel channel = new HttpChannel();
     ChannelServices.RegisterChannel( channel, false );

     // obtain a proxy for an object that implements interface IArea
        info = ( IArea ) RemotingServices.Connect(
                                typeof( IArea ), "http://localhost:50001/Circle" );


  }

  private void btnCompute_Click(object sender, EventArgs e)
  {
      info.R = Convert.ToDouble(txtInput.Text);
      // retrieve info.Area()
      txtOutput.Text = Convert.ToDouble(info.Area()).ToString();
  }

} // end class CircleClientForm
```

23.10 .NET Remoting

.NET Remoting

- `System.Runtime.Remoting`
- **A distributed computing technology**
- **Allows a program to access objects on another machine over a network**
- **For .NET remoting, both must be written in .NET languages**
- **Marshaling the objects**
  - **A client and a server can communicate via method calls and objects can be transmitted between applications**

Outline

```
// Fig. 23.9: CityWeather.cs
// Class representing the weather information for one city.
using System;

namespace Weather
{
   [ Serializable ]
   public class CityWeather : IComparable
   {
      private string cityName;
      private string description;
      private string temperature;

      public CityWeather( string city, string information,
          string degrees )
      {
         cityName = city;
         description = information;
         temperature = degrees;
      } // end constructor

      // read-only property that gets city's name
      public string CityName
      {
         get
         {
            return cityName;
         } // end get
      } // end property CityName
```

The class will be published in the `Weather.dll` class library file

Indicates that an object of this class can be marshaled by value

(1 of 3)

Implements interface `IComparable` so that the `ArrayList` of `CityWeather` objects can be sorted alphabetically

```
30
31      // read-only property that gets city's weather description
32      public string Description
33      {
34         get
35         {
36            return description;
37         } // end get
38      } // end property Description
39
40      // read-only property that gets city's temperature
41      public string Temperature
42      {
43         get
44         {
45            return temperature;
46         } // end get
47      } // end property Temperature
```

Outline

CityWeather.cs

(2 of 3)

Outline

```
48
49      // implementation of CompareTo method for alphabetizing
50      public int CompareTo( object other )
51      {
52         return string.Compare(
53            CityName, ( ( CityWeather ) other ).CityName );
54      } // end method Compare
55
56      // return string representation of this CityWeather object
57      // (used to display the weather report on the server console)
58      public override string ToString()
59      {
60         return cityName + " | " + temperature + " | " + description;
61      } // end method ToString
62   } // end class CityWeather
63 }// end namespace Weather
```

Must declare `CompareTo` since implemented `IComparable`

(3 of 3)

Update and override `ToString`

Outline

Report.cs

```
// Fig. 23.10: Report.cs
// Interface that defines a property for getting
// the information in a weather report.
using System;
using System.Collections;

namespace Weather
{
   public interface Report
   {
      ArrayList Reports
      {
         get;
      } // end property Reports
   } // end interface Report
}// end namespace Weather
```

Interface will also be included in the `Weather.dll` class library

Interface include read-only property `Reports` of type `ArrayList`

Outline

ReportInfo.cs

```
// Fig. 23.11: ReportInfo.cs
// Class that implements interface Report, retrieves
// and returns data on weather
using System;
using System.Collections;
using System.IO;
using System.Net;
using Weather;

public class ReportInfo : MarshalByRefObject, Report
{
   private ArrayList cityList; // cities, temperatures, descriptions

   public ReportInfo()
   {
      cityList = new ArrayList();

      // create WebClient to get access to Web page
      WebClient myClient = new WebClient();

      // get StreamReader for response so we can read page
      StreamReader input = new StreamReader( myClient.OpenRead(
         "http://iwin.nws.noaa.gov/iwin/us/traveler.html" ) );

      string separator1 = "TAV12"; // indicates first batch of cities
      string separator2 = "TAV13"; // indicates second batch of cities

      // locate separator1 in Web page
      while ( !input.ReadLine().StartsWith( separator1 ) ); // do nothing
      ReadCities( input ); // read the first batch of cities
```

Using `Weather.dll` class library

Extends `MarshalByRefObject` and implements `Report`

Declare data structure to hold information

Interact with a data source specified by a URL

Return a stream that the program can use to read data containing the weather information from the URL

Store separators

Read the HTML markup one line at a time

Call `ReadCities` to read the first batch of cities

```
31
32      // locate separator2 in Web page
33      while ( !input.ReadLine().StartsWith( separator2 ) ); // do nothing
34      ReadCities( input ); // read the second batch of cities
35
36      cityList.Sort(); // sort list of cities by alphabetical order
37      input.Close(); // close StreamReader to NWS server
38
39      // display the data on the server side
40      Console.WriteLine( "Data from NWS Web site:" );
41
42      foreach ( CityWeather city in cityList )
43      {
44         Console.WriteLine( city );
45      } // end foreach
46   } // end constructor
47
48   // utility method that reads a batch of cities
49   private void ReadCities( StreamReader input )
50   {
51      // day format and night format
52      string dayFormat =
53         "CITY            WEA     HI/LO   WEA     HI/LO";
54      string nightFormat =
55         "CITY            WEA     LO/HI   WEA     LO/HI";
56      string inputLine = "";
```

Read the HTML markup one line at a time

Call `ReadCities` to read the second batch of cities

ReportInfo.cs

Sort the `CityWeather` objects in alphabetical order by city name

The header for the daytime/nighttime information

```
57
58      // locate header that begins weather information
59      do
60      {
61         inputLine = input.ReadLine();
62      } while ( !inputLine.Equals( dayFormat ) &&
63         !inputLine.Equals( nightFormat ) );
64
65      inputLine = input.ReadLine(); // get first city's data
66
67      // while there are more cities to read
68      while ( inputLine.Length > 28 )
69      {
70         // create CityWeather object for city
71         CityWeather weather = new CityWeather(
72            inputLine.Substring( 0, 16 ),
73            inputLine.Substring( 16, 7 ),
74            inputLine.Substring( 23, 7 ) );
75
76         cityList.Add( weather ); // add to ArrayList
77         inputLine = input.ReadLine(); // get next city's data
78      } // end while
79   } // end method ReadCities
```

ReportInfo.cs

Continue to read the page one line at a time until it finds the header line

Reads the next line from the Web page, which is the first line containing temperature information

Create a new `CityWeather` object to represent the current city

Store the `CityWeather` object in the `ArrayList`

Continue by reading the next city's information

```
80
81     // property for getting the cities' weather reports
82     public ArrayList Reports
83     {
84        get
85        {
86           return cityList;
87        } // end get
88     } // end property Reports
89 }// end class ReportInfo
```

Declare read-only property `Reports` to satisfy the `Report` interface

Outline

`ReportInfo.cs`

(4 of 4)

23.10 .Summary

Class `WebClient`

- **Interact with a data source that is specified by a URL**
- **Method** `OpenRead`
  - **Returns a** `Stream` **that the program can use to read data from the specified URL**

Remote class registered as `Singleton`

- **One remote object will be created when the first client requests that remote class**
  - **The remote object will service all clients**

Remote class registered as `SingleCall`

- **One remote object is created for each individual remote method call to the remote class**

Outline

WeatherServer.cs

```
// Fig. 23.12: WeatherServer.cs
// Server application that uses .NET remoting to send
// weather report information to a client
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using Weather;

class WeatherServer
{
   static void Main( string[] args )
   {
      // establish HTTP channel
      HttpChannel channel = new HttpChannel( 50000 );
      ChannelServices.RegisterChannel( channel, false );

      // register ReportInfo class
      RemotingConfiguration.RegisterWellKnownServiceType(
         typeof( ReportInfo ), "Report",
         WellKnownObjectMode.Singleton );

      Console.WriteLine( "Press Enter to terminate server." );
      Console.ReadLine();
   } // end Main
} // end class WeatherServer
```

Using namespaces for remoting capabilities

Register an HTTP channel on the current machine at port 50000

Indicates that we do not wish to enable security

Register the `ReportInfo` class type at the “Report” URI as a `Singleton` remote class

Outline

WeatherClient.cs

(1 of 5)

```
// Fig. 23.13: WeatherClient.cs
// Client that uses .NET remoting to retrieve a weather report.
using System;
using System.Collections;
using System.Drawing;
using System.Windows.Forms;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using Weather;

public partial class WeatherClientForm : Form
{
   public WeatherClientForm()
   {
      InitializeComponent();
   } // end constructor

   // retrieve weather data
   private void WeatherClientForm_Load( object sender, EventArgs e )
   {
      // setup HTTP channel, does not need to provide a port number
      HttpChannel channel = new HttpChannel();
      ChannelServices.RegisterChannel( channel, false );

      // obtain a proxy for an object that implements interface Report
      Report info = ( Report ) RemotingServices.Connect(
         typeof( Report ), "http://localhost:50000/Report" );
```

Using namespaces for remoting capabilities

Create a HTTP channel without specifying a port number

Register the channel on the client computer

Connects to the server and returns a reference to the proxy for the `Report` object

```
29
30      // retrieve an ArrayList of CityWeather objects
31      ArrayList cities = info.Reports;
32
33      // create array and populate it with every Label
34      Label[] cityLabels = new Label[ 43 ];
35      int labelCounter = 0;
36
37      foreach ( Control control in displayPanel.Controls )
38      {
39         if ( control is Label )
40         {
41            cityLabels[ labelCounter ] = ( Label ) control;
42            ++labelCounter; // increment Label counter
43         } // end if
44      } // end foreach
```

Retrieve the `ArrayList` of `CityWeather` objects generated by the `ReportInfo` constructor

WeatherClient.cs

Create and place an array of `Label` so they can be access programmatically to display weather information

Outline

```
45
46      // create Hashtable and populate with all weather conditions
47      Hashtable weather = new Hashtable();
48      weather.Add( "SUNNY", "sunny" );
49      weather.Add( "PTCLDY", "pcloudy" );
50      weather.Add( "CLOUDY", "mcloudy" );
51      weather.Add( "MOCLDY", "mcloudy" );
52      weather.Add( "TSTRMS", "rain" );
53      weather.Add( "RAIN", "rain" );
54      weather.Add( "SNOW", "snow" );
55      weather.Add( "VRYHOT", "vryhot" );
56      weather.Add( "FAIR", "fair" );
57      weather.Add( "RNSNOW", "rnsnow" );
58      weather.Add( "SHWRS", "showers" );
59      weather.Add( "WINDY", "windy" );
60      weather.Add( "NOINFO", "noinfo" );
61      weather.Add( "MISG", "noinfo" );
62      weather.Add( "DRZL", "rain" );
63      weather.Add( "HAZE", "noinfo" );
64      weather.Add( "SMOKE", "mcloudy" );
65      weather.Add( "SNOWSHWRS", "snow" );
66      weather.Add( "FLRRYS", "snow" );
67      weather.Add( "FOG", "noinfo" );
68
69      // create the font for the text output
70      Font font = new Font( "Courier New", 8, FontStyle.Bold );
```

Store pairs of weather conditions and the names for images associate with those conditions

weatherClient.cs

(3 of 5)

```
71
72      // for every city
73      for ( int i = 0; i < cities.Count; i++ )
74      {
75         // use array cityLabels to find the next Label
76         Label currentCity = cityLabels[ i ];
77
78         // use ArrayList cities to find the next CityWea
79         CityWeather city = ( CityWeather ) cities[ i ];
80
81         // set current Label's image to image
82         // corresponding to the city's weather condition -
83         // find correct image name in Hashtable weather
84         currentCity.Image = new Bitmap( @"images\" +
85            weather[ city.Description.Trim() ] + ".png" );
86         currentCity.Font = font; // set font of Label
87         currentCity.ForeColor = Color.White; // set text color of Label
88
89         // set Label's text to city name and temperature
90         currentCity.Text = "\r\n " + city.CityName + city.Temperature;
91      } // end for
92   } // end method WeatherClientForm_Load
93 }// end class WeatherClientForm
```

Retrieve the `Label` that will display the weather information for the next city

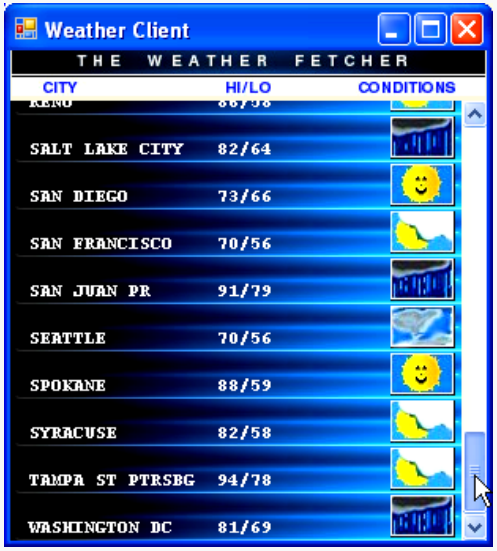
Uses `ArrayList` cities to retrieve the `CityWeather` object that contains the weather information for the city

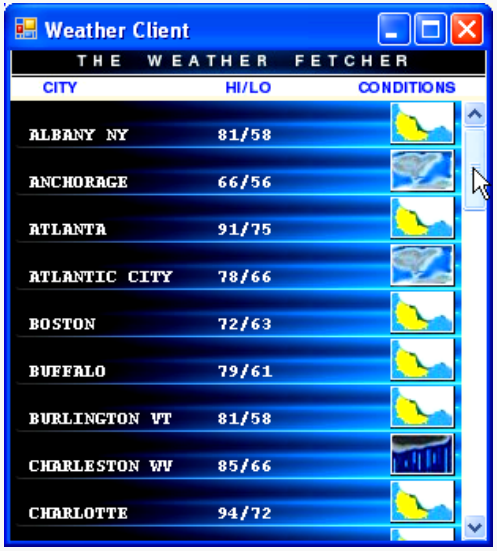
(4 of 5)

Set the `Label`'s image to the PNG image from the `Hashtable`

Display the city's name and high/low temperatures

Outline





`WeatherClient.cs`

(5 of 5)