Capitolo 15
Lavorare con i files e gestire l'I/O

.Net ci mette a disposizione un intero namespace per la gestione dei files e di tutto quanto concerne le problematiche relative al input/output e tale namespace è **System.IO**. Esso contiene dei tipi che ci aiutano nelle operazioni sui files (lettura, scrittura, cancellazione ecc...). System.IO è realmente molto potente e ci permette di eseguire molte operazioni sul file system della macchina su cui la nostra applicazione opera. Per poter esprimere tutta questa potenza il nostro namespace deve contenere un sacco di cose utili... e infatti al suo interno troviamo classi, enumeratori, strutture e delegati. Vediamo il tutto in dettaglio con qualche piccolo esempio, eventualmente, relativo alle classi che, personalmente, ritengo più utili, mentre altre, di maggiore importanza, saranno analizzate a parte:

Classe	Uso
BinaryReader	Legge tipi dati primitivi come valori binari data una specifica codifica.
BinaryWriter	Scrive tipi dati binari in un certo stream di dati e supporta la scrittura di stringhe in una data codifica.

BufferedStream	Aggiunge uno strato di buffering nelle operazioni sia di lettura che di scrittura. Non può essere ereditata. E' utile in casi in cui può essere usata come contenitore temporaneo di dati per altri stream oltre che, ovviamente, per ridurre le operazioni di I/O sullo stream usato. Trattandosi di una classe la creazione è facile:
	
	`FileStream fs = new FileStream ("file.txt",FileMode.Create, FileAccess.ReadWrite);`
	`BufferedStream bs = new BufferedStream(fs);`

Directory	Espone metodi static per una completa gestione delle directory, come vedremo. Non può essere ereditata.
DirectoryInfo	Simile a Directory. Quello, come detto espone metodi statici ed è efficiente quando si devono operare poche istruzioni, al limite una, su di una directory. DirectoryInfo invece può essere usata per creare una istanza della directory ed eseguire più comandi su essa. Vedremo un esempio anche qui.
DirectoryNotFoundException	Rappresenta l'eccezione scatenata se la directory ricercata non viene trovata

DriveInfo	Restituisce varie informazioni su di un drive specificato. Vediamo un esempio che restituisce lo spazio libero su di una unità:
	
	`using System;`
	`using System.IO;`
	`public class Test`
	`{`
	`  public static void Main()`
	`  {`
	`  DriveInfo drive = new DriveInfo("C:");`
	`  Console.WriteLine(drive.TotalFreeSpace);`
	`  }`
	`}`

Drive NotFoundException	E' l'eccezione sollevata quandi si tenta di accedere ad un drive per qualche motivo non disponibile.
EndOfStreamException	Eccezione sollevata se si cerca di leggere oltre la fine dello stream
ErrorEventArgs	Fornisce dati all'evento errore
File	Classe principe per una manipolazione completa dei file esponendo metodi statici. La vedremo in azione.

FileFormatException	Eccezione sollevata a fronte di una formato scorretto in un file
FileInfo	Altra classe per la manipolazione dei files, anche in questo caso c'è la possibilità di usare un riferimento a oggetto
FileLoadException	Si ha questa eccezione quando un file viene trovato ma non può essere caricato
FileNotFoundException	In questo caso il file cercato non esiste su disco
FileStream	Questa classe permette un accesso random a dati strutturati in stream. La vedremo in azione
FileSystemEventArgs	Provvede dati per gli eventi di creazione, cancellazione e modifica relativi ad una directory.
FileSystemInfo	E' la classe base per FileInfo e DirectoryInfo
FileSystemWatcher	Utile classe che tiene d'occhio i cambiamenti in files e directories sollevando gli eventi opportuni.
InternalBufferOverflawException	Eccezione che viene innescata se si superano le dimensioni del buffer interno. Direi autoesplicativa.
InvalidDataException	Sollevata se uno stream ha un formato non valido
IODescriptionAttribute	Sostanzialmente descrive un attributo o un evento in ambito I/O
IOException	Generico errore in fase di I/O
MemoryStream	Interessante classe che crea uno stream che attinge i suoi dati dalla memoria piuttosto che da disco.

Path	Compie operazioni su stringhe contenenti informazioni su files o directory. Il tutto in modo cross-platform, questo è importante. Esempio banale:
	
	`using System;`
	`using System.IO;`
	
	`public class Test`
	`{`
	`  public static void Main()`
	`  {`
	`  string s = @"c:\users\rl\cs\io01.cs";`
	`  Console.WriteLine`
	`  ("Path di {0} e' {1}.", s, Path.GetFullPath(s));`
	`  }`
	`}`

PathTooLongException	Eccezione sollevata quando abbiamo un percorso troppo lungo rispetto al massimo consentito dal sistema. Raro ma con alcuni percorsi di rete può accadere. Il limite è di 260 caratteri sui sistemi Windows, se ben ricordo ☺
PipeException	Eccezione su una data pipe
RenamedEventArgs	Fornisce dati all'evento rename
Stream	E' una vista su una sequenza di byte. Ne riparleremo qui di seguito.
StreamReader	Permette di leggere sequenzialmente da uno stream anche con una qualche particolare codifica. Di questa come delle classi seguenti ne parleremo in questo paragrafo.
StreamWriter	Permette di scrivere sequenzialmente su un certo stream anche con una data codifica
StringReader	Permette di leggere da una stringa
StringWriter	Permette di scrivere su di una stringa
UnmanagedMemoryAccessor	Fornisce un accesso random da parte di codice managed su di un'area di memoria unmanaged
UnmanagedMemoryStream	Fornisce un accesso diretto a memoria unmanaged da parte di codice managed.

Queste sono le classi definite nel namespace, molte le andremo ad usare nel prosieguo. E' definita anche una struct:

Struttura	Uso
WaitForChangedResult	Contiene informazioni su cambiamenti intercorsi.

Abbiamo poi 3 delegati:

Delegato	Uso
ErrorEventHandler	Rappresenta il metodo che gestirà un evento errore dell'oggetto FileSystemWatcher
FileSystemEventHandler	Rappresenta il metodo che gestirà gli eventi Created, Changed e Deleted dell'oggetto FileSystemWatcher.
RenamedEventHandler	Rappresenta il metodo che gestirà l'evento Renamed di FileSystemWatcher.

Infine abbiamo un elenco di utili enum:

Enum	Uso

DriveType	Definisce delle costanti praticamente per ogni tipo di drive reale o virtuale. Ecco una piccola utility, non la migliore, per reperire nome e tipo dei drives del proprio sistema.
	
	`using System;`
	`using System.IO;`
	
	`public class Test`
	`{`
	`public static void Main()`
	`{`
	`DriveInfo[] drives = DriveInfo.GetDrives();`
	`foreach (DriveInfo drive in drives)`
	`{`
	`Console.WriteLine(drive.Name + ' ' + drive.DriveType.ToString());`
	`}`
	`}`
	`}`

FileAccess	Definisce costanti per stabilire l'accesso ad un file in scrittura, lettura o scrittura/lettura.
FileAttributes	Provvede attributi per files e directories
FileMode	Specifica in che modo il sistema operativo deve aprire un dato file.
FileOptions	Presenta ulteriori opzioni per la creazione di un file stream.
FileShare	Contiene costanti di controllo per verificare accessi simultanei sul file.
HandleInheritability	Specifica se un handle è ereditabile.
NotifyFilters	Segnala le variazioni su files o directory
SearchOption	Definisce se effettuare una ricerca su una directory o sulla stessa e tutte le sottodirectory.
SeekOrigin	E' il punto di partenza per le ricerche
WatcherChangeTypes	Sono i cambiamenti che potrebbero occorrere in un file o in ina directory

A questo punto abbiamo un po' di armi in mano e possiamo cercare di affrontare il discorso in maniera organica. Il primo elemento cruciale è la classe **stream**. Questa è scomponibile, da un punto di vista architetturale, in 3 livelli di astrazione:

- Stream adapters
- Decorator streams
- Backing store streams

NB: uso la notazione inglese in quanto più diretta e coerente, francamente in letteratura ho trovato sempre questa, non so se recentemente siano usciti testi in italiano con traduzioni più significative di quelle che saprei dare io.

Gli **stream adapters** sono specifiche istruzioni che supportano le operazioni di lettura e scrittura. Lo scopo è sempre quello di garantire una elevata astrazione. In particolare abbiamo a disposizione:

- StreamReader e StreamWriter dedicati al testo
- BinaryReader e BinaryWriter per i valori binari, ad esempio i valori numerici
- XmlReader e XmlWriter per gestire contenuti XML

Quando parliamo di **decorators streams** invece ci riferiamo a delle classi che possiamo definire intermedie che effettuano una qualche trasformazione sui dati, un esempio tipico è Gzip che ci fornisce gli strumenti per la compressione e la decompressione.

Un **backing store stream** in sostanza possiamo vederlo come il nostro contenitore di dati sia esso una zona di memoria che un archivio su disco, il classico file di testo ad esempio. Evidentemente questo contenitore può essere usato sia per leggervi dei dati sia per scriverci dentro.

Da notare che solo i primi, gli stream adapters, lavorano su dati più complessi dei bytes. Gli altri due strati lavorano unicamente su bytes.

Questa breve introduzione serve a far capire come viene gestito l'I/O tramite la classe stream. Abbiamo delle classi delegate alla lettura/scrittura, altre ad eventuali trasformazioni mentre è bene definito l'ambito (backing store) su cui lavorare. In questo paragrafo ci concentreremo in particolare sui files di testo, in seguito parleremo diffusamente anche di XML. Questo non dimenticando che il concetto di "stream" non è legato rigidamente alle operazioni di I/O sui files ma si tratta di un concetto di portata molto più ampia. Iniziamo quindi con la cosa più semplice possibile: creare un file vuoto

C# Esempio 15.1

```
1  using System;
2  using System.IO;
3
4  public class Test
5  {
6        public static void Main()
7        {
8              Stream s = new FileStream ("test.txt", FileMode.Create);
9        }
10 }
```

Ecco fatto. Questo semplice programma crea un file vuoto di nome test.txt nella stessa directory in cui si trova l'eseguibile stesso. Un primo problema che si può presentare è la mancanza di diritti per l'accesso alla directory, cosa che si rivela, se non si prendono altre precauzioni, a runtime; ecco cosa accade se cerco di scrivere laddove il mio Windows 7 non vuole che un utente normale metta mano:

Unhandled Exception: System.UnauthorizedAccessException: Access to the path 'c:\program files\test.txt' is denied.

Oppure, altro caso tipico, la directory destinazione non esiste, anche qui in runtime ci sgrida:

Unhandled Exception: System.IO.DirectoryNotFoundException: Could not find a part of the path 'c:\pappo\test.txt'.

La riga 8 è, evidentemente, il cuore del programma 15.1, mica per niente l'ho evidenziata in rosso. La nostra attenzione va posta sulla creazione di un nuovo stream tramite la classe **FileStream**, che internamente è definita come

```
public class FileStream : Stream
```

la quale associa appunto un file di testo allo stream ed evidentemente eredita da Stream. FileStream ha diversi costruttori noi abbiamo utilizzato il formato:

```
FileStream(String, FileMode)
```

FileMode in particolare è un enumeratore, come abbiamo visto in precedenza, che ci dice in che modo possiamo aprire il nostro file. I valori ammessi sono i seguenti:

Classe	Uso
CreateNew	Crea un file ex novo restituendo errore se lo stesso esiste già
Create	Crea un nuovo file sovrascrivendo eventualmente quello avente lo stesso nome già esistente.
Open	Apre un file esistente
OpenOrCreate	Apre un file esistente o lo crea ex novo
Truncate	Apre un file e ne riduce le dimensioni a zero
Append	Apre un file e si posiziona alla fine oppure lo crea

Possiamo poi analizzare qualche proprietà del nostro file:

C# **Esempio 15.2**

```
1  using System;
2  using System.IO;
3
4  public class Test
5  {
6        public static void Main()
7        {
8              Stream s = new FileStream ("test.txt", FileMode.Create);
9              Console.WriteLine(s.CanRead);
10             Console.WriteLine(s.CanWrite);
11             Console.WriteLine(s.CanSeek);
12       }
13 }
```

Come suggerisce il loro nome CanRead, CanWrite e CanSeek sono metodi, che restituiscono true o false, che ci dicono se sul file che stiamo manipolando è possibile fare certe operazioni. Normalmente tali metodi si usano come controlli operativi preliminari. Banalmente che CanRead restituisce false allora il file è write-only mentre è read-only se è CanWrite a risultare falso. Bene, adesso sappiamo come creare un file e testare se su di esso possiamo fare quelle che sono le operazioni più comuni. Cominciamo pertanto a scrivere qualche cosa nel nostro file

C# **Esempio 15.3**

```
1  using System;
2  using System.IO;
3
4  public class Test
5  {
6        public static void Main()
7        {
8              Stream s = new FileStream ("test.txt", FileMode.Create);
9              s.WriteByte (48);
10             s.WriteByte (49);
11       }
12 }
```

Eseguendo questo programma infiliamo uno “0” ed un “1” nel nostro file. Non è un gran che ma è un inizio!

Tuttavia, quando parliamo di files di testo, noi vogliamo principalmente saper leggere e scrivere, anche se come avrete capito, c’è più di un modo per fare questa operazione. Ci rivolgiamo pertanto a due classi del nostro namespace, ovvero **StreamReader** e **StreamWriter**. Vediamo l’esempio seguente che presuppone l’esistenza di un file di nome “testo.txt” nella stessa directory in cui verrà eseguito il programma.

C# **Esempio 15.4**

```
1  using System;
2  using System.IO;
3
4  public class Test
5  {
6        public static void Main()
7        {
8              TextReader tr = new StreamReader("testo.txt");
9              while (tr.Peek() > -1)
10             {
11                   Console.WriteLine(tr.ReadLine());
12             }
13       }
14 }
```

La riga 8 ovviamente crea una nuova istanza della classe TextReader attraverso il suo discendente diretto StreamReader. La classe TextReader, come suggerisce il nome, legge sequenzialmente una serie di caratteri. La riga successiva invece fa uso del metodo **Peek** in un certo senso guarda in avanti di una posizione senza effettuare alcuna modifica nel reader o nel carattere letto restituendo un valore intero e precisamente quello corrispondente al carattere letto; se il risultato invece è -1 vuole dire che non vi è più nulla da leggere. La definizione interna è:

```
public virtual int Peek()
```

se dunque Peek ci manda avanti per indicarci se c’è ancora qualche cosa da leggere oppure no, esistono poi le istruzioni **Read** (lettura di un carattere e avanzamento di una posizione) o **ReadLine** (legge una stringa) che si occupano della fase di lettura vera e propria avanzando anche la posizione di lettura (se ci si limita a Peek si resta fermi). Evidentemente è possibile procedere a qualsivoglia elaborazione procedendo riga per riga o carattere per carattere. Da notare che Read restituisce un intero, corrispondente al codice Ascii del carattere letto mentre ReadLine ci restituisce una stringa. Un’altra istruzione che può tornare utile è **ReadBlock** che legge un blocco di caratteri riversandoli in un array di char. La definizione interna è la seguente:

```
public virtual int ReadBlock(
       char[] buffer,
       int index,
       int count
)
```

index è il punto di inizio scrittura nel buffer mentre count ci dice quanti caratteri leggere. Segue un esempio assolutamente di base:

C# Esempio 15.5

```
1  using System;
2  using System.IO;
3
4  public class Test
5  {
6        public static void Main()
7        {
8              TextReader sr = new StreamReader("testo.txt");
9              char[] a = new char[5];
10             {
11                   sr.ReadBlock(a,0,4);
12                   Console.WriteLine(a);
13             }
14       }
15 }
```

Anche interessante è il metodo **ReadToEnd** che legge dalla posizione corrente alla fine restituendo una stringa che contiene quanto letto. Poichè la posizione corrente può essere l'inizio questo significa che è un metodo comodo per impacchettare velocemente un file in una stringa. Ad esempio:

```
string s = sr.ReadToEnd();
```

si può facilmente applicare all'esempio precedente.

La fase di scrittura su di un file è abbastanza simile, anche concettualmente. Vediamo l'esempio più semplice possibile:

C# Esempio 15.6

```
1  using System;
2  using System.IO;
3
4  public class Test
5  {
6        public static void Main()
7        {
8              TextWriter tw = new StreamWriter("stringa.txt");
9              tw.WriteLine("Ciao");
10             tw.Close();
11       }
12 }
```

Anche in questo caso abbiamo definito una classe, stavolta è **TextWriter** e tramite un suo metodo, **WriteLine**, abbiamo scritto sul file "stringa.txt" che viene creato sul momento. E' necessario poi usare l'altro metodo, **Close**, per completare il processo, diversamente il file non riporta nulla. Close è necessario nelle fasi di scrittura e sarebbe ottima norma usarlo per concludere anche le fasi di lettura, pur se in questo caso ci pensa il sistema operativo a chiudere il file al termine di programma. In questo caso siamo ad un livello un po', come dire, primitivo, in quanto il file, come è facile provare, viene rasato completamente del

suo contenuto e rimane scritta solo la stringa inserita dal programma. Un modo un po' pericoloso ed in una analisi poco utile di gestire questo processo anche se lo scopo di scrivere su di un file per via programmatica viene senz'altro raggiunto.

Personalmente sono un fan della classe **File**, sempre appartenente a al namespace System.IO e perfetta per gestire, appunto, i file. Questa classe, per definizione, contiene tutto quanto serve per operare in maniera completa.

Vogliamo creare un file di testo? Ok, allora **File.CreateText** è la risposta.

C# Esempio 15.7

```
1  using System;
2  using System.IO;
3
4  public class Test
5  {
6        public static void Main()
7        {
8              StreamWriter sw = File.CreateText("alfa.txt");
9              sw.Close();
10       }
11
12 }
```

Se vogliamo invece appendere del testo nel nostro file un sistema semplice, tra i tanti, è ricorrere ad **AppendAllText**

C# Esempio 15.8

```
1  using System;
2  using System.IO;
3
4  public class Test
5  {
6        public static void Main()
7        {
8              StreamWriter sw = File.CreateText("alfa.txt");
9              sw.Close();
10             File.AppendAllText("alfa.txt", "prima riga");
11             File.AppendAllText("alfa.txt", "\nseconda riga");
12       }
13 }
```

Questo metodo evidentemente appende in coda al file il cui nome costituisce il primo parametro il testo inserito come secondo parametro. Un paio di metodi importanti della classe file sono **OpenRead** ed **OpenWrite**. Essi permettono di aprire in sola lettura o in sola scrittura un dato file. Cercare di operare in scrittura o lettura su di un file aperto rispettivamente in lettura o scrittura genera una eccezione a runtime. Diversamente si può ad esempio operare come segue, suggerendo di andare a verificare come i bytes, o meglio i caratteri Ascii corrispondenti ai byte indicati, vengono trascritti su di un file già esistente. Si vedrà come sia necessario porre attenzione su quanto si fa, anche se in molti casi il modo di operare, in pratica sovrascrive lasciando intatto il resto del file, può essere molto comodo.

C# Esempio 15.9

```
1  using System;
2  using System.IO;
3
4  public class Test
5  {
6        public static void Main()
```

```
7       {
8             FileStream fs = File.OpenWrite("alfa.txt");
9             byte[] b = {48,49,50};
10            fs.Write(b, 0, 3);
11            fs.Close();
12      }
13}
```

Avrete senza dubbio notato l'introduzione della classe FileStream. Come abbiamo detto essa permette l'accesso a dati strutturati, nel caso di file di testo il concetto è molto chiaro. Come specificato anche su MSDN consente operazioni di lettura sincrone ed asincrone e quindi è un argomento che tornerà quando affronteremo il parallelismo e i thread. Analogo è il discorso relativo alla lettura:

C# Esempio 15.10

```
1  using System;
2  using System.IO;
3
4  public class Test
5  {
6        public static void Main()
7        {
8              FileStream fs = File.OpenRead("alfa.txt");
9              byte[] b = new byte[5];
10             fs.Read(b, 0, b.Length);
11             fs.Close();
12             foreach (byte x in b)
13             Console.Write((char)x + " ");
14       }
15 }
```

Il metodo Read legge e mette quanto letto in un array di byte. La riga 13 mostra la visualizzazione a video dei caratteri corrispondenti, cosa resa possibile grazie ad un cast a char.

Un altro aspetto operativamente molto interessante viene introdotto dal metodo **Seek**. Un primo test che è possibile fare relativamente a questa proprietà è il seguente:

C# Esempio 15.11

```
1  using System;
2  using System.IO;
3
4  public class Test
5  {
6        public static void Main()
7        {
8              FileStream fs = File.OpenRead("alfa.txt");
9              Console.WriteLine(fs.CanSeek);
10       }
11 }
```

CanSeek restituisce un valore (true o false) che indica se lo stream corrente supporta la ricerca tramite il metodo Seek. Questo, in pratica, può essere un test utile prima di procedere. CanSeek è una proprietà ed è definita internamente come

```
public override bool CanSeek { get; }
```

Fatto questo si arriva a considerare il metodo principe, ovvero il nostro Seek:

```
public override long Seek(
         long offset,
         SeekOrigin origin
)
```

Questa è la definizione del metodo. Partiamo subito con un esempio che ci mostrerà la facilità con la quale si può puntare una data zona di un file. Per fare questo basta creare, tanto per stare sul più semplice possibile, un file testuale con questo elegante testo:

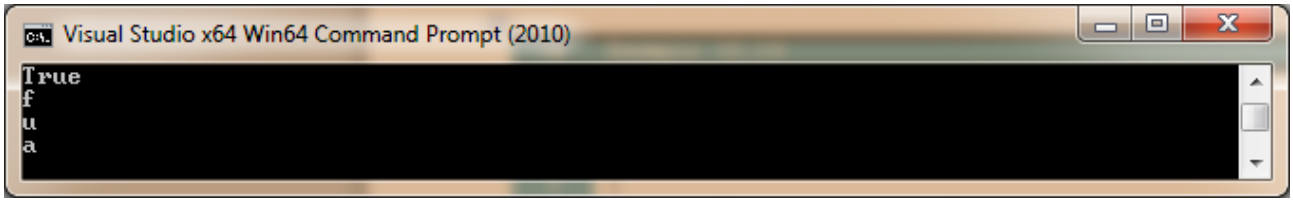
abcdefghilmnopqrstuvz

Facile no? Bene chiamiamolo "beta.txt" e mandiamo in esecuzione il seguente programma:

C# Esempio 15.12

```
1  using System;
2  using System.IO;
3
4  public class Test
5  {
6        public static void Main()
7        {
8              FileStream fs = File.OpenRead("beta.txt");
9              Console.WriteLine(fs.CanSeek);
10             long offset = 5;
11             fs.Seek(offset, SeekOrigin.Begin);
12             Console.WriteLine(Convert.ToChar(fs.ReadByte()));
13             offset = 18;
14             fs.Seek(offset, SeekOrigin.Begin);
15             Console.WriteLine(Convert.ToChar(fs.ReadByte()));
16             offset = 0;
17             fs.Seek(offset, SeekOrigin.Begin);
18             Console.WriteLine(Convert.ToChar(fs.ReadByte()));
19     }
20 }
```

L'output è il seguente:



Capito? Ci spostiamo sul file avanti e indietro senza problemi.

Alla riga 10 definiamo un long, in ossequio alla definizione di Seek appena vista e lo inizializziamo al valore 5. La riga 11 invece introduce il nostro Seek. Al suo interno **SeekOrigin** è un enumerativo con 3 valori definiti: Begin, Current e End che definiscono rispettivamente la posizione iniziale, quella corrente e quella finale.

Il nostro Seek può essere utile anche per scrivere in un certo punto di un file. Quello che è importante sapere è che piazzandoci in una certa posizione noi sovrascriveremo il contenuto in quella posizione e non inseriremo il carattere o i caratteri che andiamo a scrivere. Questo, lo ribadisco va sempre tenuto in mente, per evitare disastri, come si sa la sovrascrittura di un file è sempre complicata da recuperare. Ed ora vediamo un esempio:

C# **Esempio 15.13**

```
1  using System;
2  using System.IO;
3
4  public class Test
5  {
6        public static void Main()
7        {
8              FileStream fs = File.OpenWrite("beta.txt");
9              Console.WriteLine(fs.CanSeek);
10             long offset = 5;
11             fs.Seek(offset, SeekOrigin.Begin);
12             byte[] b1 = new byte[1];
13             b1[0] = 48;
14             fs.Write(b1, 0, 1);
15             fs.Close();
16       }
17 }
```

Il nostro file “beta.txt” che aveva il solito testo visto in precedenza, vede sovrascritto il sesto carattere, la “f”, con lo “0” corrispondente al codice Ascii 48.

Diamo uno sguardo anche ai comandi per la gestione delle directory, almeno a livello di presentazione di base.
Utilizziamo l’oggetto **Directory** che ci consegna alcuni utili comandi , di nostro particolare interesse sono quelli di creazione e cancellazione di una directory. Per il primo esiste anche un oveload che permette di definire alcuni settaggi relativi alla sicurezza ma li vedremo in azione in altra parte.

```
Directory.CreateDirectory(string)
Directory.Delete(string)
```

C# **Esempio 15.14**

```
1  using System;
2  using System.IO;
3
4  public class Test
5  {
6        public static void Main()
7        {
8              Directory.CreateDirectory("d01");
9              Console.WriteLine("Dir creata");
10             Directory.Delete("d01");
11             Console.WriteLine("Dir distrutta");
12       }
13 }
```

Direi che è tutto abbastanza semplice ed autoesplicativo. Evidentemente è necessario avere i corretti diritti per operare ed indicare i percorsi giusti, diversamente saranno sollevate le eccezioni del caso. Ad esempio fornendo un percorso scorretto avremo questa chiara segnalazione:

Unhandled Exception: System.IO.DirectoryNotFoundException:

Oppure se non si hanno i permessi il risultato sarà:

Unhandled Exception: System.UnauthorizedAccessException

Rinominare una directory si fa tramite

```
Directory.Move("origine", "destinazione");
```

Anche qui dati I corretti percorsi e avendo i giusti permessi. Inoltre è necessario che la directory che si vuole rinominare non abbia file in uso (banale, ma a volte ci si dimentica di ciò). Tramite C# è anche possibile in maniera semplice ricavare la lista dei files di una qualsivoglia directory. Usiamo un altro oggetto fornitoci dal framework:

C# Esempio 15.15

```
1  using System;
2  using System.IO;
3
4  public class Test
5  {
6        public static void Main()
7        {
8              DirectoryInfo di = new DirectoryInfo(".");
9              FileInfo[] finfo = di.GetFiles("*.*");
10             foreach(FileInfo fi in finfo)
11             {
12                   Console.WriteLine(fi.Name);
13             }
14       }
15 }
```

Questo semplice programma lista a video tutti i files (*.*) della directory corrente (.). E' possibile modificare il codice per estrarre solo i tipi di files che ci servono, ad esempio se vogliamo solo gli eseguibili possiamo modificare la riga 9 come segue:

```
FileInfo[] finfo = di.GetFiles("*.exe");
```

che crea un array in cui ogni elemento è di tipo "FileInfo"sul quale possiamo facilmente usare l'operatore foreach, come alla riga 10. Attraverso **GetDirectories**, in modo del tutto analogo troveremo l'elenco delle suttodirectory. Il codice presentato è molto semplice ma presenta parecchi spunti. **DirectoryInfo** è una classe che fornisce molti strumenti per lavorare con le directory. Nell'esempio precedente abbiamo utilizzato il metodo GetFiles(string) che restituisce la lista dei files che seguono la regola per i nomi fissata dalla stringa. Ma tante altre informazioni possiamo attingere da questa classe. Nell'esempio che segue vediamo alcune proprietà interessanti:

C# Esempio 15.16

```
1  using System;
2  using System.IO;
3
4  public class Test
5  {
6        public static void Main()
7        {
8              DirectoryInfo di = new DirectoryInfo(".");
9              Console.WriteLine("Data creazione: " + di.CreationTime);
10             Console.WriteLine("Nome completo: " + di.FullName);
11             Console.WriteLine("Ultimo accesso: " + di.LastAccessTime);
12             Console.WriteLine("Ultima op. di scrittura: " + di.LastWriteTime);
13             Console.WriteLine("Livello superiore: " + di.Parent);
14             Console.WriteLine("Root: " + di.Root);
15       }
16 }
```

Mi pare tutto sommato che siano proprietà abbastanza "parlanti". La riga 8, in questo come nell'esempio precedente, crea un handle alla directory corrente. Vi sono anche molti interessanti metodi (abbiamo già

visto GetFiles) come **Create(string)** che crea una nuova directory o **MoveTo** che "muove" la directory in pratica sotto un nuovo nome in pratica rinominandola, analogamente a quanto visto per l'oggetto Directory. Oppure **EnumerateFiles** che sarà utile in ambito Linq. Su MSDN troverete l'elenco completo, in questo momento mi basta che siano chiare le operazioni di base. Le classi Directory e DirectoryInfo sono abbastanza simili e la differenza principale sta nei tipi di ritorno dei vari metodi. Il differenziale relativo all'uso per lo più è dato quindi dal tipo di problema e di interazione con il resto dell'applicazione.

Prima di chiudere diamo uno sguardo ad una classe utile in particolar modo quando si vogliono creare applicazioni di controllo. Stiamo parlando di **FileSystemWatcher**. Il discorso è abbastanza complesso o, quanto meno, articolato dato che si tratta di uno strumento abbastanza sofisticato. Per ora mi limito ad un semplice esempio di base utile per scalfire la superficie.

C# Esempio 15.17

```
1  using System;
2  using System.IO;
3
4  public class Test
5  {
6        public static void Changed(object sender, FileSystemEventArgs e)
7        {
8            Console.WriteLine(e.FullPath.ToString() + " modificato!");
9        }
10
11       public static void Main()
12       {
13             FileSystemWatcher fw = new FileSystemWatcher();
14             fw.Path = @"c:\directory-a-piacere";
15             fw.NotifyFilter=NotifyFilters.LastAccess | NotifyFilters.LastWrite |
16 NotifyFilters.FileName;
17             fw.Changed += new FileSystemEventHandler(Changed);
18             fw.Filter = "*.txt";
19             fw.EnableRaisingEvents = true;
20             Console.WriteLine("Premi 'q' per uscire.");
21             while(Console.Read()!='q');
22       }
23 }
```

Come si vede alla riga 13 definiamo una nuva istanza della classe. Definiamo il path alla riga successiva (se volete testare il codice ovviamente personalizzarlo) e quindi, alla 15, decidiamo quali eventi intercettare. Successivamente ci limitiamo ad implementare l'evento Changed al quale associamo il metodo che inzia alla riga 6. Alla 18 filtriamo i files sui quali deve agire, precisamente quelli con estensione .txt. Per vedere all'opera questa semplice applcazione è sufficiente compilarla e lanciarla, funziona in finestra DOS; essa si metterà in ascolto e se da un file manager creiamo e modifichiamo un files .txt il programma segnalerà questi eventi. Nelle prossime versioni di questo capitolo mi riprometto di approfondire la cosa. Per il momento lascio a chi legge il compito di effettuare qualche ricerca in merito.