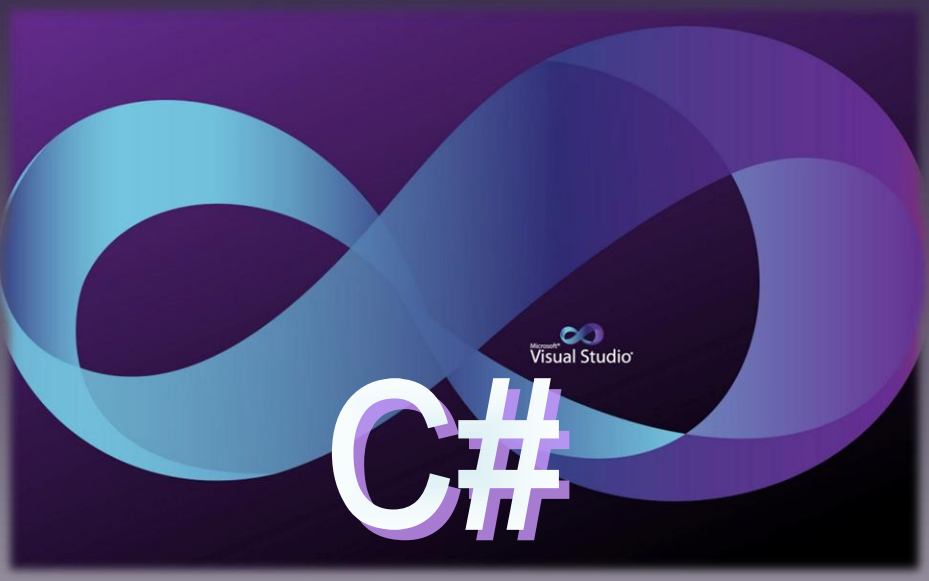
ISTITUTO TECNICO INDUSTRIALE
G. M. ANGIOY
SASSARI



CORSO DI PROGRAMMAZIONE

I FILE E GLI STREAM

DISPENSA 12.01

12-01_Stream_[15]

Versione del: 07/11/2015
Revisione numero: 15

Prof. Andrea Zoccheddu
Dipartimento di Informatica

DIPARTIMENTO
INFORMATICA E TELECOMUNICAZIONI

UNA PANORAMICA SU FILE E STREAM

INTRODUZIONE AGLI STREAM ED AI FILE

CONCETTO DI STREAM

Uno stream (flusso) è un elemento astratto che permette l'interazione (accesso, modifica, gestione) con una generica sequenza di byte (il flusso in senso stretto) prescindendo dalla sua reale natura e provenienza.

Esempi di stream potrebbero essere i flussi di dati che provengono dalla tastiera, che vanno a video, che si scambiano con la scheda di rete e in generale è possibile processare come stream anche un flusso di dati scambiati con una scheda elettronica o una periferica.. Il concetto di stream quindi è slegato da quello di file. Un file è un caso particolare di stream; uno stream può essere anche un file visto come flusso di dati che sono letti da una memoria di massa o che sono scritti su una memoria di massa.

IL FILE DI TESTO

Per file "file di testo" si intende un file il cui contenuto è espresso in forma testuale; di solito è diviso in righe e ammette caratteri speciali per individuare la fine di una riga o del file. Tuttavia è possibile usare il file di testo per memorizzare informazioni relative a qualsiasi programma, come per esempio un archivio di registrazioni (si pensi ad un vettore di strutture memorizzato su file).

IL FILE BINARIO

Il termine "file binario" o "in formato binario" indica un file che non adotta il "formato testo" per la memorizzazione dei dati, che non sono perciò convertiti in forma testuale per essere memorizzati. I dati sono memorizzati trascrivendo direttamente la rappresentazione in memoria (in binario, da cui il nome, ma rappresentata in esadecimale per comodità), rispettando la posizione.

Sono file binari i file eseguibili (estensioni: .exe , .dll , .bin , .com , eccetera), i file di immagini (estensioni: .bmp, .gif, .jpg, .png, eccetera), i file video e audio (video e audio (estensioni: .avi, .mpg, .mp3, .wav, eccetera).

OPERAZIONI

Su stream e file si possono applicare varie operazioni. Le operazioni possono essere suddivise in due grandi categorie:

A) Operazioni di file system
B) Operazioni sul contenuto

Le operazioni di file system sono quelle azioni che hanno il file come oggetto prescindendo dal suo contenuto (quindi è indifferente se il file è un testo, un file eseguibile, un multimediale o altro).

Le operazioni sul contenuto invece sono inerenti la natura del file e hanno effetto sul contenuto del file; quindi è importante stabilire se si tratti di file di testo, binari o altro.

Le operazioni sul contenuto; queste ultime costituiscono la classe di operazioni più importanti per il programmatore.

Vedremo adesso delle operazioni sul contenuto utilizzando il concetto di stream, ma in effetti faremo riferimento ad uno stream che rappresenta un file ovvero un pacchetto di dati memorizzato in memoria di massa. Da questo punto in poi, quindi, si parlerà di file ma sarà sottinteso che l'oggetto in uso è uno stream.

OPERAZIONI DEL FILE SYSTEM

Le operazioni di file system sono quelle azioni che hanno il file come oggetto prescindendo dal suo contenuto (quindi è indifferente se il file è un testo, un file eseguibile, un multimediale o altro). Di seguito sono sinteticamente descritte le possibili operazioni del file system.

"CANCELLAZIONE" DI UN FILE

L'operazione di cancellazione di un file si traduce nella sua eliminazione dal file system. Nella pratica essa non implica alcuna operazione sul contenuto del file, ma semplicemente la sua rimozione, "logica", dall'albero delle directory. Ciò è mostrato dal fatto che alcuni programmi di utilità consentono di recuperare in tutto o in parte il contenuto di un file dopo che questo è stato cancellato.

"COPIA" DI UN FILE

La copia di un file si traduce nella creazione di un nuovo file il cui contenuto è perfettamente identico a quello dell'originale, il quale non viene modificato in alcun modo. Dopo la copia, il file originale e il suo duplicato sono a tutti gli effetti dei file distinti e devono avere nomi diversi se risiedono nella stessa directory.

"RINOMINAZIONE" DI UN FILE

La modifica del nome non implica alcuna operazione sul contenuto o sulle informazioni accessorie del file e riguarda soltanto il nome in senso stretto e non il percorso. Ciò detto, la modifica del nome lascia inalterata la posizione del file all'interno dell'albero delle directory. Tale operazione fallisce se il nuovo nome coincide con il nome di un file già esistente.

"SPOSTAMENTO" DI UN FILE

Lo spostamento implica la copia del file in un'altra posizione dell'albero delle directory, o in un altro dispositivo, e la successiva cancellazione dell'originale. Nello spostare il file è possibile modificarne anche il nome.

Da un certo punto di vista, spostamento e modifica del nome sono operazioni simili, poiché in entrambe il risultato è un file identico all'originale ma con un nome, ed eventualmente un percorso, diversi. D'altra parte, spostando un file nella nuova destinazione è possibile sovrascrivere un file già esistente con lo stesso nome, cosa vietata con la semplice modifica del nome.

OPERAZIONI SUL CONTENUTO

Abbiamo già detto che le operazioni sul contenuto lavorano in base alla natura del file e hanno effetto sul suo contenuto; quindi è importante stabilire se si tratti di file di testo, binari o altro.

Le operazioni sul contenuto costituiscono la classe di operazioni più importanti per il programmatore. Possiamo classificare le operazioni sul contenuto in tre fasi:

1) "**apertura**" di un file esistente o "**creazione**" di un nuovo file;
2) "**lettura**" dei dati contenuti nel file e/o "**scrittura**" di nuovi dati e/o modifica dei dati precedentemente letti;
3) "**chiusura**" del file che spesso coincide col salvataggio dei dati.

La prima fase è necessaria per poter disporre di un flusso con cui operare.

La seconda fase è utilizzata per accedere ai dati del flusso e per inviare dati nuovi.

La terza fase libera il flusso e salva i dati dalla RAM alla memoria di massa.

FASE 1 (APERTURA / CREAZIONE)

Crea una connessione (un legame) tra il file e il programma C# che lo elabora. Immaginare il file come rappresentato in due copie distinte: una fisicamente su Memoria di Massa (es. Hard Disk), l'altro virtualmente su Memoria di Lavoro (RAM).

Apertura

Attenzione: il file deve essere esistente, altrimenti si genera un errore. Quando si "apre" un file si copia il file in memoria di massa nel file virtuale. L'apertura può essere

- In modo esclusivo, impedisce l'accesso a qualsiasi altro programma (il file è bloccato);
- In modo condiviso, permette qualche accesso a seconda dei permessi concessi;

Creazione

Anche la creazione crea un legame. Però il file può non essere esistente prima del comando. Se:

- Il file non c'era: lo crea nuovo;
- Il file c'era già prima: distrugge il preesistente e lo sostituisce con un file "vuoto".

FASE 2 (LETTURA / SCRITTURA)

Attenzione: se **non** si era "aperto" o "creato" il file, ogni tentativo di agire su di esso solleva un errore. Le operazioni agiscono sul file virtualmente su Memoria di Lavoro (RAM).

- La lettura è la copia di una singola informazione dal file verso una locazione del programma;
- La scrittura è viceversa la copia di una informazione dal programma verso il file; la scrittura può avvenire in due modi diversi:
  1. modifica, se i dati vengono sovrascritti a quelli già esistenti; il file NON cambia dimensione ma le informazioni contenute saranno diverse;
  2. aggiunta, se i dati vengono accodati a quelli già esistenti. il file cambia dimensione e vi saranno ulteriori informazioni contenute;

L'operazione di scrittura spesso è "bufferizzata"; ciò significa che i dati non sono immediatamente scritti nella memoria di massa, ma solo nella memoria di lavoro, chiamata "buffer". Quando il buffer è pieno oppure quando viene appositamente richiesto, il contenuto del buffer è trasferito sul file del supporto fisico ed il buffer svuotato. Il motivo principale è che scrivere in RAM è più veloce che in HD e pertanto le scritture sono più veloci; tuttavia vi sarà un momento in cui avviene davvero la scrittura (lenta) su disco.

FASE 3 (CHIUSURA)

Attenzione: se **non** si era "aperto" o "creato" il file, il tentativo di chiuderlo solleva un errore. La connessione tra file della memoria di lavoro e quella di massa è sciolta. Se il file è chiuso non si può agire con operazioni di lettura o scrittura. La chiusura del file è necessaria per:

- svuotare il buffer ed aggiornare eventualmente il file su memoria di massa
- rilasciare il "blocco" del file qualora fosse impedito l'accesso ad altri programmi (vedi Apertura)
- rilasciare le risorse (es. il buffer) impiegate per gestire il file

CONCETTO DI FILE

Un file può essere pensato come una sequenza di blocchi di dati sistemati di seguito nel file system del sistema operativo.

Da un punto di vista hardware il file è allocato su un dispositivo di memoria di massa (es. un hard disk). La memorizzazione fisica del file non è trattata in questo contesto, ma è il driver del controller della memoria di massa che si preoccupa di allocare i blocchi di memoria e di colloquiare col sistema operativo per eseguire le operazioni richieste.

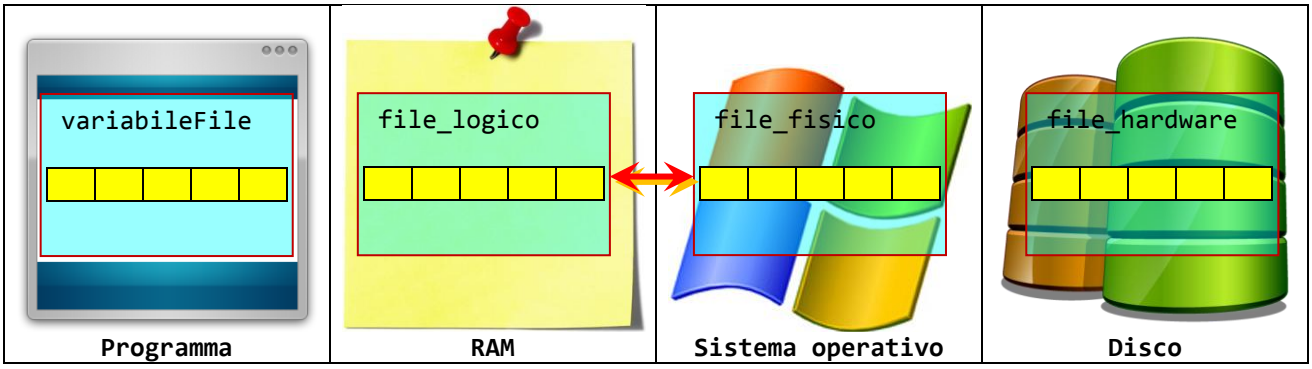
Da un punto di vista del file system (quella parte del sistema operativo che gestisce i file) il file è una sequenza di blocchi logici allocato su un dispositivo di memoria di massa (es. un hard disk). Il file system decide i parametri di memorizzazione dei file (es. l'ampiezza dei blocchi) e colloquia col driver della periferica.

Un programma è in esecuzione quando ha memoria di lavoro (RAM) allocata per lui. Per la gestione dei file occorre distinguere tra lo spazio di memoria allocato dalla applicazione in senso stretto (ad es. le sue variabili) e lo spazio di RAM affidato per svolgere alcune operazioni.

Dal punto di vista del programma avremo quindi una variabile che rappresenta il file come oggetto gestibile dall'applicazione. Questa variabile ha un nome (con le stesse regole dei consueti identificatori) ed è di un tipo riconoscibile come file (vedremo i dettagli in seguito).

Occorre ora notare che, affinché un programma possa processare un file, è necessario che questi stabilisca prima un collegamento con esso. Questo collegamento avviene attraverso un oggetto che si interfaccia con il file system e si connette al file in questione, rendendo possibili le operazioni richieste, creazione, apertura, lettura, eccetera. Questo oggetto è sistemato nella RAM, leggermente distinto dalla variabile file, poiché in realtà è una memoria tampone che conserva temporaneamente alcune informazioni in attesa di salvarle fisicamente.

Nella seguente figura è illustrato lo stesso file visto dai diversi punti di vista.



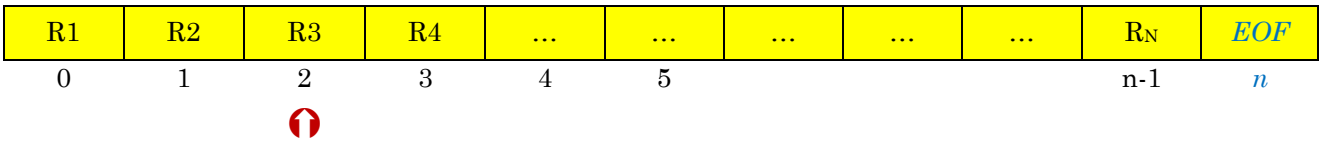
La freccia rossa nella precedente figura indica la connessione che si crea tra il file fisico (del file system) e il file logico (della RAM) all'atto dell'apertura. In Visual C# questa connessione si attua quando la variabile file è associata a un nuovo oggetto di tipo file (o stream) che specifica il nome completo del file (o dello stream). Tipicamente l'istruzione che attua wquesta connessione assume la forma seguente:

```
FileStream fs ;
fs = new FileStream(@"c:\temp\prova.txt", FileMode.Open);
```

dove fs è la variabile di tipo file, il file logico viene creato al momento di esecuzione della operazione new, il file fisico è individuato dal sistema operativo col nome completo **@"c:\temp\prova.txt"** che può essere preceduto da una chiocciola per evitare di male interpretare i caratteri speciali come il backslash.

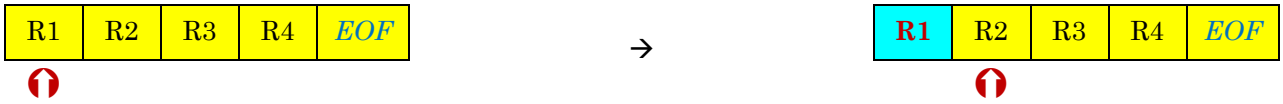
IMMAGINARE UNO STREAM

Lo stream può essere pensato come una sequenza di dati, numerati per posizione, con una testina virtuale (indicatore numerico) che mantiene la posizione corrente. Al momento del'apertura/creazione la testina sarà sullo zero (prima posizione); durante le azioni di scrittura/lettura si sposta. La figura seguente mostra uno stream di N elementi il cui elemento corrente è il numero 2.



La lettura di un file è una copia dei dati dalla posizione corrente dello stream verso il programma; per scrittura si intende una copia dei dati dal programma verso la posizione corrente dello stream. È anche possibile "forzare" il posizionamento della testina imponendo di spostare l'indicatore su una posizione specificata.

LETTURA IN UN FILE



Prima della lettura Dopo la lettura

Quando si legge da uno stream, si supponga che la testina sia su una posizione corrente. Al momento della lettura il contenuto dei dati indicati dalla testina è trascritto in una locazione del programma (per es. una variabile) e la testina si sposta verso la posizione successiva (salvo che il file sia finito).

LETTURA ALLA FINE DEL FILE

Il tentativo di leggere quando la testina è "alla fine" del file solleva un errore.

Prima della lettura Errore in lettura

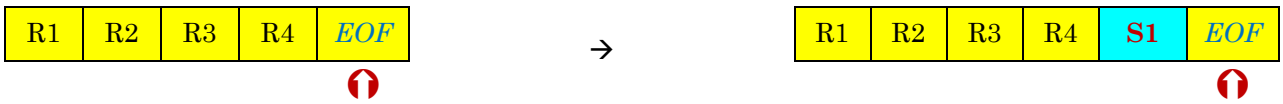
SCRITTURA IN UN FILE

Prima della scrittura Dopo la scrittura

Quando si scrive su uno stream, si supponga che la testina sia su una posizione corrente. Al momento della scrittura il contenuto dei dati del programma (per es. il contenuto di una variabile) è trascritto nella posizione dello stream indicato dalla testina e la testina si sposta verso la posizione successiva.

SCRITTURA ALLA FINE DEL FILE

Se il file è finito lo "allarga" ovvero aumenta di dimensione e crea nuovi dati.



Prima della scrittura Dopo la scrittura

L'OGGETTO STREAM

Tutte le classi esaminate, in questo e nei successivi capitoli, sono definite all'interno del namespace System.IO. Questo significa che per operare sui file e sugli stream occorre esplicitare la direttiva **using System.IO**. Per esempio se crei un nuovo programma in modalità Console, dovrai modificarlo come segue:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;  //osserva questa direttiva: và aggiunta!

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

CREAZIONE DI UN OGGETTO "FILESTREAM"

La connessione tra un oggetto di tipo FileStream e il file fisico è stabilita nel momento della creazione. Per esempio:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
namespace ConsoleApplication1
{
  class Program
  {
    static void Main(string[] args)
    { //scrivi queste due istruzioni!
      FileStream fs ;
      fs = new FileStream(@"c:\temp\prova.txt", FileMode.Open);
    }
  }
}
```

Le due righe possono essere compattate in una sola, dichiarativa e di costruzione. Quindi così:

```
FileStream fs = new FileStream(@"c:\temp\prova.txt", FileMode.Open);
```

In realtà la sintassi completa del costruttore FileStream è la seguente:

```
oggetto-stream = new FileStream ( string      nome,
                                  FileMode    modo,
                                  FileAccess  accesso-opz,
                                  FileShare   condivisione-opz);
```

ovvero il costruttore ha 4 argomenti di cui i primi due obbligatori e gli altri opzionali.

Il **primo argomento** è il nome del file: è una stringa (si antepone il carattere speciale @ per convertire la stringa) che esplicita sia il percorso che il nome del file. Un tipico esempio è:

```
@"C:\miadir\miofile.txt"
```

Perché il simbolo @? Perché la stringa viene valutata come letterale ovvero non si elaborano le sequenze escape (come \n o \r). Altrimenti quando il programma incontra la sequenza \m potrebbe tentare di vederla come un simbolo speciale (a capo, tabulatore, ecc.).

Il **secondo argomento** è il modo con cui connettersi al file. Il modo è una proprietà (statica) della classe FileMode. Sono ammessi i seguenti valori: FileMode.Append , FileMode.Create , FileMode.CreateNew , FileMode.Open , FileMode. FileMode.OpenOrCreate , FileMode.Truncate .

FileMode.Append	Se il file esiste lo apre e si posiziona alla fine. Se il file non esiste lo crea. Impone un metodo di sola scrittura ovvero in combinazione con FileAccess.Write
FileMode.Create	Crea il file. Se c'era un file già esistente il suo contenuto viene cancellato.
FileMode.CreateNew	Crea il file. Se c'era un file già esistente solleva un errore.
FileMode.Open	Apre il file. Se il file non esiste solleva un errore.
FileMode.OpenOrCreate	Apre il file se questo esiste, altrimenti lo crea.
FileMode.Truncate	Apre il file, ne azzera il contenuto e imposta la testina su 0. Se il file non esiste solleva un errore.

Il **terzo argomento** indica quali operazioni sono ammesse sul file. L'Accesso è una proprietà (statica) della classe FileAccess. Sono ammessi i seguenti valori: FileAccess.Read (apertura); FileAccess.Write (scrittura); FileAccess.ReadWrite (entrambe). Il modo di accesso è un valore opzionale: se omesso si presuppone sia FileAccess.ReadWrite.

Il **quarto argomento** indica il livello di condivisione sul file. I valori sono proprietà (statiche) della classe FileShare. Sono ammessi i seguenti valori: FileShare.None (nessuno); FileShare.Read (sola lettura); FileShare.ReadWrite (lettura e scrittura); FileShare.Write (sola scrittura).

FileShare.None	Vieta qualsiasi forma di condivisione. Nessun altro processo può accedere al file. Ricordati di chiuderlo!!!
FileShare.Read	Consente la lettura ad altri, ma non altre operazioni.
FileShare.Write	Consente la scrittura ad altri, ma non altre operazioni.
FileShare.ReadWrite	Consente ad altri processi di leggere e di scrivere il file.

ESEMPI D'USO

ESEMPIO 1. CREAZIONE DI UN NUOVO FILE IN MODALITÀ SOLA SCRITTURA

Per creare un nuovo file pippo.txt in modalità sola scrittura:

```
FileStream fs =
new FileStream (@"pippo.txt", FileMode.CreateNew, FileAccess.Write);
```

OK, purché pippo.txt **NON** esista (se no genera un errore).

ESEMPIO 2. APERTURA DI UN FILE ESISTENTE IN LETTURA

Per aprire il file pippo.txt in sola lettura:

```
FileStream fs =
     new FileStream (@"pippo.txt", FileMode.Open, FileAccess.Read);
```

se pippo.txt non esiste solleva un errore!!!

ESEMPIO 3. APERTURA DI UN FILE ESISTENTE IN SCRITTURA

Per aprire il file pippo.txt in modalità append (aggiunge in coda):

```
FileStream fs =
new FileStream (@"pippo.txt", FileMode.Append);
```

solleva un errore, perché era necessario specificare FileAccess.Write!!!

invece

```
FileStream fs =
new FileStream (@"pippo.txt", FileMode.Append, FileAccess.Write);
```

OK, purché pippo.txt esista!!!

ESEMPIO 4. CREAZIONE DI UN NUOVO FILE IN MODALITÀ SOLA LETTURA

Per creare un nuovo file pippo.txt in modalità sola lettura:

```
FileStream fs =
new FileStream (@"pippo.txt", FileMode.CreateNew, FileAccess.Read);
```

OK, purché pippo.txt **NON** esista (se no genera un errore); ma è poco utile: cosa ci faccio adesso?

ESEMPIO 5. DISTRUZIONE DI FILE

Per distruggere il contenuto del file esistente pippo.txt:

```
FileStream fs = new FileStream (@"pippo.txt", FileMode.Create);
```

OK, funziona sia che pippo.txt esista oppure no. Adesso pippo.txt ha contenuto vuoto.

STREAMREADER E STREAMWRITER

L'oggetto sopra illustrato appare abbastanza farraginoso da usare, anche perché l'operazione di scrittura si aspetta che passiamo come dati da scrivere un array di byte, abbastanza differente dalle consuete stringhe di testo cui siamo abituati.

Invece del FileStream possiamo allora usare altri due oggetti di tipo stream utili per lavorare coi file di testo. Questi sono lo StreamReader e lo StreamWriter.

Anche questi due oggetti sono definiti all'interno del namespace System.IO, per cui occorre scrivere la direttiva **using System.IO**.

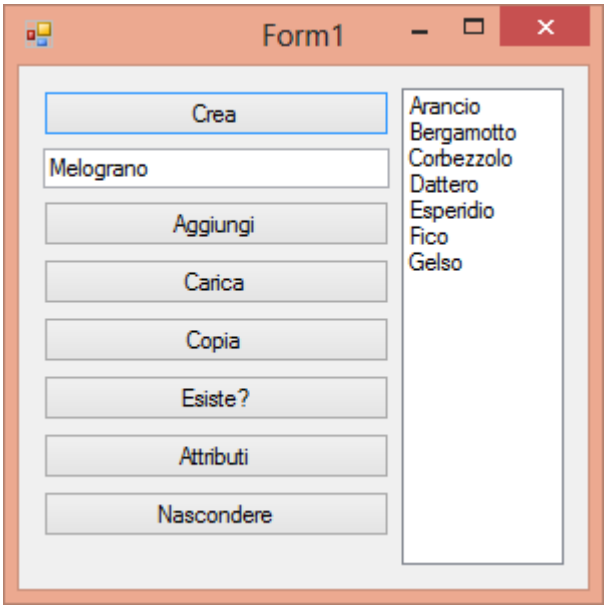
Lo StreamReader è un oggetto rivolto alla sola lettura di un file, mentre l'oggetto StreamWriter è usato per la sola scrittura del file.

DICHIARAZIONI DI STREAMREADER E STREAMWRITER

```
StreamReader sr ;
sr = new StreamReader (@"c:\temp\prova.txt", FileMode.Open);
StreamWriter sw ;
wr = new StreamWriter (@"c:\temp\altro.txt", FileMode.Open);
```

PROGETTO GUIDATO SU STREAMREADER E STREAMWRITER

- prepara un form1 simile alla figura
- la listbox può essere preparata dal seguente metodo FormLoad:




```
private void Form1_Load
      (object sender, EventArgs e)
{
listBox1.Items.Clear();
listBox1.Items.Add("Arancio");
listBox1.Items.Add("Bergamotto");
listBox1.Items.Add("Corbezzolo");
listBox1.Items.Add("Dattero");
listBox1.Items.Add("Esperidio");
listBox1.Items.Add("Fico");
listBox1.Items.Add("Gelso");
}
```

- dichiara il seguente ambiente globale:

```
//dichiaro le variabili file
StreamReader sr;
StreamWriter sw;
//preparo la stringa col path del file
string nome = @"C:\Users\Public\Documents\miofile.txt";
```

- associa al pulsante **button1** (Crea) il seguente gestore di evento:

```
//creo e apro il file sw in modalità scrittura distruttiva
sw = new StreamWriter(nome, false);
foreach (string s in listBox1.Items)
{
     //scrivo una linea di dati nel file sw
     sw.WriteLine(s);
}
//salvo i dati e chiudo il file
sw.Close();
MessageBox.Show("file salvato!");
```

- associa al pulsante **button2** (Aggiungi) il seguente gestore di evento:

```
//apro il file sw in modalità scrittura aggiuntiva
sw = new StreamWriter(nome, true);
string s = textBox1.Text;
sw.WriteLine(s);
sw.Close();
MessageBox.Show("file modificato!");
```

- associa al pulsante **button3** (Carica) il seguente gestore di evento:

```
listBox1.Items.Clear();
sr = new StreamReader(nome, false);
while (!sr.EndOfStream)
{
     string s = sr.ReadLine();
     listBox1.Items.Add(s);
}
sr.Close();
MessageBox.Show("file caricato!");
```

- associa al pulsante **button4** (Copia) il seguente gestore di evento:

```
sr = new StreamReader(nome, false);
string copia = @"C:\Users\Public\Documents\altrofile.txt";
sw = new StreamWriter(copia, true);
while (!sr.EndOfStream)
{
     string riga = sr.ReadLine();
     sw.WriteLine(riga);
}
sr.Close();
sw.Close();
MessageBox.Show("file copiato!");
```

- associa al pulsante **button5** (Esiste?) il seguente gestore di evento:

```
private void button5_Click_1(object sender, EventArgs e)
{
      //verifica se il file è presente
      if (File.Exists(nome))
            MessageBox.Show(nome + " è presente");
      else
            MessageBox.Show(nome + " NON è presente");
}
```

- associa al pulsante **button6** (Nascondere) il seguente gestore di evento:

```
if (File.GetAttributes(nome) != FileAttributes.Hidden)
File.SetAttributes(nome, File.GetAttributes(nome) | FileAttributes.Hidden);
```

- prova a eseguire il progetto

USARE STREAMREADER E STREAMWRITER

Visual Studio permette l'uso di due significativi oggetti: lo **StreamReader** e lo **StreamWriter**. Il primo serve per accedere al file in lettura, il secondo in lettura. Le dichiarazioni rispettive degli **stream** sono:

```
//dichiaro le variabili file
StreamReader sr;
StreamWriter sw;
```

dopo averli dichiarati è possibile aprire gli **stream** mediante l'operatore **new** e l'invocazione del costruttore.

L'oggetto di tipo `StreamReader` può essere aperto nel seguente modo:

```
sr = new StreamReader(nome, false);
```

dove nome è una locazione che contiene la stringa che rappresenta il **path** completo del file.

L'oggetto di tipo `StreamWriter` può essere aperto nel seguente modo:

```
sw = new StreamWriter(nome, false);
```

dove nome è una locazione che contiene la stringa che rappresenta il **path** completo del file, ma il secondo parametro (booleano) indica se il file è aperto in modalità distruttiva (si crea un file nuovo) oppure conservativa (si scrive alla fine del file, appendendo dati aggiuntivi).

Se il secondo parametro è true allora il file è distrutto e creato ex novo, eliminando i dati preesistenti; se invece l'oggetto di tipo `StreamWriter` è aperto nel seguente modo:

```
sw = new StreamWriter(copia, true);
```

i dati preesistenti sono conservati e si scrive alla fine del file.

Su un file StreamWriter è possibile scrivere mediante i metodi Write e WriteLine:

```
      sw.WriteLine(s);
```

il metodo Write scrive senza aggiungere un carattere speciale EndOfLine (fine linea ovvero un a_capo), mentre il metodo WriteLine scrive una linea di testo e poi un carattere speciale EndOfLine (fine linea ovvero un a_capo).

Da un file StreamReader è possibile leggere dati mediante i metodi Read e ReadLine:

```
int carattere = sr.Read();
string riga = sr.ReadLine();
```

il metodo Read legge un carattere e rende un codice ascii corrispondente, mentre il metodo ReadLine legge una linea di testo fino al carattere speciale EndOfLine (fine linea ovvero un a_capo).

Per verificare se il record corrente (la testina) è posizionata sulla fine del file è possibile usare la proprietà `EndOfStream` che rende true se il record corrente è alla fine del file, altrimenti rende false. Di solito questa proprietà serve per leggere fino alla fine del file nel seguente modo:

```
while (!sr.EndOfStream)
    MessageBox.Show(sr.ReadLine());
```

Per rilasciare il file e salvare le eventuali modifiche si deve usare il metodo Close che rimuove il collegamento tra file fisico e file logico, impendendo ulteriori usi del file ma consentendo l'uso ad altre applicazioni:

```
sr.Close();
sw.Close();
```

La chiusura di uno StreamWriter esegue il salvataggio delle modifiche compiute sul file, che altrimenti potrebbero andare perdute.

Prima di aprire o di creare un file potrebbe essere opportuno verificare se è preesistente; se si desidera aprire un file, per esempio, conviene verificare se esiste e in caso contrario prevenire un errore di I/O. Il metodo per verificare la presenza di un file si richiama dalla classe File direttamente, passando il nome del file come parametro:

```
bool verifica = File.Exists(nome);
```

oppure usando un if come nel precedente esercizio guidato.

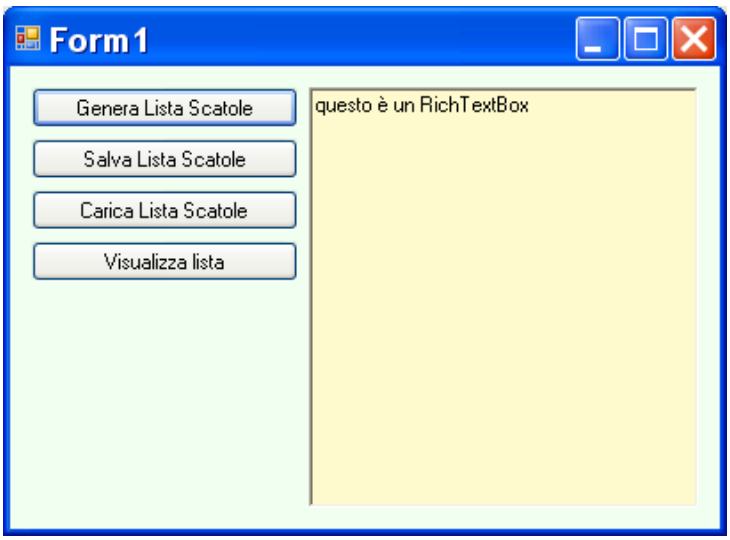
ESERCIZI

ESERCIZI SU FILE E LISTE

ESERCIZIO GUIDATO

PASSO 1 INIZIO

1. Prepara un nuovo progetto visuale come nella figura



2. Vai nell'editor del programma e modifica le librerie come segue:

```
using System.Collections;
// per usare le liste
using System.IO;
//per usare i file
```

3. Poi dentro la dichiarazione della classe Form1 inizia a dichiarare la struttura Scatola:

```
    public partial class Form1 : Form
    {
        //dichiarazioni globali
        Random d = new Random();
        public struct Scatola
        {
            public int altezza, larghezza, profondità;
            public Scatola(int a, int l, int p)
            {
                altezza = a; larghezza = l; profondità = p;
            }
        }
```

4. Subito dopo la Scatola puoi dichiarare le seguenti variabili globali:

```
        //lista di Scatole
        List<Scatola> magazzino = new List<Scatola>();
        const string nomeFile = @"C:\prova.txt";
```

5. A seguire dichiara il seguente metodo:

```
//---metodo ---
public void GeneraScatola(out Scatola s)
{
    s.altezza = d.Next(100);
    s.larghezza = d.Next(100);
    s.profondità = d.Next(100);
}
```

6. A seguire dichiara il seguente metodo:

```
//---metodo ---
public string ScatolaInString(Scatola s)
{
    string tmp = Convert.ToString(s.altezza) + "\t";
    tmp += s.larghezza + "\t";
    tmp += s.profondità;
    return tmp;
}
```

7. A seguire dichiara il seguente metodo:

```
//---metodo ---
public Scatola StringInScatola(string s)
{
    char c = Convert.ToChar(13);
    string[] tmp = s.Split(c);
    Scatola res = new Scatola();
    res.altezza = Convert.ToInt16(tmp[0]);
    res.larghezza = Convert.ToInt16(tmp[1]);
    res.profondità = Convert.ToInt16(tmp[2]);
    return res;
}
```

PASSO 1 RIFLESSIONE

Controlla quanto hai scritto finora e prova a chiederti se ci sono parti che non hai compreso. In particolare:

- ❑ Hai capito come è fatta la struttura Scatola?
- ❑ Hai capito come è fatto il costruttore di una Scatola?
- ❑ Hai capito come è dichiarata la costante per il nome del file in cui salveremo i dati?
- ❑ Hai capito a cosa serve il metodo `GeneraScatola`? E hai capito che ritrova il separatore nella stringa per dividere i campi?
- ❑ Hai capito a cosa serve il metodo `ScatolaInString`? E hai capito che si usa un separatore tra i campi della struttura?
- ❑ Hai capito a cosa serve il metodo `StringInScatola`? E hai capito che ritrova il separatore nella stringa per dividere i campi?

PASSO 2 PULSANTE PER CREARE UNA LISTA DI SCATOLE

8. Adesso torna alla finestra e fai doppio clic sul pulsante **Genera Lista Scatole**; scrivi il seguente codice:

```
private void button1_Click(object sender, EventArgs e)
{
    int quanti = d.Next(100);
    //generazione di una lista di scatole
    magazzino.Clear();
    for (int i = 0; i < quanti; i++)
    {
        Scatola s = new Scatola();
        GeneraScatola(out s);
        magazzino.Add(s);
    }
}
```

PASSO 2 RIFLESSIONE

Hai appena costruito una lista di scatole.

Quando il pulsante è premuto (clic) il programma genera una quantità casuale di Scatole (meno di 100 comunque) e ciascuna scatola è generata casualmente dalla funzione GeneraScatola che pone nel parametro la scatola generata.

È molto importante che cerchi di comprendere il meccanismo attuato. Per il momento non c'è nessun file coinvolto: solo una lista di scatole in RAM, ma niente è stato scritto su disco.

Nel prossimo passaggio proviamo a salvare la lista (l'intera lista, nota, non una sola scatola!!!) in un file. Prova a fare un'ipotesi sul meccanismo.

PASSO 3 SALVATAGGIO SU FILE

9. Adesso torna alla finestra e fai doppio clic sul pulsante **Salva Lista Scatole**; scrivi il seguente codice:

```
private void button2_Click(object sender, EventArgs e)
{
    //salvataggio di un file di scatole (cancella file preesistente)
    StreamWriter sw = new StreamWriter(@"C:\miofile.txt", false);
    foreach (Scatola s in magazzino)
        sw.WriteLine(ScatolaInString(s));
    sw.Close();
}
```

PASSO 3 RIFLESSIONE

Il pulsante salva una lista di scatole su file. Osserva che il nome del file era stato dichiarato come costante.

Quando il pulsante è premuto (clic) il programma per prima cosa crea e apre un nuovo file. Poiché si usa uno **StreamWriter** significa che stiamo usando un file in scrittura. L'operazione di **new** non solo predispone la memoria per la variabile **sw** ma anche crea un file: se il file era preesistente lo distrugge e lo sostituisce; se il file non esisteva viene comunque creato.

Dopo aver creato ed aperto il file, il programma esplora l'intera lista con l'istruzione foreach; nel foreach viene usata una variabile s di tipo Scatola che serve per contenere (una alla volta) ciascuna delle scatole della lista. Quindi s assume di volta in volta il valore di una scatola.

Ogni scatola è scritta su file. Pensa al file come a una sequenza di pacchetti, ciascuno dei quali è una scatola. L'istruzione WriteLine operata su sw impone una scrittura su file. Però poiché non è possibile scrivere una scatola su file di testo è necessario prima convertire la scatola in stringa. Per fortuna abbiamo definito una funzione apposita.

È molto importante che cerchi di comprendere il meccanismo attuato. Il file è un file di testo, usato in scrittura.

Come ultima riflessione un suggerimento: cerca il file su Hard Disk e prova ad aprirlo.

Nel prossimo passaggio proviamo a recuperare la lista (l'intera lista, nota, non una sola scatola!!!) prelevandola da un file. Prova a fare un'ipotesi sul meccanismo.

PASSO 4 CARICAMENTO DA FILE

10. Torna alla finestra e doppio clic sul pulsante **Carica Lista Scatole**; scrivi il codice:

```
private void button3_Click(object sender, EventArgs e)
{
    //caricamento di una lista da un file di scatole (sola lettura)
    StreamReader sr = new StreamReader(nomeFile);
    magazzino.Clear();//svuoto la lista
    Scatola s;
    string tmp = sr.ReadLine();
    while (tmp != null) //se il file non è finito
    {
        s = StringInScatola(tmp);//preparo la scatola
        magazzino.Add(s);//la aggiungo alla lista
        tmp = sr.ReadLine();//leggo la prossima
    }
    sr.Close();//chiudo il file
}
```

PASSO 4 RIFLESSIONE

Il pulsante carica una lista di scatole da file. Osserva che il nome del file era stato dichiarato come costante.

Quando il pulsante è premuto (clic) il programma per prima cosa apre un file esistente in lettura. Poiché si usa uno **StreamReader** significa che stiamo usando un file in lettura. L'operazione di **new** non solo predispone la memoria per la variabile **sw** ma apre il file: se il file non era preesistente si solleva un errore.

Dopo aver aperto il file, il programma pulisce l'intera lista con l'istruzione Clear; poi dichiara una scatola e una variabile stringa di appoggio. La scatola servirà per memorizzare ciascuna scatola letta; la stringa serve per leggere da file le informazioni relative ad una scatola; la stringa dovrà essere convertita in scatola.

Il ciclo while controlla se il file è finito; se fosse finito la ReadLine() avrebbe letto un valore null. Se non ha letto un null significa che ha letto una scatola e deve eseguire le istruzioni del corpo del while; altrimenti il ciclo è terminato.

Ad ogni giro del ciclo la stringa è convertita in scatola e aggiunta alla lista. Poi si legge la successiva.

Suggerimento: puoi provare a modificare manualmente (con blocco note per esempio) i dati memorizzate nel file e dopo fare clic sul pulsante di caricamento (non occorre terminare l'applicazione; puoi lasciarla attiva mentre elabori il file manualmente). Vedrai che i dati cambiano e saranno poi visualizzati diversamente dalla generazione casuale.

PASSO 5 VISUALIZZAZIONE DI UNA LISTA

11. Adesso torna alla finestra e fai doppio clic sul pulsante **Visualizza Lista**; scrivi il seguente codice:

```
//visualizza lista nel richTextBox
richTextBox1.Clear(); //pulisco il testo
foreach (Scatola s in magazzino)
{
    string z = ScatolaInString(s);
    richTextBox1.AppendText(z);
    richTextBox1.AppendText("\n");//vai a capo
}
```

PASSO 5 RIFLESSIONE

Il pulsante mostra la lista di scatole in una casella di testo.

Ovviamente è possibile usare una ListBox per fare altrettanto.

Semplicemente si pulisce la casella, poi si esplora la lista e ciascuna scatola viene scritta nella casella. Non ti piace: prova tu a usare una ListBox.

ALTRI ESERCIZI

Esercizio 1) NAZIONI

- Prepara tre file di testo: soggetti.txt ; verbi.txt ; oggetti.txt ;
- Nel file soggetti elenca (andando a capo ad ogni riga) diversi soggetti (Abele, il gatto, il cane, ecc …);
- Nel file verbi elenca (andando a capo ad ogni riga) diversi verbi alla terza persona singolare (mangia, lava, pettina, sposta, ecc …);
- Nel file oggetti elenca (andando a capo ad ogni riga) diversi oggetti (Bruto, il topo, l'uccello, ecc …);
- Prepara una applicazione visuale. Poni nella finestra tre comboBox1, comboBox2, comboBox3. Adesso associa al metodo Form1_Load (doppio clic su Form1) il gestore che carica nei tre ComboBox le stringhe lette dai rispettivi file.
- Ciascuno dei tre ComboBox deve selezionare una delle frasi possibili tra quelle del rispettivo elenco.

Esercizio 2) CONFRONTI

- Prepara due file di testo: cani.txt ; gatti.txt ;
- Nel file dei cani elenca (andando a capo ad ogni riga) i nomi di almeno 10 cani (Rex, Lassie, RinTinTin, Pluto, ecc.)
- Nel file dei gatti elenca (andando a capo ad ogni riga) i nomi di almeno 10 gatti (Felix, Garfileld, Gambadilegno, Tom, ecc.)
- Poni un pulsante nel Form1 e associa un gestore di evento che legge da entrambi i file e verifica se esiste un nome presente in entrambi i file; se c'è lo mostra con un messaggio a video, altrimenti deve mostrare un messaggio TUTTI I NOMI SONO DIVERSI

Esercizio 3) FUSIONE

- Prepara due file di testo: cani.txt ; gatti.txt ;
- Nel file dei cani elenca (andando a capo ad ogni riga) i nomi di almeno 10 cani (Rex, Lassie, RinTinTin, Pluto, ecc.)
- Nel file dei gatti elenca (andando a capo ad ogni riga) i nomi di almeno 10 gatti (Felix, Garfileld, Gambadilegno, Tom, ecc.)
- Poni un pulsante nel Form1 e associa un gestore di evento che crea un nuovo file ottenuto dalla unione dei nomi di entrambi i due file sopra descritti

SOMMARIO