Module 14

”Input and Output”

TEKNOLOGISK
INSTITUT

Agenda

- **Using the File System**
- Streams
- Accessing the Web

`FileSystemInfo` Classes

- `FileInfo` and `DirectoryInfo`
  - Contain methods for file and directory access
  - Both derive from `FileSystemInfo`

- There is also a `DriveInfo`
  - This does not derive from `FileSystemInfo`

- These are all ”stateful” file system classes
  - Performs security checks etc. once

The **FileInfo** Class

- Instantiate a **FileInfo** object representing a physical file to manipulate

```
FileInfo fi = new FileInfo( @"C:\Tmp\Demo.log" );
if( fi.Exists && fi.Length > 40)
{
   fi.CopyTo( @"C:\Tmp\DemoBackup.log" );

   fi.Delete();
}
```

The **DirectoryInfo** Class

- Similarly, a **DirectoryInfo** class represents a physical folder in the file system

```
DirectoryInfo di = new DirectoryInfo( @"C:\Tmp" );
if( di.Exists )
{
   Console.WriteLine( "Directory was last accessed: " +
      di.LastAccessTime.ToLongTimeString() );
}
```

- Use
  - DirectoryInfo.GetFiles()
  - DirectoryInfo.GetDirectories()
- Alternatively
  - Use **DirectoryInfo.GetFileSystemInfos()** and process them according to actual type

The **DriveInfo** Class

- Drives are enumerated in a similar manner through **DriveInfo** instances

```
foreach( DriveInfo di in DriveInfo.GetDrives() )
{
   if( di.IsReady )
   {
      Console.WriteLine( "{0} {1} {2} {3} {4} {5}",
         di.Name, di.DriveFormat, di.VolumeLabel,
         di.DriveType, di.TotalSize,
         di.AvailableFreeSpace );
   }
}
```

The **`File`** Class

- Stateless counterpart of **`FileInfo`** class
- Contains static methods manipulating files

```
string filename = @"C:\Tmp\Demo.log";
if( File.Exists( filename ) )
{
   File.Copy( filename, filename + ".old" );
   File.Delete( filename );
}
```

The **`Directory`** Class

- Stateless counterpart of **`DirectoryInfo`** class
- Contains static methods manipulating directories

```
string name = @"C:\Tmp";
if( Directory.Exists( name ) )
{
   DirectoryInfo directory = Directory.GetParent( name );
   Console.WriteLine( directory.FullName );
}
```

The **Path** Class

- Helper class for manipulating file and directory paths

```
if( Path.IsPathRooted( pathName ) == false )
{
   string fullPathName = Path.Combine( @"C:\Tmp", pathName );

   Console.WriteLine( Path.GetDirectoryName( fullPathName ) );
   Console.WriteLine( Path.GetFileName( fullPathName ) );
   Console.WriteLine( Path.GetExtension( fullPathName ) );
}
```

- You should (in principle) always use this!

Agenda

- Using the File System
- Streams
- Accessing the Web

Introducing Streams

- A stream is a sequence of characters or bytes from some data source

- Streams include
  - `FileStream`
  - `NetworkStream`
  - `MemoryStream`
  - `BufferedStream`
  - `IsolatedStorageFileStream`

  - `DeflateStream`
  - `GZipStream`

  - `SslStream`
  - `CryptoStream`
  - …

Stream Methods and Properties

- Stream methods
  - `Seek()`
  - `Read()`
  - `Write()`

- Stream properties
  - `Position`
  - `CanSeek`
  - `CanRead`
  - `CanWrite`

Retrieving a File Stream

- Open a file by specifying
  - `FileMode`
  - `FileAccess`
  - `FileShare`
    - Default is **`Unshared`**

```
using( FileStream fs = File.Open( @"C:\Demo.log",
                                  FileMode.OpenOrCreate,
                                  FileAccess.ReadWrite ) )
{
   // Read or write from or to the stream
   fs.WriteByte( 65 );
   ...
}
```

- `FileInfo` could also be used to open file streams

Readers and Writers

- Idea
  - Stream contents interpreted by high-level reader and writer classes
  - Separates the structure of the data from the transport itself

- Classes
  - `TextReader` and `TextWriter`
    - `StreamReader` and `StreamWriter`
    - `StringReader` and `StringWriter`

  - `BinaryReader` and `BinaryWriter`

Using Readers and Writers

- Use readers and writers on top of **Stream**

```
string input;
using( FileStream fs = File.Open( @"C:\Tmp\Demo.log",
   FileMode.OpenOrCreate, FileAccess.ReadWrite ) )
{
   using( StreamReader sr = new StreamReader( fs ) )
   {
      input = sr.ReadToEnd();
   }
}
```

- **using**-construct ensures everything is closed properly

Compression

- Compression and decompression are facilitated by chaining streams
  - `GZipStream`
  - `DeflateStream`
    - .NET <-> .NET

- Set `CompressionMode`
  - `Compress`
  - `Decompress`

- Create compression stream closest to compressed data

Compression Example

```
using( FileStream inStream = File.OpenRead( @"C:\Tmp\Demo.log" ) )
{
   using( FileStream outStream =
      File.Create( @"C:\Tmp\Demo.log.compressed" ) )
   {
      using( DeflateStream compress =
         new DeflateStream( outStream, CompressionMode.Compress ) )
         {
            for( int i = 0 ; i < inStream.Length ; i++ )
            {
               compress.WriteByte( (byte) inStream.ReadByte() );
            }
         }
      }
   }
}
```

Agenda

- Using the File System
- Streams
- **Accessing the Web**

Web Request and Responses

- Use request-response pattern for **`System.Net`** classes

- `WebRequest` abstract base class
  - `HttpWebRequest`
  - `FtpWebRequest`
  - `FileWebRequest`
- `WebResponse` abstract base class
  - `HttpWebResponse`
  - `FtpWebResponse`
  - `FileWebResponse`

```
HttpWebRequest request = WebRequest.Create( uri ) as HttpWebRequest;
HttpWebResponse response = request.GetResponse() as HttpWebResponse;
```

The **WebClient** Class

- `WebClient` contains very many different methods.

- `WebClient.`

• Download*Xxx*()	Synchronous
• Download*Xxx*Async()	”Traditional” asynchronous
• Download*Xxx*TaskAsync()	Task-based asynchronous
• Upload*Xxx*()	
• Upload*Xxx*Async()	
• Upload*Xxx*TaskAsync()	
• + many overloads and events	

```
using( WebClient client = new WebClient() )
{
   await client.DownloadFileTaskAsync( url1, "1.jpg" );
   string result = await client.DownloadStringTaskAsync( url2 )
}
```

Summary

- Using the File System
- Streams
- Accessing the Web

Question

You are creating an application uploading data using HTML form-based encoding. The application contains a method as follows:

```
public Task<byte[]> Upload( string url, int i, int j )
{
    var client = new WebClient();

}
```

You need to send the integer values i and j as form-encoded values named a and b. Which code segments should be added?

a)
```
var data = string.Format( "a={0}&b={1}", i, j );
return client.UploadStringTaskAsync( new Uri( url ), data );
```

b)
```
var data = string.Format( "a={0}&b={1}", i, j );
return client.UploadFileTaskAsync( new Uri( url ), data );
```

c)
```
var data = string.Format( "a={0}&b={1}", i, j );
return client.UploadDataTaskAsync( url, Encoding.UTF8.GetBytes( data ) );
```

d)
```
var nv = new NameValueCollection {{"a",i.ToString()}, {"b",j.ToString()} };
return client.UploadValuesTaskAsync( new Uri( url ), nv );
```