The human subconscious is a fascinating place—malleable, permeable, fallible.

—Harvey, Farscape

The introduction of the .NET Framework made the last year or so an extremely exciting time for software developers. Not only does .NET provide an entirely new platform for creating software, it also introduces an extremely rich (and quite large) set of class libraries for building managed applications, as well as a new type-safe object-oriented programming language known as C#.

The .NET Compact Framework is a version of .NET specifically designed for small form factor devices, such as Pocket PC. The class library provided with the Compact Framework is extremely similar to its desktop counterpart, except that certain functionality has been “slimmed down” (or entirely eliminated) to better support the limited memory, storage space, and performance of a mobile device.

Because covering the entire Compact Framework would be a book in itself, this chapter provides you with information about using some of the .NET classes that are of particular interest to Pocket PC application developers. We first take a look at performing Winsock communications (see Chapter 1) between devices using the Sockets class library that is provided by the Compact Framework. This is followed by an explanation of how to write applications that request data using standard Internet protocols, such as HTTP (see Chapter 2).

This chapter also describes how you can consume Web Services, probably one of the most intriguing concepts for a mobile developer. A Web Service is a standardized way to access distributed program logic by using “off-the-shelf” Internet protocols. For example, suppose you had
an application running on a Pocket PC device that kept an itinerary of your travel plans. You could use one Web Service to get information about flight delays, another to get the weather report at your destination, and another to pull gate information, tying all of the information together within your application. What makes Web Services unique is that any communications with the server hosting the Web Service are done through a standardized XML format. By using Web Services, you can easily create robust mobile applications that pull data from a variety of sources on the Internet.

Finally, we’ll take a look at using some of the APIs that are native to the Pocket PC, such as the Connection Manager (see Chapter 7) and SMS Messaging (see Chapter 8), from applications written in C#.

Unlike writing standard C++ applications for a Pocket PC device using Embedded Visual C++ 3.0, you use Visual Studio 2003.NET for developing C# and VB.NET applications. At this time, you cannot use C++ to develop .NET applications for the Compact Framework.

Networking with the Compact Framework

When developing applications that communicate over a network using .NET, most of the classes that you will need to familiarize yourself with are part of the System.Net namespace. It contains classes for handling Internet communications with objects that support proxy servers, IP addresses, DNS name resolution, network data streams, and specific classes for handling pluggable protocols such as the Hypertext Transfer Protocol (HTTP).

Table 12.1 describes the objects contained in the System.Net namespace.
### Table 12.1 The System.Net Namespace

<table>
<thead>
<tr>
<th>Name</th>
<th>Object Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AuthenticationManager</td>
<td>Class</td>
<td>Manages authentication models</td>
</tr>
<tr>
<td>Authorization</td>
<td>Class</td>
<td>Handles authorization messages to a sever</td>
</tr>
<tr>
<td>Dns</td>
<td>Class</td>
<td>Handles domain name resolution</td>
</tr>
<tr>
<td>EndPoint</td>
<td>Class</td>
<td>Abstract class for identifying a network address</td>
</tr>
<tr>
<td>GlobalProxySelection</td>
<td>Class</td>
<td>Handles the default proxy for HTTP requests</td>
</tr>
<tr>
<td>HttpContinueDelegate</td>
<td>Delegate</td>
<td>Callback used for HTTP requests</td>
</tr>
<tr>
<td>HttpStatusCode</td>
<td>Enumeration</td>
<td>Status codes used for HTTP requests</td>
</tr>
<tr>
<td>HttpVersion</td>
<td>Class</td>
<td>Handles version numbers supported by HTTP requests</td>
</tr>
<tr>
<td>HttpWebRequest</td>
<td>Class</td>
<td>Handles an HTTP request</td>
</tr>
<tr>
<td>HttpWebResponse</td>
<td>Class</td>
<td>Handles the response of an HTTP request</td>
</tr>
<tr>
<td>IAuthenticationModule</td>
<td>Interface</td>
<td>Interface used for Web authentication</td>
</tr>
<tr>
<td>ICertificatePolicy</td>
<td>Interface</td>
<td>Interface that validates a server’s certificate</td>
</tr>
<tr>
<td>ICredentials</td>
<td>Interface</td>
<td>Interface for handling Web client authentication</td>
</tr>
<tr>
<td>IPAddress</td>
<td>Class</td>
<td>Handles IP addressing</td>
</tr>
<tr>
<td>IPEndPoint</td>
<td>Class</td>
<td>Handles an IP address and port number</td>
</tr>
<tr>
<td>IPEndPoint</td>
<td>Class</td>
<td>Handles Internet host address information</td>
</tr>
<tr>
<td>IPHostEntry</td>
<td>Class</td>
<td>Handles an infrared connection to another device</td>
</tr>
<tr>
<td>IrDAEndPoint</td>
<td>Class</td>
<td></td>
</tr>
<tr>
<td>IWebProxy</td>
<td>Interface</td>
<td>Interface to handle a proxy request</td>
</tr>
<tr>
<td>IWebRequestCreate</td>
<td>Interface</td>
<td>Interface to handle new HttpRequest instances</td>
</tr>
</tbody>
</table>

(continued)
TCP/IP Addresses

In Chapter 1, you learned about the Internet Protocol version 4 (or IPv4) address scheme on Pocket PC. You may remember that an IPv4 address is used by a device to specify its unique host and subnet address, which it uses to communicate over a TCP/IP network. All of the methods and properties that are needed to manage an Internet address within the Compact Framework are handled by the System.Net.IPEndPoint class.

The IPAddress constructor is defined as follows:

```java
public IPAddress(long newAddress);
```

The only parameter needed is the 32-bit value of the IP address. The class also contains the methods and properties described in Table 12.2.
One of the most useful methods in the IPAddress class is the Parse() method. You can use this to easily construct a new IPAddress object using the standard dotted-notation Internet address, as shown in the following example:

```csharp
System.Net.IPAddress localIPAddress =
    System.Net.IPAddress.Parse("127.0.0.1");
```

Although the IPAddress class by itself is useful for managing an Internet address, most of the networking functions in the Compact Framework use the System.Net.IPEndPoint class to specify another machine on the network. An IPEndPoint not only specifies the IP address of the remote connection, but also contains information about the port that will be used to connect with the service running on the remote device (for more information about Internet ports, see Chapter 1).

There are two ways to construct a new IPEndPoint class. The first method takes the 32-bit value of the IP address and a port:

```csharp
public IPEndPoint(long address, int port);
```
You can also create a new `IPEndPoint` by passing in a previously created `IPAddress` object:

```csharp
public IPEndPoint(IPAddress address, int port);
```

The following code shows how you can create an `IPEndPoint` that represents a connection to the local machine on port 80:

```csharp
System.Net.IPAddress localIPAddress =
    System.Net.IPAddress.Parse("127.0.0.1");

System.Net.IPEndPoint localIPEndPoint = new
    System.Net.IPEndPoint(localIPAddress, 80);
```

The `IPEndPoint` class consists of the methods and properties described in Table 12.3.

**Table 12.3** `IPEndPoint` Class Methods and Properties

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create()</td>
<td>Creates an <code>IPEndPoint</code> based on an IP address and port</td>
</tr>
<tr>
<td>Serialize()</td>
<td>Serializes <code>IPEndPoint</code> information into a <code>SocketAddress</code> instance</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
<th>Get/Set/Read-Only</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Get/set</td>
<td>Value of the IP address</td>
</tr>
<tr>
<td>AddressFamily</td>
<td>Get</td>
<td>Gets the address family for the IP address</td>
</tr>
<tr>
<td>Port</td>
<td>Get/set</td>
<td>Value of the port</td>
</tr>
<tr>
<td>MaxPort</td>
<td>Read-only field</td>
<td>Specifies the maximum value for the port</td>
</tr>
<tr>
<td>MinPort</td>
<td>Read-only field</td>
<td>Specifies the minimum value for the port</td>
</tr>
</tbody>
</table>

**Name Resolution**

The resolution of a domain name (such as `www.furrygoat.com`) or IP address is handled by the `System.Net.Dns` class. It contains the methods described in Table 12.4.
After the DNS resolution process has completed, information about the domain is stored in a new instance of the `System.Net.IPHostEntry` class. The class has the properties described in Table 12.5.

**Table 12.4 Dns Class Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeginGetHostByName()</td>
<td>Starts an asynchronous <code>GetHostByName()</code> request</td>
</tr>
<tr>
<td>BeginResolve()</td>
<td>Starts an asynchronous <code>Resolve()</code> request</td>
</tr>
<tr>
<td>EndGetHostByName()</td>
<td>Ends an asynchronous <code>GetHostByName()</code> request</td>
</tr>
<tr>
<td>EndResolve()</td>
<td>Ends an asynchronous <code>Resolve()</code> request</td>
</tr>
<tr>
<td>GetHostByAddress()</td>
<td>Gets host information based on the IP address</td>
</tr>
<tr>
<td>GetHostByName()</td>
<td>Gets host information based on the name</td>
</tr>
<tr>
<td>Resolve()</td>
<td>Resolves a host name or IP address to an <code>IPHostEntry</code> class</td>
</tr>
</tbody>
</table>

After the DNS resolution process has completed, information about the domain is stored in a new instance of the `System.Net.IPHostEntry` class. The class has the properties described in Table 12.5.

**Table 12.5 IPHostEntry Class Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Get/Set/Read-Only</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddressList</td>
<td>Get/set</td>
<td>Gets or sets a list of <code>IPAddress</code> objects associated with the host</td>
</tr>
<tr>
<td>Aliases</td>
<td>Get/set</td>
<td>Gets or sets a list of aliases associated with the host</td>
</tr>
<tr>
<td>HostName</td>
<td>Get/set</td>
<td>Gets or sets the DNS host name</td>
</tr>
</tbody>
</table>

The following code shows how you can create an `IPEndPoint` that is associated with the Microsoft Web Server by using the `System.Net.Dns` class to first resolve the IP address:

```csharp
// Resolve the MS Web Server IP address
System.Net.IPHostEntry microsoftHost =

// Copy the resolved IP address to a string
String msIP = microsoftHost.AddressList[0].ToString();

// Create the endpoint
System.Net.IPEndPoint microsoftEndPoint = new 
    System.Net.IPEndPoint(microsoftHost.AddressList[0], 80);
```
Winsock and .NET

The System.Net.Sockets namespace provides all of the classes that are needed to communicate over the Winsock interface (see Chapter 1 for more information about general Winsock programming) when using the Compact Framework. The namespace provides the classes and enumerations described in Table 12.6.

Table 12.6 The System.Net.Sockets Namespace

<table>
<thead>
<tr>
<th>Name</th>
<th>Object Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddressFamily</td>
<td>Enumeration</td>
<td>Address scheme for a Socket class</td>
</tr>
<tr>
<td>IrDACharacterSet</td>
<td>Enumeration</td>
<td>Character sets supported for infrared transfers</td>
</tr>
<tr>
<td>IrDAClient</td>
<td>Class</td>
<td>Handles the client in an infrared transfer</td>
</tr>
<tr>
<td>IrDADeviceInfo</td>
<td>Class</td>
<td>Provides information about infrared connections and servers</td>
</tr>
<tr>
<td>IrDAHints</td>
<td>Enumeration</td>
<td>Infrared device types</td>
</tr>
<tr>
<td>IrDALevel</td>
<td>Class</td>
<td>Handles the server in an infrared transfer</td>
</tr>
<tr>
<td>LingerOption</td>
<td>Class</td>
<td>Handles the socket linger options</td>
</tr>
<tr>
<td>multicastOption</td>
<td>Class</td>
<td>Handles multicast address groups</td>
</tr>
<tr>
<td>NetworkStream</td>
<td>Class</td>
<td>Handles a stream over a network connection</td>
</tr>
<tr>
<td>ProtocolFamily</td>
<td>Enumeration</td>
<td>Socket protocol types that are available</td>
</tr>
<tr>
<td>ProtocolType</td>
<td>Enumeration</td>
<td>Socket protocols</td>
</tr>
<tr>
<td>SelectMode</td>
<td>Enumeration</td>
<td>Socket polling modes</td>
</tr>
<tr>
<td>Socket</td>
<td>Class</td>
<td>Class to handle socket communications</td>
</tr>
<tr>
<td>SocketException</td>
<td>Class</td>
<td>Exception that is used when an error occurs in a Socket class</td>
</tr>
<tr>
<td>SocketFlags</td>
<td>Enumeration</td>
<td>Socket constants</td>
</tr>
<tr>
<td>SocketOptionLevel</td>
<td>Enumeration</td>
<td>Socket level option constant values</td>
</tr>
<tr>
<td>SocketOptionName</td>
<td>Enumeration</td>
<td>Socket names option constant values</td>
</tr>
<tr>
<td>SocketShutdown</td>
<td>Enumeration</td>
<td>Socket shutdown constants</td>
</tr>
<tr>
<td>SocketType</td>
<td>Enumeration</td>
<td>Type of socket</td>
</tr>
<tr>
<td>TcpClient</td>
<td>Class</td>
<td>Class to handle TCP socket connections to a server</td>
</tr>
<tr>
<td>TcpListener</td>
<td>Class</td>
<td>Class to handle TCP socket connections as a server</td>
</tr>
<tr>
<td>UdpClient</td>
<td>Class</td>
<td>Class to handle UDP socket connections for both client and server</td>
</tr>
</tbody>
</table>
The namespace provides four classes that you will use primarily when working with Winsock connections:

1. The `System.Net.Sockets.Socket` class is essentially a full wrapper around a traditional SOCKET handle. It provides all of the functionality for both connectionless and connection-based TCP and UDP communications.
2. The `System.Net.Sockets.TcpClient` class provides all of the methods and properties for the client side of a TCP connection to a server.
3. The `System.Net.Sockets.TcpListener` class provides all of the methods and properties for the server side of a TCP connection that will listen for incoming connections on a specific port.

**The Generic Socket Class**

The `System.Net.Sockets.Socket` class is used to perform basic Winsock functionality in a manner similar to using a standard SOCKET handle. To create a new `Socket` object, you use the following constructor:

```csharp
public Socket(AddressFamily addressFamily, SocketType socketType, ProtocolType protocolType);
```

All of the parameters that you use are standard enumerations that are part of the `System.Net.Sockets` namespace. The first parameter, `addressFamily`, should specify the addressing scheme for the socket, such as `AddressFamily.InterNetwork` for an IPv4 socket. This is followed by the type of socket you are creating, which is followed by the protocol that the socket should use.

The following example creates a standard IPv4 socket for communicating over a TCP connection using the IP protocol:

```csharp
using System;
using System.Data;
using System.Net.Sockets;

namespace PocketPCNetworkProgramming {
    class SocketTestClass {
```
static void Main(string[] args) {
    // Create a new socket
        AddressFamily.InterNetwork,
        SocketType.Stream,
        ProtocolType.IP);

    // Do something with the new socket
}

The `System.Net.Sockets.Socket` class supports the methods and properties described in Table 12.7.

### Table 12.7 Socket Class Methods and Properties

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Accept()</code></td>
<td>Creates a new <code>System.Net.Sockets.Socket</code> for the incoming connection</td>
</tr>
<tr>
<td><code>BeginAccept()</code></td>
<td>Begins asynchronous <code>Accept()</code> operation</td>
</tr>
<tr>
<td><code>BeginConnect()</code></td>
<td>Begins asynchronous <code>Connect()</code> operation</td>
</tr>
<tr>
<td><code>BeginReceive()</code></td>
<td>Begins asynchronous <code>Receive()</code> operation</td>
</tr>
<tr>
<td><code>BeginReceiveFrom()</code></td>
<td>Begins asynchronous <code>Receive()</code> operation from a specific remote <code>EndPoint</code></td>
</tr>
<tr>
<td><code>BeginSend()</code></td>
<td>Begins asynchronous <code>Send()</code> operation</td>
</tr>
<tr>
<td><code>BeginSendTo()</code></td>
<td>Begins asynchronous <code>Send()</code> operation to a specific remote <code>EndPoint</code></td>
</tr>
<tr>
<td><code>Bind()</code></td>
<td>Associates the socket with a local <code>EndPoint</code></td>
</tr>
<tr>
<td><code>Close()</code></td>
<td>Closes the socket</td>
</tr>
<tr>
<td><code>Connect()</code></td>
<td>Establishes a connection with another host</td>
</tr>
<tr>
<td><code>EndAccept()</code></td>
<td>Asynchronously accepts an incoming connection</td>
</tr>
<tr>
<td><code>EndConnect()</code></td>
<td>Ends asynchronous <code>Connect()</code> operation</td>
</tr>
<tr>
<td><code>EndReceive()</code></td>
<td>Ends asynchronous <code>Receive()</code> operation</td>
</tr>
<tr>
<td><code>EndReceiveFrom()</code></td>
<td>Ends asynchronous <code>Receive()</code> operation from a specific remote <code>EndPoint</code></td>
</tr>
<tr>
<td><code>EndSend()</code></td>
<td>Ends asynchronous <code>Send()</code> operation</td>
</tr>
<tr>
<td><code>EndSendTo()</code></td>
<td>Ends asynchronous <code>Send()</code> operation to a specific remote <code>EndPoint</code></td>
</tr>
<tr>
<td><code>GetSocketOption()</code></td>
<td>Returns the value of the socket options</td>
</tr>
<tr>
<td><code>IOControl()</code></td>
<td>Sets low-level socket options</td>
</tr>
<tr>
<td><code>Listen()</code></td>
<td>Listens for an incoming socket connection</td>
</tr>
</tbody>
</table>
Once you have created your `Socket` class, communicating over the Internet is relatively straightforward. The class supports methods such as `Send()` and `Receive()`, which are almost identical to the standard Winsock functions:

```csharp
// Create a new socket
    Socket(AddressFamily.InterNetwork, SocketType.Stream,
    ProtocolType.IP);

// Make a request from a Web server
// Resolve the IP address for the server, and get the
// IPEndPoint for it on port 80
```
System.Net.IPHostEntry webServerHost =
System.Net.IPEndPoint webServerEndPt = new
    System.Net.IPEndPoint(webServerHost.AddressList[0], 80);

// Set up the HTTP request string to get the main index page
byte[] httpRequestBytes =
    System.Text.Encoding.ASCII.GetBytes("GET /
    HTTP/1.0\r\n\r\n");

// Connect the socket to the server
webSocket.Connect(webServerEndPt);

// Send the request synchronously
int bytesSent = webSocket.Send(httpRequestBytes,
    httpRequestBytes.Length, SocketFlags.None);

// Get the response from the request. We will continue to request
// 4096 bytes from the response stream and concat the string into
// the strResponse variable
byte[] httpResponseBytes = new byte[4096];
int bytesRecv = webSocket.Receive(httpResponseBytes,
    httpResponseBytes.Length, SocketFlags.None);
strResponse = System.Text.Encoding.ASCII.GetString
    (httpResponseBytes, 0, bytesRecv);

while(bytesRecv > 0) {
    bytesRecv = webSocket.Receive(httpResponseBytes,
        httpResponseBytes.Length, SocketFlags.None);

    strResponse = strResponse +
        System.Text.Encoding.ASCII.GetString(
            httpResponseBytes, 0, bytesRecv);
}

// At this point, the strResponse string has the Web page.
// Do something with it
// ...
// Clean up the socket
webSocket.Shutdown(SocketShutdown.Both);
webSocket.Close();

Although using the Socket class provides you with a robust set of methods to handle almost any type of connection, you are more likely to use one of the more specific connection classes, such as TcpClient or TcpListener, to handle your protocol-specific network communications.

**TCP Connections**

As described in Chapter 1, a TCP (or streaming) socket provides you with an error-free data pipe (between a client and server) that is used to send and receive data over a communications session. The format of the data sent over the connection is typically up to you, but several well-known Internet protocols, such as HTTP and FTP, use this type of connection.

The .NET Compact Framework provides you with two separate classes that can be used to handle TCP communications. The System.Net.Sockets.TcpListener class is used to create a socket that can accept an incoming connection request. This is also known as a server.

To create a TCP client, you use the System.Net.Sockets.TcpClient class. The methods provide functionality to connect to a server that is listening on a specific port.

**TCP Servers**

To create a new TcpListener object, you can use one of the following constructors:

```csharp
public TcpListener(int port);
public TcpListener(IPAddress localaddr, int port);
public TcpListener(IPEndPoint localEP);
```

All three constructors basically do the same thing. The first one needs only the port number on which you want the object to listen. The second requires an IPAddress class that represents the local IP address of the device, and is followed by the port. The final constructor takes an IPEndPoint class, which should represent the local IP address and port on which to listen.
The following example shows how you can use each one of the constructors to initialize a new TcpListener class:

```csharp
// Method 1 - Listen on the local IP address, port 80.
System.Net.Sockets.TcpListener tcpServerSocket = new TcpListener(80);

// Method 2 - Listen on the local IP address, port 80.
System.Net.IPAddress localIPAddr = System.Net.IPAddress.Parse("127.0.0.1");
System.Net.Sockets.TcpListener tcpServerSocket2 = new TcpListener(localIPAddr, 80);

// Method 3 - Listen on the local IP address by creating an endpoint
System.Net.IPEndPoint localIpEndPoint = new System.Net.IPEndPoint(localIPAddr, 80);
System.Net.Sockets.TcpListener tcpServerSocket3 = new TcpListener(localIpEndPoint);
```

The TcpListener object provides the methods and property described in Table 12.8.

### Table 12.8 TCPEndPoint Class Methods and Properties

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AcceptSocket()</td>
<td>Accepts an incoming TCP connection request and returns a Socket class</td>
</tr>
<tr>
<td>AcceptTcpClient()</td>
<td>Accepts an incoming TCP connection request and returns a TcpClient class</td>
</tr>
<tr>
<td>Pending()</td>
<td>Determines whether any incoming connection requests are waiting</td>
</tr>
<tr>
<td>Start()</td>
<td>Starts listening for incoming requests</td>
</tr>
<tr>
<td>Stop()</td>
<td>Stops listening for incoming requests</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
<th>Get/Set</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LocalEndpoint</td>
<td>Get</td>
<td>Gets the local Endpoint to which the TcpListener is bound</td>
</tr>
</tbody>
</table>
Once you have constructed a TcpListener object, you can have it start listening on the port that you passed in by calling the Start() method. Now that you have a TcpListener socket that is awaiting a connection, let's take a brief look at network streams.

**Using Network Streams**

The System.Net.Sockets.NetworkStream class is used for both sending and receiving data over a TCP socket. To create a Network Stream object, use one of the following constructors:

```csharp
public NetworkStream(Socket socket);
public NetworkStream(Socket socket, bool ownsSocket);
public NetworkStream(Socket socket, FileAccess access);
public NetworkStream(Socket socket, FileAccess access, bool ownsSocket);
```

Each constructor specifies a Socket class with which the new stream object should be associated. The ownsSocket parameter should be set to TRUE if you want the Stream object to assume ownership of the socket. The access parameter can be used to specify any FileAccess values for determining access to the stream (such as Read, Write, or ReadWrite).

In addition, you can use the TcpClient.GetStream() method (as you will see in the next section) to get the NetworkStream for the active connection.

The NetworkStream class supports the methods and properties described in Table 12.9.

The following example shows how you can use the NetworkStream class to send data to a client that is connected to a TcpListener object:

```csharp
// Create a socket that is listening for incoming connections on // port 8080
string hostName = System.Net.Dns.GetHostName();
System.Net.IPAddress localIPAddress =
    System.Net.Dns.Resolve(hostName).AddressList[0];
    TcpListener(localIPAddress, 8080);

// Start listening synchronously
tcpServer.Start();
```
// Get the client socket when a request comes in
Socket tcpClient = tcpServer.AcceptSocket();

// Make sure the client is connected
if (tcpClient.Connected == false)
    return;

// Create a network stream to send data to the client
NetworkStream clientStream = new NetworkStream(tcpClient);

// Write some data to the stream
byte[] serverBytes = System.Text.Encoding.ASCII.GetBytes("Howdy. You've connected!\r\n");
clientStream.Write(serverBytes, 0, serverBytes.Length);

---

### Table 12.9 NetworkStream Class Methods and Properties

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeginRead()</td>
<td>Begins an asynchronous Read() operation</td>
</tr>
<tr>
<td>BeginWrite()</td>
<td>Begins an asynchronous Write() operation</td>
</tr>
<tr>
<td>Close()</td>
<td>Closes the NetworkStream</td>
</tr>
<tr>
<td>CreateWaitHandle()</td>
<td>Creates a WaitHandle object for handling asynchronous operation blocking events</td>
</tr>
<tr>
<td>Dispose()</td>
<td>Releases resources used by the NetworkStream object</td>
</tr>
<tr>
<td>EndRead()</td>
<td>Ends asynchronous Read() operation</td>
</tr>
<tr>
<td>EndWrite()</td>
<td>Ends asynchronous Write() operation</td>
</tr>
<tr>
<td>Read()</td>
<td>Reads from the NetworkStream</td>
</tr>
<tr>
<td>ReadByte()</td>
<td>Reads a byte from the NetworkStream</td>
</tr>
<tr>
<td>Write()</td>
<td>Writes to the NetworkStream</td>
</tr>
<tr>
<td>WriteByte()</td>
<td>Writes a byte to the NetworkStream</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
<th>Get/Set</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanRead</td>
<td>Get</td>
<td>Returns TRUE if the NetworkStream supports reading</td>
</tr>
<tr>
<td>CanWrite</td>
<td>Get</td>
<td>Returns TRUE if the NetworkStream supports write operations</td>
</tr>
<tr>
<td>DataAvailable</td>
<td>Get</td>
<td>Returns TRUE if the NetworkStream has data to be read</td>
</tr>
<tr>
<td>Length</td>
<td>Get</td>
<td>Returns the amount of data waiting to be read on the stream</td>
</tr>
</tbody>
</table>
// Immediately disconnect the client
tcpClient.Shutdown(SocketShutdown.Both);
tcpClient.Close();

**TCP Clients**

To establish a connection with a TCP server listening on a specific port, you use the `System.Net.Sockets.TcpClient` class. Its constructor is defined as follows:

```csharp
public TcpClient();
public TcpClient(IPEndPoint localEP);
public TcpClient(string hostname, int port);
```

The `TcpClient` class has the methods and properties described in Table 12.10.

**Table 12.10**  
**TcpClient Class Methods and Properties**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Close()</td>
<td>Closes the <code>TcpClient</code> socket</td>
</tr>
<tr>
<td>Connect()</td>
<td>Connects to a remote host</td>
</tr>
<tr>
<td>GetStream()</td>
<td>Gets the <code>NetworkStream</code> object to send and receive data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
<th>Get/Set</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LingerState</td>
<td>Get/set</td>
<td>Gets or sets the socket linger time</td>
</tr>
<tr>
<td>NoDelay</td>
<td>Get/set</td>
<td>Set to <code>TRUE</code> to disable the delay on a socket when the receive buffer is not full</td>
</tr>
<tr>
<td>ReceiveBufferSize</td>
<td>Get/set</td>
<td>Gets or sets the receive buffer size</td>
</tr>
<tr>
<td>SendBufferSize</td>
<td>Get/set</td>
<td>Gets or sets the send buffer size</td>
</tr>
</tbody>
</table>

Now that you have looked at both of the TCP client and server classes, let’s examine how you could use the `TcpListener` class to write a small (and extremely simple) Web server that runs on the Pocket PC:

```csharp
using System;
using System.Data;
using System.Net.Sockets;
```
namespace TCPServer {
    class WebServer {
        static void Main(string[] args) {
            // Create a socket that is listening for incoming
            // connections on port 80.
            string hostName = System.Net.Dns.GetHostName();
            System.Net.IPAddress localIPAddress =
                System.Net.Dns.Resolve(hostName).AddressList[0];
                TcpListener(localIPAddress, 80);

            // Start listening synchronously and wait for an
            // incoming socket
            tcpServer.Start();
            Socket tcpClient = tcpServer.AcceptSocket();

            // Make sure the client is connected
            if (tcpClient.Connected == false)
                return;

            // Create a network stream that we will use to send
            // and receive data.
            NetworkStream clientStream = new NetworkStream
                (tcpClient);

            // Get a basic request.
            byte[] requestString = new byte[1024];
            clientStream.Read(requestString, 0, 1024);

            // Do something with the client request here.
            // Typically, you'll need to parse the request, open the
            // file and send the contents back. For this example,
            // we'll just write out a simple HTTP response to the
            // stream.
            byte[] responseString =
                System.Text.Encoding.ASCII.GetBytes("HTTP/1.0 200 OK\r\n\r\nTest Response\r\n\r\n");
            clientStream.Write(responseString, 0,
                responseString.Length);
        }
    }
}
// Disconnect the client
tcpClient.Shutdown(SocketShutdown.Both);
tcpClient.Close();
}
}

Let's also take a look at the code for a small client that requests a Web page from the server:

using System;
using System.Data;
using System.Net.Sockets;

namespace TCPWebClientTest {
    class WebClientTest {
        static void Main(string[] args) {
            // Create a socket that will grab a Web page

            // Set up the HTTP request string to get the main index page
            byte[] httpRequestBytes = System.Text.Encoding.ASCII.GetBytes("GET / HTTP/1.0\r\n\r\n");

            // Connect the socket to the server
            tcpWebClient.Connect("www.microsoft.com", 80);

            // Make sure we are connected
            if(tcpWebClient == null)
                return;

            // Create a network stream that we will use to send // and receive data.
            NetworkStream webClientStream = tcpWebClient.GetStream();

            // Send the request synchronously
            webClientStream.Write(httpRequestBytes, 0, httpRequestBytes.Length);
// Get the response from the request. We will continuously
// request 4096 bytes from the response stream and concat
// the string into the strResponse variable.
string strResponse = "";
byte[] httpResponseBytes = new byte[4096];
int bytesRead = webClientStream.Read
    (httpResponseBytes, 0, httpResponseBytes.Length);

strResponse = System.Text.Encoding.ASCII.GetString(httpResponseBytes, 0, bytesRead);

while(bytesRead > 0) {
    bytesRead = webClientStream.Read
        (httpResponseBytes, 0, httpResponseBytes.Length);
    strResponse = strResponse + System.Text.Encoding.ASCII.GetString(httpResponseBytes, 0, bytesRead);
}

// At this point, the strResponse string has the
// Web page. Do something with it

// Clean up the socket
tcpWebClient.Close();
}

Sending and Receiving Data over UDP

Both the sending and receiving of a datagram (or packet) over a connectionless socket is handled by the System.Net.Sockets.UdpClient class. A new UdpClient object is created by using one of the following constructors:

public UdpClient();
public UdpClient(int port);
public UdpClient(IPEndPoint localEP);
public UdpClient(string hostname, int port);

The UdpClient class supports the methods and properties described in Table 12.11.

**Table 12.11 UdpClient Class Methods and Properties**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Close()</td>
<td>Closes the UDP socket</td>
</tr>
<tr>
<td>Connect()</td>
<td>Connects to a remote host</td>
</tr>
<tr>
<td>DropMulticastGroup()</td>
<td>Leaves a multicast group</td>
</tr>
<tr>
<td>JoinMulticastGroup()</td>
<td>Joins a multicast group</td>
</tr>
<tr>
<td>Receive()</td>
<td>Receives a UDP datagram from a remote host</td>
</tr>
<tr>
<td>Send()</td>
<td>Sends a UDP datagram to a remote host</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
<th>Get/Set</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>Get/set</td>
<td>Indicates whether a connection has been made to a remote host</td>
</tr>
<tr>
<td>Client</td>
<td>Get/set</td>
<td>Gets or sets the socket handle</td>
</tr>
</tbody>
</table>

The following code shows how you can create a socket that sends a UDP datagram to a specific host and port:

```csharp
using System;
using System.Data;
using System.Net;
using System.Net.Sockets;

namespace udpTest {
    class UdpTestSend {
        static void Main(string[] args) {
            // Setup the target device address. For this sample, we are assuming it is a machine at 192.168.123.199, and on port 40040.
            System.Net.IPEndPoint ipTarget = new IPEndPoint(System.Net.IPAddress.Parse("192.168.123.199"), 40040);
```
System.Net.Sockets.UdpClient udpSend = new UdpClient(ipTarget);

    // Send a datagram to the target device
    byte[] sendBytes = System.Text.Encoding.ASCII.GetBytes("Testing a datagram buffer");
    udpSend.Send(sendBytes, sendBytes.Length);

The code for receiving the datagram would look like the following:

using System;
using System.Data;
using System.Net;
using System.Net.Sockets;

namespace udpTest {
    class UdpTestListen {
        static void Main(string[] args) {
            // Listen for datagrams on port 40040
            if (udpListener == null)
                return;
            // Create an endpoint for the incoming datagram
            IPEndPoint remoteEndPoint = new IPEndPoint(IPAddress.Any, 40040);
            // Get the datagram
            byte[] recvBytes = udpListener.Receive(ref remoteEndPoint);
            string returnData = System.Text.Encoding.ASCII.GetString(recvBytes, 0, recvBytes.Length);
            // Do something with the data....
        }
    }
}
Internet Protocols and the .NET Pluggable Protocol Model

When requesting data over the Internet using a standardized protocol such as HTTP (the protocol for the Web), you use a Uniform Resource Identifier (URI) to specify the protocol, server, and name of the resource that you are attempting to access. The .NET Compact Framework provides two abstract classes for handling any Internet resource request and response: System.Net.WebRequest and System.NetWebResponse.

Client applications use the WebRequest class to make the request for a specific URI from an Internet location over a specific protocol (such as HTTP or FTP). Instead of calling a constructor for the WebRequest class, you initialize a new request by calling the WebRequest.Create() method. This automatically instantiates a new request object based on the protocol that you used for the request. For example, if you are trying to access a resource on the Web using the HTTP protocol, you are returned an HttpWebRequest object for which you can set properties and receive a response stream.

Once your request has been configured, you can call the WebRequest.GetResponse() method to get a Stream class that is used to receive the data from the request.

The WebRequest object is an abstract class that contains the methods and properties described in Table 12.12.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abort()</td>
<td>Cancels an asynchronous request to an Internet resource</td>
</tr>
<tr>
<td>BeginGetRequestStream()</td>
<td>Begins an asynchronous GetRequestStream() operation</td>
</tr>
<tr>
<td>BeginGetResponse()</td>
<td>Begins an asynchronous GetResponse() operation</td>
</tr>
<tr>
<td>Create()</td>
<td>Creates a new WebRequest object</td>
</tr>
<tr>
<td>EndGetRequestStream()</td>
<td>Ends an asynchronous GetRequestStream() operation</td>
</tr>
<tr>
<td>EndGetResponse()</td>
<td>Ends an asynchronous GetResponse() operation</td>
</tr>
</tbody>
</table>

(continued)
The `WebResponse` object is also abstract, and contains the methods and properties described in Table 12.13.

### Table 12.12 WebRequest Class Methods and Properties (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetRequestStream()</td>
<td>Gets a <code>Stream</code> class for writing data to the Internet resource</td>
</tr>
<tr>
<td>GetResponse()</td>
<td>Gets a <code>WebResponse</code> object that returns the response to an Internet request</td>
</tr>
<tr>
<td>RegisterPrefix()</td>
<td>Registers a new URI type</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
<th>Get/Set</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectionGroupName</td>
<td>Get/set</td>
<td>Abstract property used to get or set the connection group name in descendant classes</td>
</tr>
<tr>
<td>ContentLength</td>
<td>Get/set</td>
<td>Abstract property used to get or set the length of the request data</td>
</tr>
<tr>
<td>ContentType</td>
<td>Get/set</td>
<td>Abstract property used to get or set the content type of the request</td>
</tr>
<tr>
<td>Credentials</td>
<td>Get/set</td>
<td>Abstract property used to get or set the credentials for the request</td>
</tr>
<tr>
<td>Headers</td>
<td>Get/set</td>
<td>Abstract property used to get or set the headers and values for the request</td>
</tr>
<tr>
<td>Method</td>
<td>Get/set</td>
<td>Abstract property used to get or set the method used for the request</td>
</tr>
<tr>
<td>PreAuthenticate</td>
<td>Get/set</td>
<td>Abstract property used to determine whether the request should be pre-authenticated</td>
</tr>
<tr>
<td>Proxy</td>
<td>Get/set</td>
<td>Abstract property used to get or set the proxy to be used for the request</td>
</tr>
<tr>
<td>RequestUri</td>
<td>Get/set</td>
<td>Abstract property used to get or set the URI for the request</td>
</tr>
<tr>
<td>Timeout</td>
<td>Get/set</td>
<td>Abstract property used to get or set the length of time before the request times out</td>
</tr>
</tbody>
</table>
Both the WebRequest and WebResponse abstract classes form the basis for what is known as **pluggable protocols**. The concept of pluggable protocols is fairly straightforward—a client application can make a request for any Internet resource using a URI and not have to worry about the underlying details of the network protocol being used. When a request is made using the WebRequest.Create() method, the appropriate protocol-specific class is automatically instantiated and returned to the client application.

Consider the following request for a Web resource:

```csharp
// Set up the URI

// Make the request
HttpWebRequest httpReq = (HttpWebRequest)WebRequest.Create(urlRequest);

// Get the response
HttpWebResponse webResponse = (HttpWebResponse)httpReq.GetResponse();
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Close()</td>
<td>Closes the response stream</td>
</tr>
<tr>
<td>GetResponseStream()</td>
<td>Gets the Stream for reading the response</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
<th>Get/Set</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContentLength</td>
<td>Get/set</td>
<td>Abstract property used to get or set the length of the data being received</td>
</tr>
<tr>
<td>ContentType</td>
<td>Get/set</td>
<td>Abstract property used to get or set the content type for the data being received</td>
</tr>
<tr>
<td>Headers</td>
<td>Get/set</td>
<td>Abstract property used to get or set the headers and values of the request</td>
</tr>
<tr>
<td>RequestUri</td>
<td>Get/set</td>
<td>Abstract property used to get or set the URI for the resource requested</td>
</tr>
</tbody>
</table>

Table 12.13  WebResponse Class Methods and Properties
This request will return a new object that is based on the `HttpWebRequest` class. The `HttpWebRequest` class is actually derived from `WebRequest`, but adds all of the protocol specifics surrounding HTTP.

What makes the pluggable protocol model extremely useful is that you can also use it to create your own classes for handling new protocols that are not native to the .NET Compact Framework.

**Creating a Pluggable Protocol**

Any new class that is designed to be used as a pluggable protocol is always derived from `WebRequest` and `WebResponse`. All new pluggable protocol classes must also be registered with the base `WebRequest` object in order for the `WebRequest.Create()` method to appropriately instantiate the correct object for the protocol.

To register a new protocol with the `WebRequest` class, you can use the following function:

```csharp
public static bool WebRequest.RegisterPrefix(string prefix, IWebRequestCreate creator);
```

The first parameter, `prefix`, is a string that represents the protocol that will be used in URI requests for the new object. For example, if you were creating a new protocol that handled requests for resources over the File Transfer Protocol (such as `ftp://ftp.microsoft.com/dir/filename.txt`), you could simply use `ftp` for the prefix value. The `creator` parameter should be set to an object that implements the `IWebRequestCreate` interface, which is used to create the new `WebRequest` class.

The following code shows the basic layout for creating a new protocol-specific class that can be used by the `WebRequest.Create()` method:

```csharp
/// <summary>Ftp request protocol handler</summary>
class FtpWebRequest: WebRequest {
    // Private internal variables.
    private NetworkCredential reqCredentials;
    private WebHeaderCollection reqHeaders;
    private WebProxy reqProxy;
    private System.Uri reqUri;
    private string reqConnGroup;
    private long reqContentLength;
    private string reqContentType;
    private string reqMethod;
```
private bool reqPreAuthen;
private int reqTimeout;

// Constructor
public FtpWebRequest(System.Uri uri) {
    reqHeaders = new WebHeaderCollection();
    reqUri = uri;
}

// Properties
public override string ConnectionGroupName {
    get { return reqConnGroup; }
    set { reqConnGroup = value; }
}

public override long ContentLength {
    get { return reqContentLength; }
    set { reqContentLength = value; }
}

public override string ContentType {
    get { return reqContentType; }
    set { reqContentType = value; }
}

public override ICredentials Credentials {
    get { return reqCredentials; }
    set { reqCredentials = (System.Net.NetworkCredential)value; }
}

public override WebHeaderCollection Headers {
    get { return reqHeaders; }
    set { reqHeaders = value; }
}

public override string Method {
    get { return reqMethod; }
    set { reqMethod = value; }
}

public override bool PreAuthenticate {
    get { return reqPreAuthen; }
    set { reqPreAuthen = value; }
}

public override IWebProxy Proxy {
    get { return reqProxy; }
    set { reqProxy = (System.Net.WebProxy)value; }
}

public override Uri RequestUri {
get { return reqUri; }
}

public override int Timeout {
    get { return reqTimeout; }
    set { reqTimeout = value; }
}

// Methods. These are just stubbed in here for this example.
// In an actual FTP client, you would need to implement these by
// using p/Invoke to call into the WinInet FTP functions.
public override void Abort() {
    base.Abort();
}

public override IAsyncResult BeginGetRequestStream
(AsyncCallback callback, object state) {
    return base.BeginGetRequestStream (callback, state);
}

public override IAsyncResult BeginGetResponse
(AsyncCallback callback, object state) {
    return base.BeginGetResponse (callback, state);
}

public override Stream EndGetRequestStream(IAsyncResult asyncResult) {
    return base.EndGetRequestStream (asyncResult);
}

public override WebResponse EndGetResponse(IAsyncResult asyncResult) {
    return base.EndGetResponse (asyncResult);
}

public override Stream GetRequestStream() {
    return base.GetRequestStream();
}

public override WebResponse GetResponse() {
    return base.GetResponse();
}
}

/// <summary>Ftp request registration interface</summary>
class FtpWebRequestCreate: IWebRequestCreate {
    public System.Net.WebRequest Create(System.Uri uri) {

System.Net.WebRequest request = new FtpWebRequest
    (uri);
    return request;
}
}

/// <summary>Ftp request response handler</summary>
class FtpWebResponse : WebResponse {
    // Private internal variables.
    private WebHeaderCollection respHeaders;
    private System.Uri respUri;
    private long respContentLength;
    private string respContentType;

    // Properties
    public override long ContentLength {
        get { return respContentLength; }
        set { respContentLength = value; }
    }
    public override string ContentType {
        get { return respContentType; }
        set { respContentType = value; }
    }
    public override WebHeaderCollection Headers {
        get { return respHeaders; }
        set { respHeaders = value; }
    }
    public override Uri ResponseUri {
        get { return reqUri; }
    }

    // Methods. These are just stubbed in here for this example.
    // In an actual FTP client, you would need to implement these by
    // using p/Invoke to call into the WinInet FTP functions.
    public override void Close() {
        base.Close();
    }
    public override Stream GetResponseStream() {
        return base.GetResponseStream();
    }
}
Remember that you also need to register the protocol with the WebRequest class in order for it to be properly instantiated:

```csharp
class FtpTest {
    static void Main(string[] args) {
        // Create a pluggable protocol

        // Register it
        WebRequest.RegisterPrefix("ftp", new FtpWebRequestCreate());

        // Make the request
        FtpWebRequest ftpClient = (FtpWebRequest)WebRequest.Create(urlRequest);

        // Get the response
        FtpWebResponse ftpResponse = (FtpWebResponse)ftpClient.GetResponse();

        // Use a StreamReader class to read in the response

        // Since FTP can be binary or ASCII, you would want
        // to copy it in chunks to the destination file...

        // Close the stream
        responseStream.Close();
    }
}
```

**Accessing Content on the Web**

One of the built-in pluggable protocols available in the .NET Compact Framework for handling HTTP and HTTPS requests to the Internet is the HttpWebRequest class. As with any other protocol-specific class, it has been derived from the WebRequest class and can be created by using the WebRequest.Create() method:
HttpWebRequest httpReq =
furrygoat.com");

The HttpWebRequest class contains the methods and properties described in Table 12.14.

Table 12.14 HttpWebRequest Class Methods and Properties

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abort()</td>
<td>Cancels an asynchronous request to an Internet resource</td>
</tr>
<tr>
<td>AddRange()</td>
<td>Adds a Range header to the request</td>
</tr>
<tr>
<td>BeginGetRequestStream()</td>
<td>Begins an asynchronous GetRequestStream() operation</td>
</tr>
<tr>
<td>BeginGetResponse()</td>
<td>Begins an asynchronous GetResponse() operation</td>
</tr>
<tr>
<td>EndGetRequestStream()</td>
<td>Ends an asynchronous GetRequestStream() operation</td>
</tr>
<tr>
<td>EndGetResponse()</td>
<td>Ends an asynchronous GetResponse() operation</td>
</tr>
<tr>
<td>GetRequestStream()</td>
<td>Gets a Stream class for writing data to the Internet resource</td>
</tr>
<tr>
<td>GetResponse()</td>
<td>Gets a WebResponse object that returns the response to an Internet request</td>
</tr>
<tr>
<td>RegisterPrefix()</td>
<td>Registers a new URI type</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
<th>Get/Set</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accept</td>
<td>Get/set</td>
<td>Gets or sets the HTTP Accept header</td>
</tr>
<tr>
<td>Address</td>
<td>Get</td>
<td>Gets the URI of the resource that responded to the request</td>
</tr>
<tr>
<td>AllowAutoRedirect</td>
<td>Get/set</td>
<td>Indicates whether the request should follow a redirect</td>
</tr>
<tr>
<td>AllowWriteStream Buffering</td>
<td>Get/set</td>
<td>Indicates whether to buffer the data sent to the resource</td>
</tr>
<tr>
<td>Connection</td>
<td>Get/set</td>
<td>Gets or sets the HTTP Connection header</td>
</tr>
<tr>
<td>ConnectionGroupName</td>
<td>Get/set</td>
<td>Gets or sets the name of the connection group</td>
</tr>
</tbody>
</table>

(continued)
Table 12.14  HttpRequest Class Methods and Properties *(continued)*

<table>
<thead>
<tr>
<th>Property</th>
<th>Get/Set</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContentLength</td>
<td>Get/set</td>
<td>Gets or sets the HTTP Content-Length header</td>
</tr>
<tr>
<td>ContentType</td>
<td>Get/set</td>
<td>Gets or sets the HTTP Content-Type header</td>
</tr>
<tr>
<td>ContinueDelegate</td>
<td>Get/set</td>
<td>Gets or sets the delegate for HTTP requests</td>
</tr>
<tr>
<td>Credentials</td>
<td>Get/set</td>
<td>Gets or sets credentials for the request</td>
</tr>
<tr>
<td>Expect</td>
<td>Get/set</td>
<td>Gets or sets the HTTP Expect header</td>
</tr>
<tr>
<td>Headers</td>
<td>Get</td>
<td>Gets the collection of HTTP headers for the request</td>
</tr>
<tr>
<td>IfModifiedSince</td>
<td>Get/set</td>
<td>Gets or sets the HTTP If-Modified-Since header</td>
</tr>
<tr>
<td>KeepAlive</td>
<td>Get/set</td>
<td>Indicates whether or not the HTTP request should use a persistent connection</td>
</tr>
<tr>
<td>MaximumAutomatic</td>
<td>Get/set</td>
<td>Gets or sets the number of HTTP redirects the request will comply with</td>
</tr>
<tr>
<td>Redirections</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MediaType</td>
<td>Get/set</td>
<td>Gets or sets the media type of the request</td>
</tr>
<tr>
<td>Method</td>
<td>Get/set</td>
<td>Gets or sets the HTTP method used with the request</td>
</tr>
<tr>
<td>Pipelined</td>
<td>Get/set</td>
<td>Indicates whether the request is pipelined</td>
</tr>
<tr>
<td>PreAuthenticate</td>
<td>Get/set</td>
<td>Indicates whether to pre-authenticate a request</td>
</tr>
<tr>
<td>ProtocolVersion</td>
<td>Get/set</td>
<td>Gets or sets the HTTP version to use with the request</td>
</tr>
<tr>
<td>Proxy</td>
<td>Get/set</td>
<td>Gets or sets proxy information</td>
</tr>
<tr>
<td>Referer</td>
<td>Get/set</td>
<td>Gets or sets the HTTP Referer header</td>
</tr>
<tr>
<td>RequestUri</td>
<td>Get</td>
<td>Gets the original request URI</td>
</tr>
<tr>
<td>SendChunked</td>
<td>Get/set</td>
<td>Indicates whether to send the data in segments</td>
</tr>
<tr>
<td>ServicePoint</td>
<td>Get</td>
<td>Gets the service point for the request</td>
</tr>
<tr>
<td>Timeout</td>
<td>Get/set</td>
<td>Gets or sets the time-out value</td>
</tr>
<tr>
<td>TransferEncoding</td>
<td>Get/set</td>
<td>Gets or sets the HTTP Transfer-Encoding header</td>
</tr>
<tr>
<td>UserAgent</td>
<td>Get/set</td>
<td>Gets or sets the HTTP User-Agent header</td>
</tr>
</tbody>
</table>
To get the results for the request that was made by the \texttt{HttpWebRequest} object, you can use the \texttt{GetResponse()} method:

\begin{verbatim}
HttpWebResponse webResponse =
  (HttpWebResponse)httpReq.GetResponse();
\end{verbatim}

The \texttt{HttpWebResponse} class supports the methods and properties described in Table 12.15.

\begin{table}[h]
\centering
\caption{HttpWebResponse Class Methods and Properties}
\begin{tabular}{|l|l|}
\hline
\textbf{Method} & \textbf{Description} \\
\hline
Close() & Closes the response stream \\
GetResponseHeader() & Gets the header that was returned for the response \\
GetResponseStream() & Gets the Stream for reading the response \\
\hline
\end{tabular}
\end{table}

\begin{table}[h]
\centering
\caption{HttpWebResponse Class Methods and Properties}
\begin{tabular}{|l|l|l|}
\hline
\textbf{Property} & \textbf{Get/Set} & \textbf{Description} \\
\hline
CharacterSet & Get & Gets the character set for the response \\
ContentEncoding & Get & Gets the encoding scheme used for the response \\
ContentLength & Get & Gets the length of the response \\
ContentType & Get & Gets the type of the response \\
Headers & Get & Gets the headers associated with the response \\
LastModified & Get & Gets the last modified time of the response \\
Method & Get & Gets the method used to return the response \\
ProtocolVersion & Get & Gets the HTTP version used for the response \\
ResponseUri & Get & Gets the URI of the resource that responded to the request \\
Server & Get & Gets the name of the server that sent the response \\
StatusCode & Get & Gets the HTTP status code for the response \\
StatusDescription & Get & Gets the HTTP status description for the response \\
\hline
\end{tabular}
\end{table}

The following code shows how to create a new request for a Web resource, using the \texttt{StreamReader} class to read in the response that you receive from the Web server:

\begin{verbatim}
using System;
using System.Data;
\end{verbatim}
using System.Net;
using System.IO;
namespace WebSample {
    class WebTest {
        static void Main(string[] args) {
            // Make a new WebRequest object
            HttpWebRequest webClient = (HttpWebRequest)WebRequest.Create(urlRequest);

            // Get the response
            HttpWebResponse webResponse = (HttpWebResponse)webClient.GetResponse();

            // Use a StreamReader class to read in the response
            StreamReader responseStream = new StreamReader(webResponse.GetResponseStream(), System.Text.Encoding.ASCII);

            // Copy the stream to a string, do something with it
            // string strResponse = responseStream.ReadToEnd();

            // Close the stream
            responseStream.Close();
        }
    }
}

The response stream, strResponse, contains the HTML code that was downloaded from the Web site:

<HTML>
<title>The Furrygoat Experience</title>
<body>
<p><b><font face="Arial">This is the Furrygoat homepage!</font></b></p>
</body>
</HTML>
Consuming Web Services

.NET Web Services is a form of distributed computing that enables your application to use the logic of a remote component over the Internet using standard protocols. Web Services is one of the most exciting aspects of using the Compact Framework on a mobile device such as Pocket PC, because it enables you to create rich applications that can access Web Service data from one or many sources without being tethered to a desktop.

For example, consider a Pocket PC device that has a GPS unit attached to it over the serial port (you may someday even be able to use a Pocket PC Phone Edition device to request your current position based on the nearest cellular tower). You could hypothetically use a Web Service to request a map of your current surroundings based on the longitude and latitude that the GPS provides. You could then access another Web Service to get a list of the ATMs or restaurants in your local area. By using Web Services, your applications can focus on tying remote data together into a useful program, rather than concentrate on how to get the data to your device, or replicate functionality that has already been developed elsewhere.

The .NET Compact Framework supports the following functionality regarding Web Services on a Pocket PC device:

- All Web Services must be based on the HTTP protocol. Other protocols, such as SMTP, are not supported.
- Data is transmitted using the Simple Object Access Protocol (SOAP) XML format.
- The Compact Framework supports consuming Web Services by client applications only, and does not natively support hosting them. If you need to support hosting a Web Service using the Compact Framework, you can manually build an HTTP listener (using the TcpListener class) and manually handle incoming SOAP requests.

**TIP:** A great Web site for finding Web Services that are publicly available on the Internet is www.xmethods.com. There you can find Web Services for everything from currency conversion to stock quotes.

In the next section, you will learn what is involved on the client side to consume a Web Service on the Pocket PC.
The Microsoft TerraServer Web Service

The Microsoft TerraServer, located at http://terraserver.microsoft.com, is a massive database (about 3.3 terabytes) of both satellite images and topographic maps for much of the United States. By using TerraServer's search engine, you can zoom in on aerial images for almost any street in the U.S., as well as obtain data about surrounding landmarks. TerraServer fortunately also provides a Web Service that you can use to perform queries and get maps from the database (which is rather nice, as it would be rather difficult to store all 3.3 terabytes on a Pocket PC).

In this section, we will use the TerraServer Web Services (also called TerraService) as an example of how you can use and consume .NET Web Services on a Pocket PC device using the .NET Compact Framework. More information about the Web Service API that TerraServer provides is available at http://terraserver.homeadvisor.msn.com/webservices.aspx.

The first thing you need to do to consume a Web Service is create a new project. To do this, select the Smart Device Application project type under the Visual C# Project tree. For this example, let's call the new project TerraServiceTest.

After the project has been created, you need to add a new reference for the Web Service you are planning to use in your class. All you need to do is right-click on References in the Solution Explorer and select Add Web Reference (see Figure 12.1).

The Add Web Reference dialog box will appear (see Figure 12.2). In it, you specify the URL for the WSDL or ASMX file that describes the Web Service. The TerraServer Web Service description is located at http://terraserver.homeadvisor.msn.com/TerraService.asmx.

After you have entered the URL, click the Add Reference button. This will cause Visual Studio to generate a proxy class that will be used by
your project to access the Web Service. Once this has completed, you will notice that the reference to the Web Service is now in your project (see Figure 12.3).

![Figure 12.2 Entering the URL for the Web Service](image)

![Figure 12.3 The Web reference is added to the project.](image)
Now, all you need to do to use the TerraServer Web Service is add the namespace to your current project as follows:

```csharp
using TerraServiceTest.com.msn.homeadvisor.terraserver;
```

That's it! Your C# program can now use the APIs and structures that are part of the Web Service just as if the component were on the device. Let's take a look at some sample code that uses TerraService to download a “tile” of satellite image data for the Redmond, Washington, area:

```csharp
using System;
using System.Data;
using System.IO;
using TerraServiceTest.com.msn.homeadvisor.terraserver;

namespace TerraServiceTest {
    /// <summary>Summary description for Class1. </summary>
    class Class1 {
        /// <summary> The main entry point for the application. </summary>
        static void Main(string[] args) {
            // Create a new TerraService object
            TerraService ts = new TerraService();

            // Build a place to request tile information on
            Place pl = new Place();
            pl.City = "Redmond";
            pl.State = "WA";
            pl.Country = "USA";
            PlaceFacts pf = ts.GetPlaceFacts(pl);

            // Get the bounding box for the area
            AreaBoundingBox abb = ts.GetAreaFromPt(pf.Center,
                Theme.Photo, Scale.Scale16m, 640, 480);

            // Grab the center tile
            Byte[] imageBytes = ts.GetTile(abb.Center.
                TileMeta.Id);

            // Create a new file and dump the buffer to it
            FileStream outputFileStream = new FileStream("\map.jpg", FileMode.CreateNew);
```
BinaryWriter outputBinaryWriter = new BinaryWriter
    (outputFileStream);

    // Write
    outputBinaryWriter.Write(imageBytes, 0, imageBytes.
    Length);

    // Clean up
    outputBinaryWriter.Close();
    outputFileStream.Close();
}

After the class has completed, you can view the downloaded map by
launching Pocket Internet Explorer (see Figure 12.4).

Figure 12.4  Satellite map of the Redmond, WA, area downloaded via
TerraServer
Pocket PC and P/Invoke

The last topic we are going to cover regarding the .NET Compact Framework is its ability to call into unmanaged code using **Platform Invoke (P/Invoke)**. As you have seen throughout this book, most of the APIs that are supported on a Pocket PC platform are exported by using dynamic link libraries (DLLs) that your application imports. By using the P/Invoke service, you can also access the same API functions from within a .NET application. This enables you to integrate much of the functionality that is native to the Pocket PC, and not natively supported by the Compact Framework. For example, the Pocket PC Phone Edition supports the capability to send and receive SMS messages (see Chapter 8). Although the Compact Framework does not come with any classes to support this, you can use P/Invoke to enable your managed code to call into the unmanaged SMS API found in the cellcore.dll library.

To declare within your application a method that will use P/Invoke, you need to use the **DllImport** attribute, which supports the fields described in Table 12.16.

**Table 12.16** DllImport Attributes

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EntryPoint</td>
<td>The function name that you want to call into</td>
</tr>
<tr>
<td>CharSet</td>
<td>Specifies how the string arguments should be marshaled</td>
</tr>
<tr>
<td>CallingConvention</td>
<td>Specifies the calling convention to use when passing arguments</td>
</tr>
<tr>
<td>SetLastError</td>
<td>Set this value to TRUE to enable calling the Marshal.GetLastWin32Error method to check if an error occurred when invoking this method</td>
</tr>
</tbody>
</table>

For example, the following code shows how you can use the MessageBox() function from a managed application by using P/Invoke:

```csharp
using System;
using System.Data;
using System.Runtime.InteropServices;

namespace invokeTest {
    class Class1 {
```
// Hook up Windows API methods
[DllImport("coredll.dll", EntryPoint="MessageBox",
CharSet=CharSet.Unicode, SetLastError=true)
static extern Int32 MessageBox(Int32 hWnd, string
stText,
string stCaption, Int32 mbType);

static void Main(string[] args) {
    // Call into the MessageBox function
    MessageBox(0, "MessageText", "MessageCaption", 0);
}

Once a function has been declared with the DllImport attribute, you can then call it in the same manner as any other managed function.

Note a few minor differences regarding P/Invoke on the .NET Compact Framework when comparing it to its desktop counterpart:

- There is no Unicode-to-ANSI string conversion. All string pointers are passed to an unmanaged function as a Unicode string.
- There is no marshaling of objects contained within structures.
- If a function returns a pointer to a structure, it is not marshaled to a managed structure. You need to create a wrapper function that handles simple data types.
- Platform Invoke services does not support COM interoperability with the Compact Framework. If you wish to call into COM objects, you need to create a wrapper DLL that exports non-COM-based functions.
- The DllImport attribute supports only the CharSet.Unicode and CharSet.Auto character sets.
- The DllImport attribute supports only the CallingConvention.Winapi calling convention.

Sending an SMS Message from .NET

The following example shows a slightly more complicated way of using the Platform Invoke services. Because the Compact Framework does not support the marshaling of objects that are contained within a structure, you need to create a C++ “wrapper” library in order to call the Pocket PC Phone Edition’s SMS API functions (see Chapter 8).
First, create the wrapper library using Embedded Visual C++ 3.0. The code for the library will look as follows:

```
// First is the definition file for the DLL
// smsinvoke.def
LIBRARY SMSINVOKE
EXPORTS
    SendSMSInvokeMsg 01

// Here is the wrapper DLL
// smsinvoke.cpp
#include <windows.h>
#include <sms.h>

#ifdef __cplusplus
extern "C" {
#endif
__declspec(dllexport) BOOL SendSMSInvokeMsg(TCHAR *tchPhoneNumber, TCHAR *tchMessage);
#ifdef __cplusplus
}
#endif

BOOL WINAPI DllMain(HANDLE hinstDLL, DWORD dwReason, LPVOID lpvReserved)
{
    return TRUE;
}

BOOL SendSMSInvokeMsg(TCHAR *tchPhoneNumber, TCHAR *tchMessage)
{
    SMS_HANDLE hSms = NULL;
    HANDLE hSmsEvent = NULL;
    HRESULT hr = S_OK;
    BOOL fReturn = FALSE;

    // Make sure we have a number and a message
    if(!tchPhoneNumber || !tchMessage)
        return fReturn;
```
// Open up SMS
hr = SmsOpen(SMS_MSGTYPE_TEXT, SMS_MODE_SEND, &hSms,
 &hSmsEvent);

if(FAILED(hr)) {
    OutputDebugString(TEXT("Could not open a handle to
     the SMS text message service.")));      
    return fReturn;
}

// Wait for SMS to become signaled as ready
DWORD dwReturn = 0;
dwReturn = WaitForSingleObject(hSmsEvent, INFINITE);

// SMS Event has become signaled
if(dwReturn == WAIT_ABANDONED || dwReturn ==
 WAIT_TIMEOUT) {OutputDebugString(TEXT("No longer waiting for
    a message");
    SmsClose(hSms);
    return fReturn;
}

// Send an SMS Message through default SMSC
SMS_ADDRESS smsDestination;
SMS_MESSAGE_ID smsMsgId = 0;

// Set the destination address for the message
memset(&smsDestination, 0, sizeof(SMS_ADDRESS));
smsDestination.smsatAddressType = SMSAT_INTERNATIONAL;
_tcsncpy(smsDestination.ptsAddress, tchPhoneNumber,
        SMS_MAX_ADDRESS_LENGTH);

// Create the message
DWORD dwMessageLength = 0;
dwMessageLength = lstrlen(tchMessage)*sizeof(TCHAR);

// Configure the Text Provider
TEXT_PROVIDER_SPECIFIC_DATA txtProviderData;
DWORD dwProviderLength = 0;
memset(&txtProviderData, 0, sizeof(TEXT_PROVIDER_SPECIFIC_DATA));
txtProviderData.dwMessageOptions = PS_MESSAGE_OPTION_STATUSREPORT;
txtProviderData.psMessageClass = PS_MESSAGE_CLASS0;
txtProviderData.psReplaceOption = PSRO_NONE;
dwProviderLength = sizeof(TEXT_PROVIDER_SPECIFIC_DATA);

// Send the message
hr = SmsSendMessage(hSms, NULL, &smsDestination, NULL,
    (BYTE *)tchMessage, dwMessageLength, (LPBYTE)&txtProviderData,
    dwProviderLength, SMSDE_OPTIMAL, SMS_OPTION_DELIVERY_NONE,
    &smsMsgId);

if(FAILED(hr))
    OutputDebugString(TEXT("Could not send SMS Text Message."));
else {
    OutputDebugString(TEXT("Message has been sent.");
    fReturn = TRUE;
}

SmsClose(hSms);
return fReturn;

Second, use P/Invoke from C# to send an SMS by calling into the wrapper function, as follows:

using System;
using System.Data;
using System.Runtime.InteropServices;
namespace SmsInvokeTest {
    class Class1 {
        // Hook up to wrapper function
        [DllImport("smsinvoke.dll", EntryPoint=
            "SendSMSInvokeMsg", CharSet=CharSet.Unicode,
            SetLastError=true)]
        static extern Int32 SendSmsMessage(string
            stPhoneNumber, string stMessage);

        static void Main(string[] args) {
            // Create a message, and send it via SMS
    
}
string stPhone = "4254432273";
string stMessage = "Hi there from the Compact Framework!";
int nResult = 0;

nResult = SendSmsMessage(stPhone, stMessage);
}  
}  
}