Reading and Writing Isolated Storage

Isolated storage uses streams that work much like ordinary file streams. To obtain an isolated storage stream, you first specify the kind of isolation you want by calling one of the static methods on `IsolatedStorageFile`—as shown previously in Table 14-4. You then use it to construct an `IsolatedStorageFileStream`, along with a filename and `FileMode`:

```
// IsolatedStorage classes live in System.IO.IsolatedStorage

using (IsolatedStorageFile f =
       IsolatedStorageFile.GetMachineStoreForDomain())
using (var s = new IsolatedStorageFileStream ("hi.txt",FileMode.Create,f))
using (var writer = new StreamWriter (s))
  writer.WriteLine ("Hello, World");
```

```
// Read it back:
```

```
using (IsolatedStorageFile f =
       IsolatedStorageFile.GetMachineStoreForDomain())
using (var s = new IsolatedStorageFileStream ("hi.txt", FileMode.Open, f))
using (var reader = new StreamReader (s))
  Console.WriteLine (reader.ReadToEnd());        // Hello, world
```

You can optionally omit the first step, and then the default isolation (Domain User) is used:

```
using (var s = new IsolatedStorageFileStream ("a.txt", FileMode.Create))
using (var writer = new StreamWriter (s))
```

```
using (var s = new IsolatedStorageFileStream ("a.txt", FileMode.Create))
using (var writer = new StreamWriter (s))
  ...
```


`IsolatedStorageFile` is poorly named in that it doesn't represent a file, but rather a *container* for files (basically, a directory).

`IsolatedStorageFile` doesn't provide methods to directly access the roaming stores: you must instead call `GetStore` with an `IsolatedStorageScope` argument. `IsolatedStorageScope` is a flags enum whose members you must combine in exactly the right way to get a valid store. Figure 14-6 lists all the valid combinations.

	Assembly	Assembly & domain
Local user	Assembly | User	Assembly | Domain | User
Roaming user	Assembly | User | Roaming	Assembly | Domain | User | Roaming
Machine	Assembly | Machine	Assembly | Domain | Machine

Roaming user	Assembly | User | Roaming	Assembly | Domain | User | Roaming
Machine	Assembly | Machine	Assembly | Domain | Machine

Figure 14-6. Valid IsolatedStorageScope combinations

Here's how to write to a store isolated by assembly and roaming user:

```
var flags = IsolatedStorageScope.Assembly
          | IsolatedStorageScope.User
          | IsolatedStorageScope.Roaming;
```

```
using (IsolatedStorageFile f = IsolatedStorageFile.GetStore (flags,
                                                             null, null))
using (var s = new IsolatedStorageFileStream ("a.txt", FileMode.Create, f))
using (var writer = new StreamWriter (s))
  writer.WriteLine ("Hello, World");
```

Store Location

Here's where .NET writes isolated storage files:

Here's where .NET writes isolated storage files:

Scope	Location
Local user	[LocalApplicationData]*IsolatedStorage*
Roaming user	[ApplicationData]*IsolatedStorage*
Machine	[CommonApplicationData]*IsolatedStorage*

You can obtain the locations of each of the folders in square brackets by calling the `Environment.GetFolderPath` method. Here are the defaults for Windows Vista and above:

Scope	Location
Local user	*\Users\<user>\AppData\Local\IsolatedStorage*
Roaming user	*\Users\<user>\AppData\Roaming\IsolatedStorage*

Roaming user	*\Users\<user>\AppData\Roaming\IsolatedStorage*
Machine	*\ProgramData\IsolatedStorage*

For Windows XP:

Scope	Location
Local user	*\Documents and Settings\<user>\Local Settings\Application Data\IsolatedStorage*
Roaming user	*\Documents and Settings\<user>\Application Data\IsolatedStorage*
Machine	*\Documents and Settings\All Users\Application Data\IsolatedStorage*

These are merely the base folders; the data files themselves are buried deep in a labyrinth of subdirectories whose names derive from hashed assembly names. This is both a reason to use—and not to use—isolated storage. On the one hand, it makes isolation possible: a permission-restricted application wanting to interfere with another can be stumped by being denied a directory listing—despite having the same filesystem rights as its peers. On the other hand, it makes administration impractical from outside the application. Sometimes it's handy—or essential—to edit an XML

from outside the application. Sometimes it's handy—or essential—to edit an XML configuration file in Notepad so that an application can start up properly. Isolated storage makes this impractical.

Enumerating Isolated Storage

An `IsolatedStorageFile` object also provides methods for listing files in the store:

```
using (IsolatedStorageFile f = IsolatedStorageFile.GetUserStoreForDomain())
{
  using (var s = new IsolatedStorageFileStream ("f1.x",FileMode.Create,f))
    s.WriteByte (123);
```

```
using (var s = new IsolatedStorageFileStream ("f2.x",FileMode.Create,f))
  s.WriteByte (123);

  foreach (string s in f.GetFileNames ("*.*"))
    Console.Write (s + " ");                          // f1.x f2.x
}
```

You can also create and remove subdirectories, as well as files:

```
using (IsolatedStorageFile f = IsolatedStorageFile.GetUserStoreForDomain())
{
  f.CreateDirectory ("subfolder");

 foreach (string s in f.GetDirectoryNames ("*.*"))
```

```
foreach (string s in f.GetDirectoryNames ("*.*"))
  Console.WriteLine (s);

// subfolder

using (var s = new IsolatedStorageFileStream (@"subfolder\sub1.txt",
```



```
    s.WriteByte (100);
  f.DeleteFile (@"subfolder\sub1.txt");
  f.DeleteDirectory ("subfolder");
}

FileMode.Create, f))
```

With sufficient permissions, you can also enumerate over all isolated stores created

With sufficient permissions, you can also enumerate over all isolated stores created by the current user, as well as all machine stores. This function can violate program privacy, but not user privacy. Here's an example:

```
System.Collections.IEnumerator rator =
  IsolatedStorageFile.GetEnumerator (IsolatedStorageScope.User);

while (rator.MoveNext())
{
  var isf = (IsolatedStorageFile) rator.Current;

  Console.WriteLine (isf.AssemblyIdentity);
  Console.WriteLine (isf.CurrentSize);
  Console.WriteLine (isf.Scope);
}

// Strong name or URI
```

```
// Strong name or URI
// User + ...
```

The `GetEnumerator` method is unusual in accepting an argument (this makes its containing class `foreach`-unfriendly). `GetEnumerator` accepts one of three values:

`IsolatedStorageScope.User`
: Enumerates all local stores belonging to the current user

`IsolatedStorageScope.User | IsolatedStorageScope.Roaming`
: Enumerates all roaming stores belonging to the current user

`IsolatedStorageScope.Machine`
: Enumerates all machine stores on the computer

Once you have the `IsolatedStorageFile` object, you can list its content by calling `GetFiles` and `GetDirectories`.

15

Networking

The Framework offers a variety of classes in the `System.Net.*` namespaces for communicating via standard network protocols, such as HTTP, TCP/IP, and FTP. Here's a summary of the key components:

-
-
-
-
-
-

A `WebClient` façade class for simple download/upload operations via HTTP or FTP

`WebRequest` and `WebResponse` classes for more control over client-side HTTP or

`WebRequest` and `WebResponse` classes for more control over client-side HTTP or FTP operations

`HttpListener` for writing an HTTP server

`SmtpClient` for constructing and sending mail messages via SMTP

`Dns` for converting between domain names and addresses

`TcpClient`, `UdpClient`, `TcpListener`, and `Socket` classes for direct access to the transport and network layers

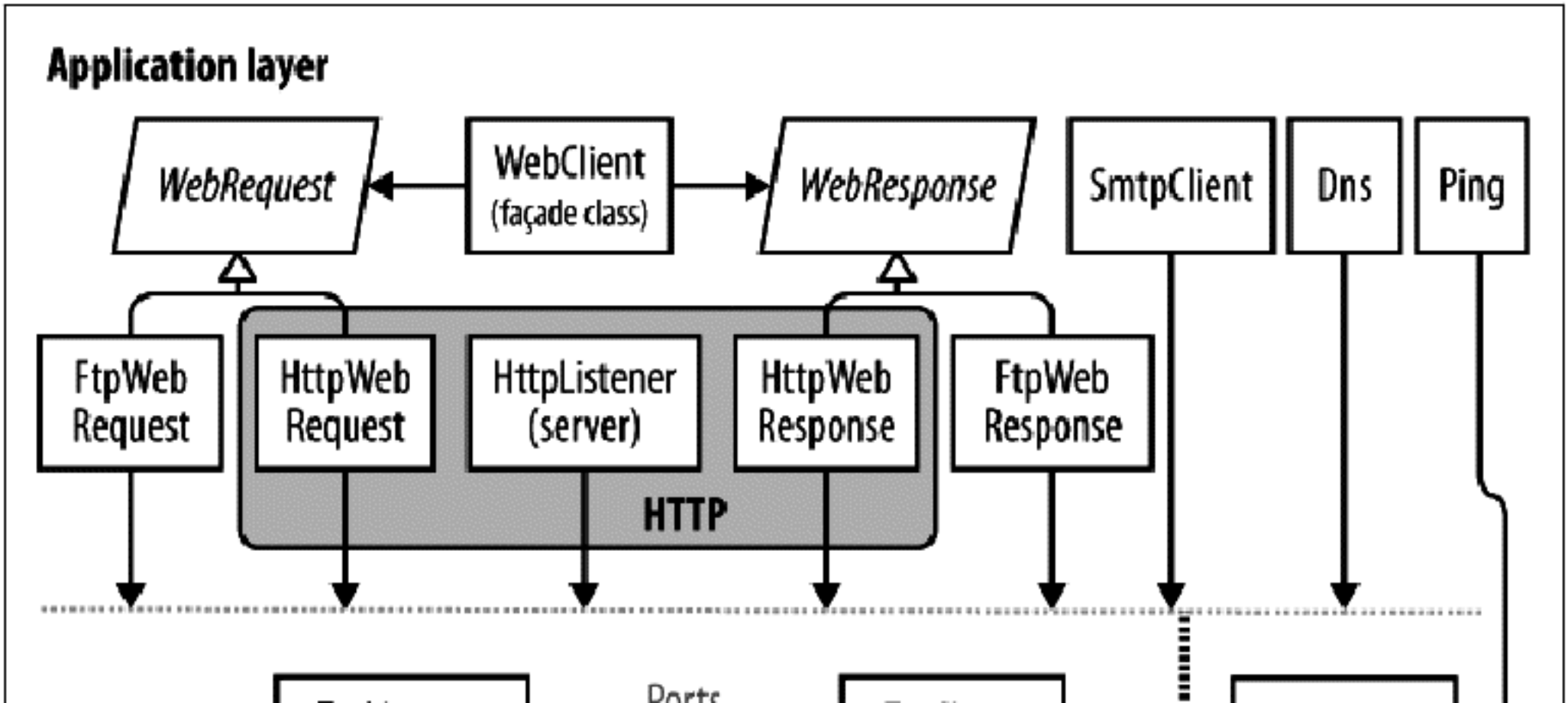
The Framework supports primarily Internet-based protocols, although this doesn't limit applicability to the Internet; protocols such as TCP/IP also dominate local area networks.

The types described in this chapter are defined mostly in the `System.Net` and `System.Net.Sockets` namespaces; however, many of the examples also use types in `System.IO`.

Network Architecture

Figure 15-1 illustrates the .NET networking types and the communication layers in

Figure 15-1 illustrates the .NET networking types and the communication layers in which they reside. Most types reside in the *transport layer* or *application layer*. The transport layer defines basic protocols for sending and receiving bytes (TCP and UDP); the application layer defines higher-level protocols designed for specific applications such as retrieving web pages (HTTP), transferring files (FTP), sending mail (SMTP), and converting between domain names and IP addresses (DNS).



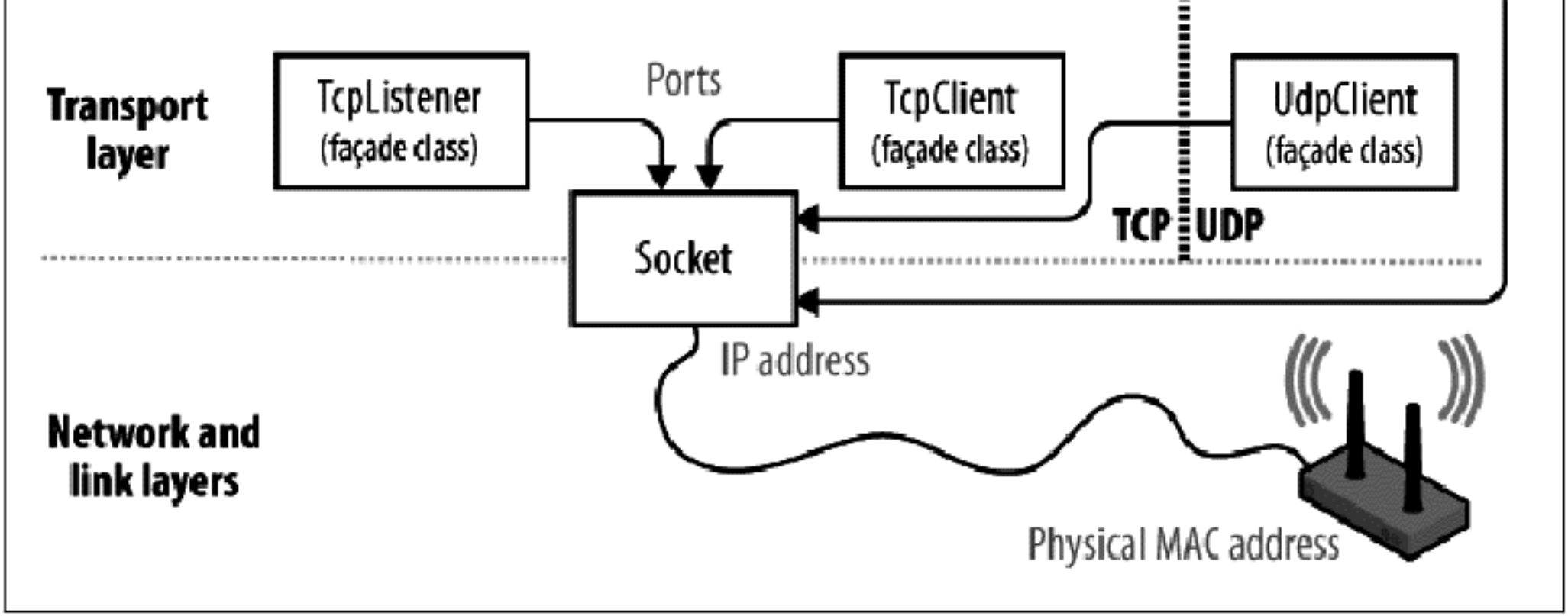


Figure 15-1. Network architecture

It's usually most convenient to program at the application layer; however, there are a couple of reasons you might want to work directly at the transport layer. One is if you need an application protocol not provided in the Framework, such as POP3 for retrieving mail. Another is if you want to invent a custom protocol for a special application, such as a peer-to-peer client.

Of the application protocols, HTTP is special in that its use has extended to general-purpose communication. Its basic mode of operation—"give me the web page with this URL"—adapts nicely to "give me the return value from calling this method with

purpose communication. Its basic mode of operation—"give me the web page with this URL"—adapts nicely to "give me the return value from calling this method with these arguments." HTTP has a rich set of features that are useful in multitier business applications and service-oriented architectures, such as protocols for authentication and encryption, message chunking, extensible headers and cookies, and the ability to have many server applications share a single port and IP address. For these reasons, HTTP is well supported in the Framework—both directly, as described in this chapter, and at a higher level, through such technologies as WCF, Web Services, and ASP.NET.

The Framework provides client-side support for FTP, the popular Internet protocol for sending and receiving files. Server-side support comes in the form of IIS or Unix-based server software.

As the preceding discussion makes clear, networking is a field that is awash in acronyms. Table 15-1 is a handy Network TLA (three-letter and more acronym buster).

Table 15-1. Network TLA (three-letter acronym) buster

Acronym	Expansion	Notes
DNS	Domain Name Service	Converts between domain names (e.g., *ebay.com*) and IP addresses

Acronym	Expansion	Notes
DNS	Domain Name Service	Converts between domain names (e.g., *ebay.com*) and IP addresses (e.g., 199.54.213.2)
FTP	File Transfer Protocol	Internet-based protocol for sending and receiving files

HTTP

IIS
IP
LAN
POP
SMTP
TCP
UDP

Hypertext Transfer Protocol

Hypertext Transfer Protocol

Internet Information Services

Retrieves web pages and runs web services

Microsoft's web server software

Internet Protocol

Local Area Network

Network-layer protocol below TCP and UDP

Most LANs use Internet-based protocols such as TCP/IP

Post Office Protocol

Simple Mail Transfer Protocol

Simple Mail Transfer Protocol

Transmission and Control Protocol

Universal Datagram Protocol

Retrieves Internet mail

Sends Internet mail

Transport-layer Internet protocol on top of which most higher-layer services are built

Transport-layer Internet protocol used for low-overhead services such as VoIP

UNC	Universal Naming Convention	*computer**sharename**filename*

UNC	Universal Naming Convention	*\\computer\sharename\filename*
URI	Uniform Resource Identifier	Ubiquitous resource naming system (e.g., *http://www.amazon.com* or *mailto:joe@bloggs.org*)
URL	Uniform Resource Locator	Technical meaning (fading from use): subset of URI; popular meaning: synonym of URI

Addresses and Ports

For communication to work, a computer or device requires an address. The Internet uses two addressing systems:

IPv4
: Currently the dominant addressing system; IPv4 addresses are 32 bits wide. When string-formatted, IPv4 addresses are written as four dot-separated decimals (e.g., 101.102.103.104). An address can be unique in the world—or unique within a particular *subnet* (such as on a corporate network).

IPv6
: The newer 128-bit addressing system. Addresses are string-formatted in hexadecimal with a colon separator (e.g., [3EA0:FFFF:198A:E4A3:4FF2:54f-A:41BC:8D31]). The .NET Framework requires that you add square brackets

decimal with a colon separator (e.g., [3EA0:FFFF:198A:E4A3:4FF2:54f-A:41BC:8D31]). The .NET Framework requires that you add square brackets around the address.

The `IPAddress` class in the `System.Net` namespace represents an address in either protocol. It has a constructor accepting a byte array, and a static `Parse` method accepting a correctly formatted string:

```
IPAddress a1 = new IPAddress (new byte[] { 101, 102, 103, 104 });
IPAddress a2 = IPAddress.Parse ("101.102.103.104");
```

```
IPAddress a1 = new IPAddress (new byte[] { 101, 102, 103, 104 });
IPAddress a2 = IPAddress.Parse ("101.102.103.104");
Console.WriteLine (a1.Equals (a2));                    // True
Console.WriteLine (a1.AddressFamily);                  // InterNetwork
```



```
IPAddress a3 = IPAddress.Parse
  ("[3EA0:FFFF:198A:E4A3:4FF2:54fA:41BC:8D31]");
Console.WriteLine (a3.AddressFamily);   // InterNetworkV6
```

The TCP and UDP protocols break out each IP address into 65,535 ports, allowing a computer on a single address to run multiple applications, each on its own port. Many applications have standard port assignments; for instance, HTTP uses port 80; SMTP uses port 25.

The TCP and UDP ports from 49152 to 65535 are officially unassigned, so they are good for testing and small-scale deployments.

An IP address and port combination is represented in the .NET Framework by the `IPEndPoint` class:

```
IPAddress a = IPAddress.Parse ("101.102.103.104");
IPEndPoint ep = new IPEndPoint (a, 222);          // Port 222
Console.WriteLine (ep.ToString());                // 101.102.103.104:222
```

Firewalls block ports. In many corporate environments, only a few ports are in fact open—typically, port 80 (for unencrypted HTTP) and port 443 (for secure HTTP).

URIs

A URI is a specially formatted string that describes a resource on the Internet or a LAN, such as a web page, file, or email address. Examples include *http://www.ietf.org*, *ftp://myisp/doc.txt*, and *mailto:joe@bloggs.com*. The exact formatting is defined by the Internet Engineering Task Force (*http://www.ietf.org/*).

by the Internet Engineering Task Force (*http://www.ietf.org/*).

A URI can be broken up into a series of elements—typically, *scheme*, *authority*, and *path*. The `Uri` class in the `System` namespace performs just this division, exposing a property for each element. This is illustrated in Figure 15-2.

The `Uri` class is useful when you need to validate the format of a URI string or to split a URI into its component parts. Otherwise, you can treat a URI simply as a string—most networking methods are overloaded to accept either a `Uri` object or a string.

You can construct a `Uri` object by passing any of the following strings into its constructor:

- A URI string, such as *http://www.ebay.com* or *file://janespc/sharedpics/dolphin.jpg*

-

-
-

An absolute path to a file on your hard disk, such as *c:\myfiles\data.xls*

A UNC path to a file on the LAN, such as *\\janespc\sharedpics\dolphin.jpg*

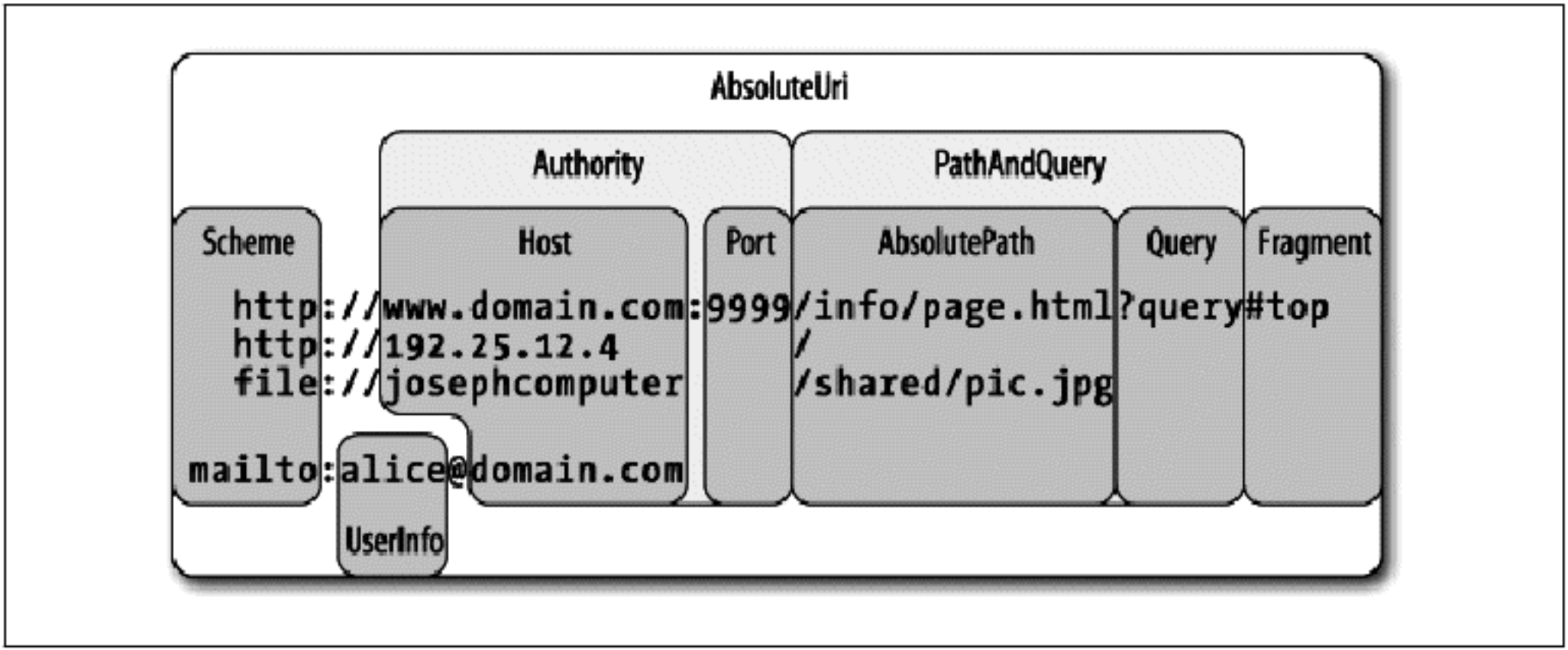


Figure 15-2. URI properties

File and UNC paths are automatically converted to URIs; the "file:" protocol is

File and UNC paths are automatically converted to URIs: the "file:" protocol is added, and backslashes are converted to forward slashes. The `Uri` constructors also perform some basic cleanup on your string before creating the `Uri`, including converting the scheme and hostname to lowercase and removing default and blank port numbers. If you supply a URI string without the scheme, such as "*www.test.com*", a `UriFormatException` is thrown.

`Uri` has an `IsLoopback` property, which indicates whether the `Uri` references the local host (IP address 127.0.0.1), and an `IsFile` property, which indicates whether the `Uri` references a local or UNC (`IsUnc`) path. If `IsFile` returns `true`, the `LocalPath` property returns a version of `AbsolutePath` that is friendly to the local operating system (with backslashes), on which you can call `File.Open`.

Instances of `Uri` have read-only properties. To modify an existing `Uri`, instantiate a `UriBuilder` object—this has writable properties and can be converted back via its `Uri` property.

`Uri` also provides methods for comparing and subtracting paths:

```
Uri info = new Uri ("http://www.domain.com:80/info/");
Uri page = new Uri ("http://www.domain.com/info/page.html");
```

Networking

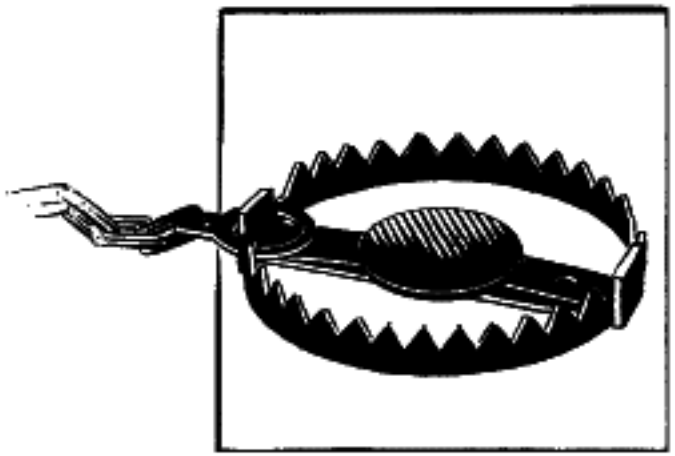
```
// www.domain.com
Console.WriteLine("Host: " + page.Host);
Console.WriteLine("Port: " + page.Port);
Console.WriteLine("Path: " + page.PathAndQuery);
```

```
// www.domain.com
// 80
// 80  (Uri knows the default HTTP port)

Console.WriteLine (info.IsBaseOf (page));
Uri relative = info.MakeRelativeUri (page);
Console.WriteLine (relative.IsAbsoluteUri);
Console.WriteLine (relative.ToString());

// True

// False
// page.html
```

A relative `Uri`, such as *page.html* in this example, will throw an exception if you call

A relative `Uri`, such as *page.html* in this example, will throw an exception if you call almost any property or method other than `IsAbsoluteUri` and `ToString()`. You can instantiate a relative `Uri` directly as follows:


```
Uri u = new Uri ("page.html", UriKind.Relative);
```



A trailing slash is significant in a URI and makes a difference as to how a server processes a request if a path component is present.

For instance, given the URI *http://www.albahari.com/nutshell/*,

For instance, given the URI *http://www.albahari.com/nutshell/*, you can expect an HTTP web server to look in the *nutshell* subdirectory in the site's web folder and return the default document (usually *index.html*).

Without the trailing slash, the web server will instead look for a file called *nutshell* (without an extension) directly in the site's root folder—which is usually not what you want. If no such file exists, most web servers will assume the user mistyped and will return a 301 *Permanent Redirect* error, suggesting the client retries with the trailing slash. A .NET HTTP client, by default, will respond transparently to a 301 in the same way as a web browser—by retrying with the suggested URI. This means that if you omit a trailing slash when it should have been included, your request will still work—but will suffer an unnecessary extra round trip.

The `Uri` class also provides static helper methods such as `EscapeUriString()`, which converts a string to a valid URL by converting all characters with an ASCII value greater than 127 to hexadecimal representation. The `CheckHostName()` and `CheckSchemeName()` methods accept a string and check whether it is syntactically valid for the given property (although they do not attempt to determine whether a host or URI exists).

Request/Response Architecture

`WebRequest` and `WebResponse` are the common base classes for managing both HTTP and FTP client-side activity, as well as the "file:" protocol. They encapsulate the "request/response" model that these protocols all share: the client makes a request, and then awaits a response from a server.

`WebClient` is simply a façade class that does the work of calling `WebRequest` and `WebResponse`, saving you some coding. `WebClient` gives you a choice of dealing in strings, byte arrays, files, or streams; `WebRequest` and `WebResponse` support just streams. Unfortunately, you cannot rely entirely on `WebClient`; some features (such as cookies) are available only through `WebRequest` and `WebResponse`.

as cookies) are available only through `WebRequest` and `WebResponse`.

WebClient

Here are the steps in using `WebClient`:

1. Instantiate a `WebClient` object.
2. Assign the `Proxy` property.
3. Assign the `Credentials` property if authentication is required.

4. Call a `Download`*`XXX`* or `Upload`*`XXX`* method with the desired URI.

Its download methods are as follows:

```
public void   DownloadFile   (string address, string fileName);
public string DownloadString (string address);
public byte[] DownloadData   (string address);
```

```
public string DownloadString (string address);
public byte[] DownloadData   (string address);
public Stream OpenRead       (string address);
```

Each is overloaded to accept a `Uri` object instead of a string address. The upload methods are similar; their return values contain the response (if any) from the server:

```
public byte[] UploadFile  (string address, string fileName);
public byte[] UploadFile  (string address, string method, string fileName);
public string UploadString(string address, string data);
public string UploadString(string address, string method, string data);
public byte[] UploadData  (string address, byte[] data);
public byte[] UploadData  (string address, string method, byte[] data);
public byte[] UploadValues(string address, NameValueCollection data);
public byte[] UploadValues(string address, string method,
                                           NameValueCollection data);
public Stream OpenWrite    (string address);
public Stream OpenWrite    (string address, string method);
```

The `UploadValues` methods can be used to post values to an HTTP form, with a `method` argument of "POST". `WebClient` also has a `BaseAddress` property; this allows you to specify a string to be prefixed to all addresses, such as *http://www.mysite.com/data/*.

Here's how to download the code samples page for this book to a file in the current

Here's how to download the code samples page for this book to a file in the current folder, and then display it in the default web browser:

```
WebClient wc = new WebClient();
wc.Proxy = null;
wc.DownloadFile ("http://www.albahari.com/nutshell/code.aspx", "code.htm");
System.Diagnostics.Process.Start ("code.htm");
```

`WebClient` implements `IDisposable` under *duress*—by virtue of deriving from `Component` (this allows it to be sited in Visual Studio's designer). Its `Dispose` method does nothing useful at run-time, however, so you don't need to dispose `WebClient` instances.

You can use the same `WebClient` object to perform many tasks in sequence. It will crash, however, if you try to make it do two things at once with multithreading. Instead, you must create a separate `WebClient` object for each thread.

WebRequest and WebResponse

`WebRequest` and `WebResponse` are more complex to use than `WebClient`, but also more flexible. Here's how to get started:

1. Call `WebRequest.Create` with a URI to instantiate a web request.

2. Assign the `Proxy` property.
3. Assign the `Credentials` property if authentication is required.

To upload data:

4. Call `GetRequestStream` on the request object, and then write to the stream. Go to step 5 if a response is expected.

To download data:

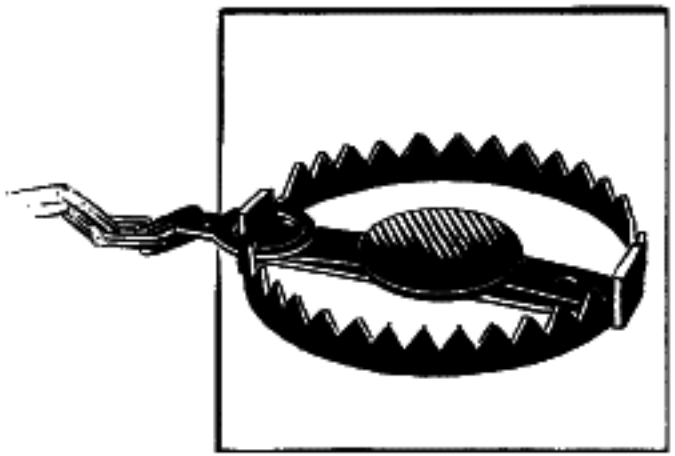
5. Call `GetResponse` on the request object to instantiate a web response.
6. Call `GetResponseStream` on the response object, and then read the stream (a `StreamReader` can help!).

The following downloads and displays the code samples web page (a rewrite of the preceding example):

preceding example):

```
WebRequest req = WebRequest.Create
                 ("http://www.albahari.com/nutshell/code.html");
req.Proxy = null;
using (WebResponse res = req.GetResponse())
using (Stream s = res.GetResponseStream())
using (StreamReader sr = new StreamReader(s))
  File.WriteAllText ("code.html", sr.ReadToEnd());
```

```
System.Diagnostics.Process.Start ("code.html");
```



The web response object has a `ContentLength` property, indi-
cating the length of the response stream in bytes, as reported by

The web response object has a `ContentLength` property, indicating the length of the response stream in bytes, as reported by the server. This value comes from the response headers and may be missing or incorrect. In particular, if an HTTP server chooses the "chunked" mode to break up a large response, the `ContentLength` value is usually –1. The same can apply with dynamically generated pages.

The static `Create` method instantiates a subclass of the `WebRequest` type, such as `HttpWebRequest` or `FtpWebRequest`. Its choice of subclass depends on the URI's prefix, and is shown in Table 15-2.

Table 15-2. URI prefixes and web request types

Prefix	Web request type
http: or https:	`HttpWebRequest`
ftp:	`FtpWebRequest`

ftp:	FtpWebRequest
file:	FileWebRequest

Casting a web request object to its concrete type (`HttpWebRequest` or `FtpWebRequest`) allows you to access its protocol-specific features.

You can also register your own prefixes by calling `WebRequest.RegisterPrefix`. This requires a prefix along with a factory object with a `Create` method that instantiates an appropriate web request object.

The "https:" protocol is for secure (encrypted) HTTP, via Secure Sockets Layer or SSL. Both `WebClient` and `WebRequest` activate SSL transparently upon seeing this prefix (see "SSL" on page 596 under "HTTP-Specific Support" on page 592, later in this chapter). The "file:" protocol simply forwards requests to a `FileStream` object. Its purpose is in meeting a consistent protocol for reading a URI, whether it be a web page, FTP site, or file path.

web page, FTP site, or file path.

`WebRequest` has a `Timeout` property, in milliseconds. If a timeout occurs, a `WebException` is thrown with a `Status` property of `WebExceptionStatus.Timeout`. The default timeout is 100 seconds for HTTP and infinite for FTP.

You cannot recycle a `WebRequest` object for multiple requests—each instance is good for one job only.

Proxies

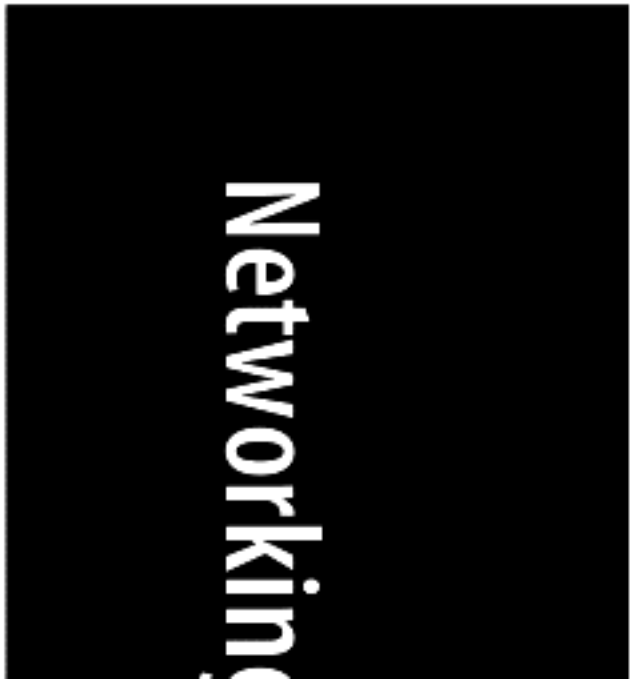
A *proxy server* is an intermediary through which HTTP and FTP requests can be routed. Organizations sometimes set up a proxy server as the only means by which employees can access the Internet—primarily because it simplifies security. A proxy has an address of its own and can demand authentication so that only selected users on the local area network can access the Internet.

You can instruct a `WebClient` or `WebRequest` object to route requests through a proxy server with a `WebProxy` object:

```
// Create a WebProxy with the proxy's IP address and port. You can
// optionally set Credentials if the proxy needs a username/password.
```

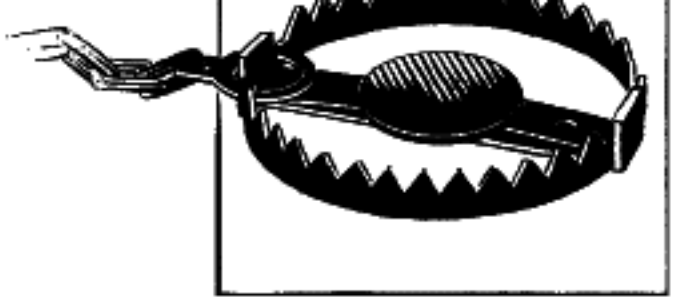
```
    // optionally set Credentials if the proxy needs a username/password.

WebProxy p = new WebProxy ("192.178.10.49", 808);
p.Credentials = new NetworkCredential ("username", "password");
// or:
p.Credentials = new NetworkCredential ("username", "password", "domain");
```

```
WebClient wc = new WebClient();
wc.Proxy = p;
   ...
```



```
// Same procedure with a WebRequest object:
WebRequest req = WebRequest.Create ("...");
req.Proxy = p;
```

If you supply a domain when constructing the `NetworkCredential`, Windows-based authentication protocols are used. To use the currently authenticated Windows user, assign the static `CredentialCache.DefaultNetworkCredentials` value to the proxy's `Credentials` property.



If you don't have a proxy, you must set the `Proxy` property to `null` on all `WebClient` and `WebRequest` objects. Otherwise, the Framework may attempt to "auto-detect" your proxy settings, adding up to 30 seconds to your request. If you're wondering why your web requests execute slowly, this is probably it!

As an alternative to setting the `Proxy` on every `WebClient` and `WebRequest` object, you can set the global default as follows:

```
WebRequest.DefaultWebProxy = myWebProxy;
```

or:

```
WebRequest.DefaultWebProxy = null;
```

Whatever you set applies for the life of the application domain (unless some other code changes it!).

Authentication

You can supply a username and password to an HTTP or FTP site by creating a `NetworkCredential` object and assigning it to the `Credentials` property of `WebClient` or `WebRequest`:

```
WebClient wc = new WebClient();
wc.Proxy = null;
wc.BaseAddress = "ftp://ftp.albahari.com";

// Authenticate, then upload and download a file to the FTP server.
// The same approach also works for HTTP and HTTPS.

string username = "nutshell";
string password = "oreilly";
wc.Credentials = new NetworkCredential (username, password);
```

```
wc.DownloadFile ("guestbook.txt", "guestbook.txt");

string data = "Hello from " + Environment.UserName + "!\r\n";
File.AppendAllText ("guestbook.txt", data);

wc.UploadFile ("guestbook.txt", "guestbook.txt");
```

This works with dialog-based authentication protocols, such as Basic and Digest, and is extensible through the `AuthenticationManager` class. It also supports Windows NTLM and Kerberos (if you include a domain name when constructing the `NetworkCredential` object). If you want to use the currently authenticated Windows user, you can leave the `Credentials` property `null` and instead set `UseDefaultCredentials true`.

Assigning `Credentials` is useless for getting through forms-based authentication. We discuss forms-based authentication separately, in "HTTP-Specific Support" on page 592.

The authentication is ultimately handled by a `WebRequest` subtype (in this case, `FtpWebRequest`), which automatically negotiates a compatible protocol. In the case of HTTP, there can be a choice: if you examine the initial response from a Microsoft Exchange server web mail page, for instance, it might contain the following headers:

```
HTTP/1.1 401 Unauthorized
Content-Length: 83
Content-Type: text/html
Server: Microsoft-IIS/6.0
WWW-Authenticate: Negotiate
WWW-Authenticate: NTLM
WWW-Authenticate: Basic realm="exchange.somedomain.com"
X-Powered-By: ASP.NET
Date: Sat, 05 Aug 2006 12:37:23 GMT
```

The 401 code signals that authorization is required; the "WWW-Authenticate" headers indicate what authentication protocols are understood. If you configure a `WebClient` or `WebRequest` object with the correct username and password, however, this message will be hidden from you because the Framework responds automatically by choosing a compatible authentication protocol, and then resubmitting the original request with an extra header. For example:

cally by choosing a compatible authentication protocol, and then resubmitting the original request with an extra header. For example:

```
Authorization: Negotiate TlRMTVNTUAAABAAAt5II2gjACDArAAACAwACACgAAAAQ
ATmKAAAADolVDRdPUksHUq9VUA==
```

This mechanism provides transparency, but generates an extra round trip with each request. You can avoid the extra round trips on subsequent requests to the same URI by setting the `PreAuthenticate` property to `true`. This property is defined on the `WebRequest` class (and works only in the case of `HttpWebRequest`). `WebClient` doesn't support this feature at all.

CredentialCache

You can force a particular authentication protocol with a `CredentialCache` object. A credential cache contains one or more `NetworkCredential` objects, each keyed to a particular protocol and URI prefix. For example, you might want to avoid the Basic protocol when logging into an Exchange Server, as it transmits passwords in plain text:


```
CredentialCache cache = new CredentialCache();
Uri prefix = new Uri ("http://exchange.somedomain.com");
cache.Add (prefix, "Digest",  new NetworkCredential ("joe", "passwd"));
cache.Add (prefix, "Negotiate", new NetworkCredential ("joe", "passwd"));
```

```
WebClient wc = new WebClient();
wc.Credentials = cache;
```

```
wc.Credentials = cache;
...
```

An authentication protocol is specified as a string. The valid values are as follows:

```
Basic, Digest, NTLM, Kerberos, Negotiate
```

In this particular example, `WebClient` will choose Negotiate, because the server didn't indicate that it supported Digest in its authentication headers. Negotiate is a

Windows protocol that boils down to either Kerberos or NTLM, depending on the capabilities of the server.

The static `CredentialCache.DefaultNetworkCredentials` property allows you to add the currently authenticated Windows user to the credential cache without having to specify a password:

```
cache.Add (prefix, "Negotiate", CredentialCache.DefaultNetworkCredentials);
```

Concurrency

Concurrency

Because communicating across a network can be time-consuming, it makes sense to run `WebClient` or `WebRequest` on a parallel execution path. This allows you to do other things at the same time, and also maintain a responsive user interface. There are a number of ways to achieve parallel execution:

- Create a new thread.
- Use the Task Parallel Library's `Task` class.
- Use asynchronous delegates.
- Use `BackgroundWorker`.

Use BackgroundWorker.

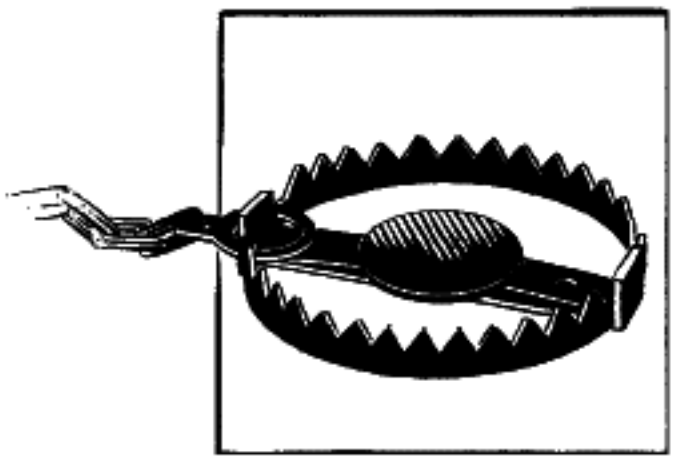
We describe each in Chapter 21. Creating a new thread is simplest, although you must deal with exceptions explicitly on the worker thread:

```
using System;
using System.Net;
using System.Threading;

class ThreadTest
{
  static void Main()
  {
    new Thread (Download).Start();
    Console.WriteLine ("I'm still here while the download's happening!");
    Console.ReadLine();
  }

  static void Download()
  {
```

```
  static void Download()
  {
    WebClient wc = new WebClient();
    try
    {
      wc.Proxy = null;
      wc.DownloadFile ("http://www.oreilly.com", "oreilly.html");
      Console.WriteLine ("Finished!");
    }
    catch (Exception ex)
    {
      // Process exception...
    }
  }
}
```


You can cancel an active `WebClient` operation from another thread by calling `CancelAsync`. (This works whether or not you used an "asynchronous" method to initiate the download or upload.) You can cancel a `WebRequest` in a similar manner,

initiate the download or upload.) You can cancel a `WebRequest` in a similar manner, by calling its `Abort` method from another thread.



Another way to achieve parallel execution is to call the asynchronous event methods on `WebClient` (ending in "Asynch"), such as `DownloadFileAsynch`. In theory, these methods return instantly, allowing the caller to do other things while they run. Unfortunately, these methods are flawed—they block the caller for a portion of the download or upload.

The asynchronous *methods* on `WebRequest` and `WebResponse` (starting in "Begin" and "End") should also be avoided if you're

(starting in "Begin" and "End") should also be avoided if you're simply after parallel execution—these methods serve a subtly different purpose, described in Chapter 23.

When a `WebClient` or `WebRequest` is canceled, a `WebException` is thrown on its thread. The exception has a `Status` property of `WebExceptionStatus.RequestCanceled`. You can catch and deal with this exception just as you would any other, such as an invalid domain name.

Exception Handling

`WebRequest`, `WebResponse`, `WebClient`, and their streams all throw a `WebException` in the case of a network or protocol error. You can determine the specific error via the `WebException`'s `Status` property; this returns a `WebExceptionStatus` enum that has the following members:

`CacheEntryNotFound`

`ConnectFailure`

ConnectFailure
ConnectionClosed
KeepAliveFailure
MessageLengthLimitExceeded
NameResolutionFailure
Pending
PipelineFailure
ProtocolError
ProxyNameResolutionFailure
ReceiveFailure
RequestCanceled

RequestCanceled

SecureChannelFailure

SendFailure

ServerProtocolViolation

Success

Timeout
TrustFailure
UnknownError

- `RequestProhibitedByCachePolicy`
- `RequestProhibitedByProxy`

An invalid domain name causes a `NameResolutionFailure`; a dead network causes a `ConnectFailure`; a request exceeding `WebRequest.Timeout` milliseconds causes a `Timeout`.

Errors such as "Page not found," "Moved Permanently," and "Not Logged In" are specific to the HTTP or FTP protocols, and so are all lumped together under the `ProtocolError` status. To get a more specific code:

1. Cast the `WebException`'s `FtpWebResponse`.

- `Response`

property

to

`HttpWebResponse`

or

2. Examine the response object's `Status` property (an `HttpStatusCode` `FtpStatusCode` enum) and/or its `StatusDescription` property (string).

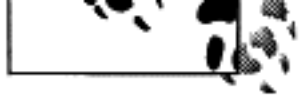
or

For example:

```
WebClient wc = new WebClient();
try
```

```
WebClient wc = new WebClient();
try
{
  wc.Proxy = null;
  string s = wc.DownloadString ("http://www.albahari.com/notthere");
}
catch (WebException ex)
{
  if (ex.Status == WebExceptionStatus.NameResolutionFailure)
    Console.WriteLine ("Bad domain name");
  else if (ex.Status == WebExceptionStatus.ProtocolError)
  {
    HttpWebResponse response = (HttpWebResponse) ex.Response;
    Console.WriteLine (response.StatusDescription);      // "Not Found"
    if (response.StatusCode == HttpStatusCode.NotFound)
      Console.WriteLine ("Not there!");                  // "Not there!"
  }
  else throw;
}
```

If you want the three-digit status code, such as 401 or 404, simply cast the `HttpStatusCode` or `FtpStatusCode` enum to an integer.



integer.

By default, you'll never get a redirection error because `WebClient` and `WebRequest` automatically follow redirection responses. You can switch off this behavior in a `WebRequest` object by setting `AllowAutoRedirect` to `false`.

The redirection errors are 301 (Moved Permanently), 302 (Found/Redirect), and 307 (Temporary Redirect).

If an exception is thrown because you've incorrectly used the `WebClient` or `WebRequest` classes, it will more likely be an `InvalidOperationException` or `ProtocolViolationException` than a `WebException`.

HTTP-Specific Support

This section describes HTTP-specific request and response features.

Headers

Headers

Both `WebClient` and `WebRequest` allow you to add custom HTTP headers, as well as enumerate the headers in a response. A header is simply a key/value pair containing metadata, such as the message content type or server software. Here's how to add a custom header to a request, then list all headers in a response message:


```
WebClient wc = new WebClient();
wc.Proxy = null;
wc.Headers.Add ("CustomHeader", "JustPlaying/1.0");
wc.DownloadString ("http://www.oreilly.com");

foreach (string name in wc.ResponseHeaders.Keys)
  Console.WriteLine (name + "=" + wc.ResponseHeaders [name]);
```

```
Age=51
```

```
Age=31
X-Cache=HIT from oregano.bp
X-Cache-Lookup=HIT from oregano.bp:3128
Connection=keep-alive
Accept-Ranges=bytes
Content-Length=95433
Content-Type=text/html
...
```

Query Strings

A query string is simply a string appended to a URI with a question mark, used to send simple data to the server. You can specify multiple key/value pairs in a query string with the following syntax:

```
?key1=value1&key2=value2&key3=value3...
```

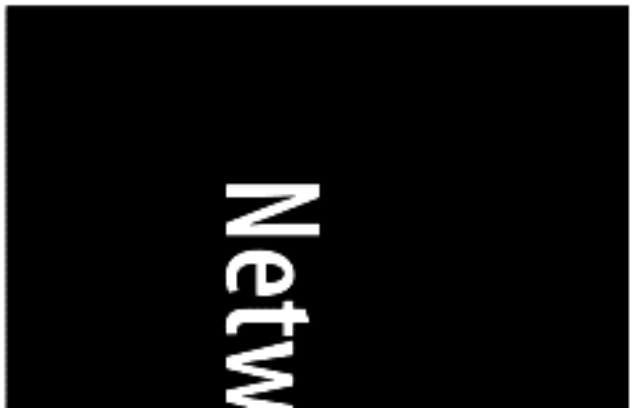
```
?key1=value1&key2=value2&key3=value3...
```

WebClient provides an easy way to add query strings through a dictionary-style property. The following searches Google for the word "WebClient", displaying the result page in French:

```
WebClient wc = new WebClient();
wc.Proxy = null;
wc.QueryString.Add ("q", "WebClient");     // Search for "WebClient"
wc.QueryString.Add ("hl", "fr");           // Display page in French
wc.DownloadFile ("http://www.google.com/search", "results.html");
System.Diagnostics.Process.Start ("results.html");
```

To achieve the same result with WebRequest, you must manually append a correctly formatted string to the request URI:

```
string requestURI = "http://www.google.com/search?q=WebClient&hl=fr";
```



Uploading Form Data

`WebClient` provides `UploadValues` methods for posting data to an HTML form. Here's how to query the Safari website for books containing the term "WebClient":

```
WebClient wc = new WebClient();
wc.Proxy = null;

var data = new System.Collections.Specialized.NameValueCollection();
data.Add ("searchtextbox", "webclient");
data.Add ("searchmode", "simple");
```

```
data.Add ("searchMode", "simple");

byte[] result = wc.UploadValues ("http://my.safaribooksonline.com/search",
                                 "POST", data);
```



```
System.IO.File.WriteAllBytes ("SearchResults.html", result);
System.Diagnostics.Process.Start ("SearchResults.html");
```

The keys in the `NameValueCollection`, such as `searchtextbox` and `searchMode`, correspond to the names of input boxes on the HTML form.

Uploading form data is more work via `WebRequest`. (You'll need to take this route if you need to use features such as cookies.) Here's the procedure:

1. Set the request's `ContentType` to "application/x-www-form-urlencoded" and its `Method` to "POST".
2. Build a string containing the data to upload, encoded as follows:

```
name1=value1&name2=value2&name3=value3...
```

```
name1=value1&name2=value2&name3=value3...
```

3. Convert the string to a byte array, with `Encoding.UTF8.GetBytes`.
4. Set the web request's `ContentLength` property to the byte array length.
5. Call `GetRequestStream` on the web request and write the data array.
6. Call `GetResponse` to read the server's response.

Here's the previous example written with `WebRequest`:

```
WebRequest req = WebRequest.Create ("http://safari.oreilly.com/search");
req.Proxy = null;
req.Method = "POST";
req.ContentType = "application/x-www-form-urlencoded";
string reqString = "searchtextbox=webclient&searchmode=simple";
byte[] reqData = Encoding.UTF8.GetBytes (reqString);
req.ContentLength = reqData.Length;
using (Stream reqStream = req.GetRequestStream())
```

```
using (Stream reqStream = req.GetRequestStream())
  reqStream.Write (reqData, 0, reqData.Length);

using (WebResponse res = req.GetResponse())
using (Stream resSteam = res.GetResponseStream())
using (StreamReader sr = new StreamReader (resSteam))
  File.WriteAllText ("SearchResults.html", sr.ReadToEnd());

System.Diagnostics.Process.Start ("SearchResults.html");
```

Cookies

A cookie is a name/value string pair that an HTTP server sends to a client in a response header. A web browser client typically remembers cookies, and replays them to the server in each subsequent request (to the same address) until their expiry. A cookie allows a server to know whether it's talking to the same client it was a minute ago—or yesterday—without needing a messy query string in the URI.

By default, `HttpWebRequest` ignores any cookies received from the server. To accept

By default, `HttpWebRequest` ignores any cookies received from the server. To accept cookies, create a `CookieContainer` object and assign it to the `WebRequest`. The cookies received in a response can then be enumerated:


```
var cc = new CookieContainer();
```

```
var request = (HttpWebRequest) WebRequest.Create ("http://www.google.com");
request.Proxy = null;
request.CookieContainer = cc;
using (var response = (HttpWebResponse) request.GetResponse())
{
  foreach (Cookie c in response.Cookies)
  {
    Console.WriteLine (" Name:   " + c.Name);
    Console.WriteLine (" Value:  " + c.Value);
    Console.WriteLine (" Path:   " + c.Path);
    Console.WriteLine (" Domain: " + c.Domain);
  }
  // Read response stream...
}
```

```
}
```

```
Name:   PREF
Value:  ID=6b10df1da493a9c4:TM=1179025486:LM=1179025486:S=EJCZri0aWEHlk4tt
Path:   /
Domain: .google.com
```

The `WebClient` façade class does not provide direct support for cookies.

To replay the received cookies in future requests, simply assign the same `CookieContainer` object to each new `WebRequest` object. (`CookieContainer` is serializable, so it can be written to disk—see Chapter 16.) Alternatively, you can start with a fresh `CookieContainer`, and then add cookies manually as follows:

```
Cookie c = new Cookie ("PREF",
                       "ID=6b10df1da493a9c4:TM=1179...",
                       "/",
                       ".google.com");
freshCookieContainer.Add (c);
```

```
freshCookieContainer.Add (c);
```

The third and fourth arguments indicate the path and domain of the originator. A `CookieContainer` on the client can house cookies from many different places; `WebRequest` sends only those cookies whose path and domain match those of the server.

Forms Authentication

Forms Authentication

We saw in the previous section how a `NetworkCredentials` object can satisfy authentication systems such as Basic or NTLM (that pop up a dialog in a web browser). Most websites requiring authentication, however, use some type of forms-based approach. Enter your username and password into text boxes that are part of an HTML form decorated in appropriate corporate graphics, press a button to post the data, and then receive a cookie upon successful authentication. The cookie allows

you greater privileges in browsing pages in the website. With `WebRequest`, you can do all this with the features discussed in the preceding two sections.

A typical website that implements forms authentication will contain HTML like this:

```
<form action="http://www.somesite.com/login" method="post">
  <input type="text" id="user" name="username">
  <input type="password" id="pass" name="password">
  <button type="submit" id="login-btn">Log In</button>
</form>
```

Here's how to log into such a site:

Here's how to log into such a site:

```
string loginUri = "http://www.somesite.com/login";
string username = "username";
string password = "password";
string reqString = "username=" + username + "&password=" + password;
byte[] requestData = Encoding.UTF8.GetBytes (reqString);

CookieContainer cc = new CookieContainer();
var request = (HttpWebRequest)WebRequest.Create (loginUri);
request.Proxy = null;
request.CookieContainer = cc;
request.Method = "POST";

request.ContentType = "application/x-www-form-urlencoded";
request.ContentLength = requestData.Length;
using (Stream s = request.GetRequestStream())
  s.Write (requestData, 0, requestData.Length);

using (var response = (HttpWebResponse) request.GetResponse())
  foreach (Cookie c in response.Cookies)
```

```
  foreach (Cookie c in response.Cookies)
    Console.WriteLine (c.Name + " = " + c.Value);

// We're now logged in. As long as we assign cc to subsequent WebRequest
// objects, we'll be treated as an authenticated user.
```

SSL

Both `WebClient` and `WebRequest` use SSL automatically when you specify an "https:" prefix. The only complication that can arise relates to bad X.509 certificates. If the server's site certificate is invalid in any way (for instance, if it's a test certificate), an exception is thrown when you attempt to communicate. To work around this, you can attach a custom certificate validator to the static `ServicePointManager` class:

```
using System.Net;
using System.Net.Security;
using System.Security.Cryptography.X509Certificates;
...
static void ConfigureSSL()
{
  ServicePointManager.ServerCertificateValidationCallback = CertChecker;
```

```
{
  ServicePointManager.ServerCertificateValidationCallback = CertChecker;
}
```

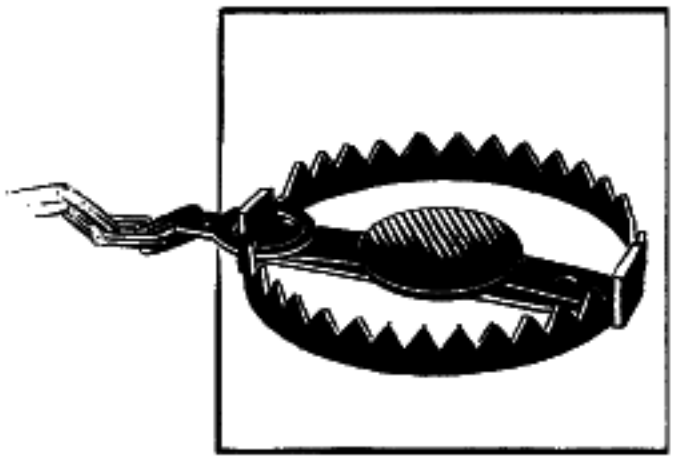
ServerCertificateValidationCallback is a delegate. If it returns true, the certificate is accepted:


```
static bool CertChecker (object sender, X509Certificate certificate,
                         X509Chain chain, SslPolicyErrors errors)
{
  // Return true if you're happy with the certificate
  ...
}
```

Writing an HTTP Server

You can write your own HTTP server with the HttpListener class. The following is a simple server that listens on port 51111, waits for a single client request, and then

You can write your own HTTP server with the `HttpListener` class. The following is a simple server that listens on port 51111, waits for a single client request, and then returns a one-line reply.



`HttpListener` does not work on operating systems prior to Windows XP.

```
static void Main()
{
  new System.Threading.Thread (Listen).Start(); // Run server in parallel.
  Thread.Sleep (500);                           // Wait half a second.

  WebClient wc = new WebClient();               // Make a client request.
  Console.WriteLine (wc.DownloadString
    ("http://localhost:51111/MyApp/Request.txt"));
}
```

```
    ("http://localhost:51111/MyApp/Request.txt"));
}

static void Listen()
{
  HttpListener listener = new HttpListener();
  listener.Prefixes.Add ("http://localhost:51111/MyApp/");
  listener.Start();

  // Listen on
  // port 51111.

 // Wait for a client request:
 HttpListenerContext context = listener.GetContext();

// Respond to the request:
string msg = "You asked for: " + context.Request.RawUrl;
context.Response.ContentLength64 = Encoding.UTF8.GetByteCount (msg);
```

```
context.Response.ContentLength64 = Encoding.UTF8.GetByteCount (msg);
context.Response.StatusCode = (int) HttpStatusCode.OK;
```


```
using (Stream s = context.Response.OutputStream)
using (StreamWriter writer = new StreamWriter (s))
  writer.Write (msg);
```

```
    listener.Stop();
}
OUTPUT: You asked for: /MyApp/Request.txt
```


In this example, we sleep for 500 ms to give the server time to start before connecting to it. A better solution would be for the server to signal that it's ready with an `EventWaitHandle` (described in Chapter 21). An example of when you might consider doing this in real life is if writing a unit testing framework for your HTTP server.

`HttpListener` does not internally use .NET `Socket` objects; it instead calls the Windows HTTP Server API. This is supported on Windows XP and above and allows many applications on a computer to listen on the same IP address and port—as long as each registers different address prefixes. In our example, we registered the prefix *http://localhost/myapp*, so another application would be free to listen on the same

as each registers different address prefixes. In our example, we registered the prefix *http://localhost/myapp*, so another application would be free to listen on the same IP and port on another prefix such as *http://localhost/anotherapp*. This is of value because opening new ports on corporate firewalls can be politically arduous.

`HttpListener` waits for the next client request when you call `GetContext`, returning an object with `Request` and `Response` properties. Each is analogous to a `WebRequest` and `WebResponse` object, but from the server's perspective. You can read and write headers and cookies, for instance, to the request and response objects, much as you would at the client end.

You can choose how fully to support features of the HTTP protocol, based on your anticipated client audience. At a bare minimum, you should set the content length and status code on each request.

Here's a very simple web page server that handles up to 50 concurrent requests:

```
using System;
using System.IO;
using System.Net;
```

```
using System.Net;
using System.Text;
using System.Threading;

class WebServer
{
  HttpListener _listener;
  string _baseFolder;

// Your web page folder.

public WebServer (string uriPrefix, string baseFolder)
{
  System.Threading.ThreadPool.SetMaxThreads (50, 1000);
```

```
{
  System.Threading.ThreadPool.SetMaxThreads (50, 1000);
  System.Threading.ThreadPool.SetMinThreads (50, 50);
  _listener = new HttpListener();
  _listener.Prefixes.Add (uriPrefix);
  _baseFolder = baseFolder;
}
```

```
public void Start()
{
  _listener.Start();
  while (true)
    try
    {
```

```
// Run this on a separate thread, so
```

```
// Run this on a separate thread, as
// we did before.
```



```
  }

  HttpListenerContext request = _listener.GetContext();
  ThreadPool.QueueUserWorkItem (ProcessRequest, request);
}
catch (HttpListenerException)     { break; }   // Listener stopped.
catch (InvalidOperationException) { break; }   // Listener stopped.

public void Stop() { _listener.Stop(); }

void ProcessRequest (object listenerContext)
{
  try
```

```
    try
    {
      var context = (HttpListenerContext) listenerContext;
      string filename = Path.GetFileName (context.Request.RawUrl);
      string path = Path.Combine (_baseFolder, filename);
      byte[] msg;
      if (!File.Exists (path))
      {
        context.Response.StatusCode = (int) HttpStatusCode.NotFound;
        msg = Encoding.UTF8.GetBytes ("Sorry, that page does not exist");
      }
      else
      {
        context.Response.StatusCode = (int) HttpStatusCode.OK;
        msg = File.ReadAllBytes (path);
      }
      context.Response.ContentLength64 = msg.Length;
      using (Stream s = context.Response.OutputStream)
        s.Write (msg, 0, msg.Length);
    }
    catch (Exception ex) { Console.WriteLine ("Request error: " + ex); }
  }
}
```

```
}
```

Here's a main method to set things in motion:

```
static void Main()
{
  // Listen on the default port (80), serving files in e:\mydocs\webroot:
  var server = new WebServer ("http://localhost/", @"e:\mydocs\webroot");
```


```
// Start the server on a parallel thread:
```

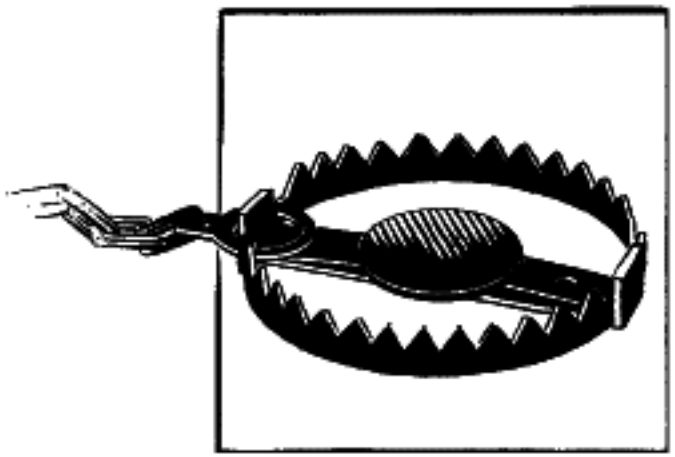
```
// Start the server on a parallel thread:
new System.Threading.Thread (server.Start).Start();

  Console.WriteLine ("Server running... press Enter to stop");
  Console.ReadLine();
  server.Stop();
}
```

You can test this at the client end with any web browser; the URI in this case will be *http://localhost/* plus the name of the web page.

Calling `SetMinThreads` instructs the thread pool not to delay the allocation of threads in an attempt to save memory. This results in a responsive and performant server, up to its limit of 50 requests. If you want to go higher, you can—much higher, and

without needing more threads—by following the asynchronous method pattern. This means calling `BeginRead` and `BeginWrite` on the request and response streams, each time exiting with a callback (bringing the investment in programming time almost on par with configuring IIS!). We describe this in detail in Chapter 23.

each time exiting with a callback (bringing the investment in programming time almost on par with configuring IIS!). We describe this in detail in Chapter 23.



`HttpListener` will not start if other software is competing for the same port (unless that software also uses the Windows HTTP Server API). Examples of applications that might listen on port 80 include a web server or a peer-to-peer program such as Skype.

Using FTP

For simple FTP upload and download operations, you can use `WebClient` as we did previously.

For simple FTP upload and download operations, you can use `WebClient` as we did previously:

```
WebClient wc = new WebClient();
wc.Proxy = null;
wc.Credentials = new NetworkCredential ("nutshell", "oreilly");
wc.BaseAddress = "ftp://ftp.albahari.com";
wc.UploadString ("tempfile.txt", "hello!");
Console.WriteLine (wc.DownloadString ("tempfile.txt"));   // hello!
```

There's more to FTP, however, than just uploading and downloading files. The protocol also lists a set of commands or "methods," defined as string constants in `WebRequestMethods.Ftp`:

```
AppendFile
DeleteFile
DownloadFile
GetDateTimestamp
GetFileSize
```

```
GetFileSize

ListDirectory
ListDirectoryDetails
MakeDirectory
PrintWorkingDirectory
RemoveDirectory

Rename
UploadFile
UploadFileWithUniqueName
```

To run one of these commands, you assign its string constant to the web request's `Method` property, and then call `GetResponse()`. Here's how to get a directory listing:

To run one of these commands, you assign its string constant to the web request's `Method` property, and then call `GetResponse()`. Here's how to get a directory listing:

```
var req = (FtpWebRequest) WebRequest.Create ("ftp://ftp.albahari.com");
req.Proxy = null;
req.Credentials = new NetworkCredential ("nutshell", "oreilly");
req.Method = WebRequestMethods.Ftp.ListDirectory;

using (WebResponse resp = req.GetResponse())
using (StreamReader reader = new StreamReader (resp.GetResponseStream()))
  Console.WriteLine (reader.ReadToEnd());
```

```
RESULT:
.
..
guestbook.txt
tempfile.txt
test.doc
```

In the case of getting a directory listing, we needed to read the response stream to get the result. Most other commands, however, don't require this step. For instance, to get the result of the `GetFileSize` command, just query the response's `ContentLength` property:

```
var req = (FtpWebRequest) WebRequest.Create (
                          "ftp://ftp.albahari.com/tempfile.txt");
req.Proxy = null;
req.Credentials = new NetworkCredential ("nutshell", "oreilly");

req.Method = WebRequestMethods.Ftp.GetFileSize;

using (WebResponse resp = req.GetResponse())
  Console.WriteLine (resp.ContentLength);

// 6
```

The `GetDateTimestamp` command works in a similar way, except that you query the response's `LastModified` property. This requires that you cast to `FtpWebResponse`:

response's LastModified property. This requires that you cast to FtpWebResponse:

```
...
req.Method = WebRequestMethods.Ftp.GetDateTimestamp;
using (var resp = (FtpWebResponse) req.GetResponse() )
  Console.WriteLine (resp.LastModified);
```

To use the Rename command, you must populate the request's RenameTo property with the new filename (without a directory prefix). For example, to rename a file in the *incoming* directory from *tempfile.txt* to *deleteme.txt*:

```
var req = (FtpWebRequest) WebRequest.Create (
                          "ftp://ftp.albahari.com/tempfile.txt");
req.Proxy = null;
req.Credentials = new NetworkCredential ("nutshell", "oreilly");
```

```
req.Method = WebRequestMethods.Ftp.Rename;
req.RenameTo = "deleteme.txt";
```

```
req.GetResponse().Close();
```

```
req.GetResponse().Close();

// Perform the rename
```

Here's how to delete a file:

```
var req = (FtpWebRequest) WebRequest.Create (
                                "ftp://ftp.albahari.com/deleteme.txt");
req.Proxy = null;
req.Credentials = new NetworkCredential ("nutshell", "oreilly");
```

```
req.Method = WebRequestMethods.Ftp.DeleteFile;

req.GetResponse().Close();

// Perform the deletion
```


In all these examples, you would typically use an exception handling block to catch network and protocol errors. A typical `catch` block looks like this:

```
catch (WebException ex)
{
```

```
catch (WebException ex)
{
   if (ex.Status == WebExceptionStatus.ProtocolError)
   {
     // Obtain more detail on error:
     var response = (FtpWebResponse) ex.Response;
     FtpStatusCode errorCode = response.StatusCode;
     string errorMessage = response.StatusDescription;
     ...
   }
   ...
 }
```

Using DNS

The static `Dns` class encapsulates the Domain Name Service, which converts between a raw IP address, such as 66.135.192.87, and a human-friendly domain name, such as *ebay.com*.

The `GetHostAddresses` method converts from domain name to IP address (or

The `GetHostAddresses` method converts from domain name to IP address (or addresses):

```
foreach (IPAddress a in Dns.GetHostAddresses ("albahari.com"))
  Console.WriteLine (a.ToString());     // 208.43.7.176
```

The `GetHostEntry` method goes the other way around, converting from address to domain name:

```
IPHostEntry entry = Dns.GetHostEntry ("208.43.7.176");
Console.WriteLine (entry.HostName);                    // si-eios.com
```

`GetHostEntry` also accepts an `IPAddress` object, so you can specify an IP address as a byte array:

```
IPAddress address = new IPAddress (new byte[] { 208, 43, 7, 176 });
IPHostEntry entry = Dns.GetHostEntry (address);
Console.WriteLine (entry.HostName);                    // si-eios.com
```

Domain names are automatically resolved to IP addresses when you use a class such as `WebRequest` or `TcpClient`. If you plan to make many network requests to the same address over the life of an application, however, you can sometimes improve performance by first using `Dns` to explicitly convert the domain name into an IP address.

address over the life of an application, however, you can sometimes improve performance by first using `Dns` to explicitly convert the domain name into an IP address, and then communicating directly with the IP address from that point on. This avoids repeated round-tripping to resolve the same domain name, and it can be of benefit when dealing at the transport layer (via `TcpClient`, `UdpClient`, or `Socket`).

The DNS class also provides asynchronous methods for high-concurrency applications (see Chapter 23).

Sending Mail with SmtpClient

The `SmtpClient` class in the `System.Net.Mail` namespace allows you to send mail messages through the ubiquitous Simple Mail Transfer Protocol. To send a simple text message, instantiate `SmtpClient`, set its `Host` property to your SMTP server address, and then call `Send`:

```
SmtpClient client = new SmtpClient();
client.Host = "mail.myisp.net";
client.Send ("from@adomain.com", "to@adomain.com", "subject", "body");
```

```
client.Host = "mail.myisp.net";
client.Send ("from@adomain.com", "to@adomain.com", "subject", "body");
```

To frustrate spammers, most SMTP servers on the Internet will accept connections only from the ISP's subscribers, so you need the SMTP address appropriate to the current connection for this to work.

Constructing a `MailMessage` object exposes further options, including the ability to add attachments:

```
SmtpClient client = new SmtpClient();
client.Host = "mail.myisp.net";
MailMessage mm = new MailMessage();
```

```
mm.Sender = new MailAddress ("kay@domain.com", "Kay");
mm.From   = new MailAddress ("kay@domain.com", "Kay");
mm.To.Add  (new MailAddress ("bob@domain.com", "Bob"));
mm.CC.Add  (new MailAddress ("dan@domain.com", "Dan"));
mm.Subject = "Hello!";
```

```
mm.Subject = "Hello!";
mm.Body = "Hi there. Here's the photo!";
mm.IsBodyHtml = false;
mm.Priority = MailPriority.High;
```

```
Attachment a = new Attachment ("photo.jpg",
                               System.Net.Mime.MediaTypeNames.Image.Jpeg);
mm.Attachments.Add (a);
client.Send (mm);
```

`SmtpClient` allows you to specify `Credentials` for servers requiring authentication, `EnableSsl` if supported, and change the TCP `Port` to a nondefault value. By changing the `DeliveryMethod` property, you can instruct the `SmtpClient` to instead use IIS to send mail messages or simply to write each message to an *.eml* file in a specified directory:

```
SmtpClient client = new SmtpClient();
client.DeliveryMethod = SmtpDeliveryMethod.SpecifiedPickupDirectory;
client.PickupDirectoryLocation = @"c:\mail";
```

Using TCP

TCP and UDP constitute the transport layer protocols on top of which most Internet—and local area network—services are built. HTTP, FTP, and SMTP use TCP; DNS uses UDP. TCP is connection-oriented and includes reliability mecha-