Sommario

13 4

Introduzione al "file system" 4

1 Concetti generali 4

1.1 Dispositivi fisici per la memorizzazione dei file 4
1.2 File system 4
1.3 Organizzazione logica dei *file system* 5
1.4 Percorsi assoluti e relativi di un file o una directory 7
1.5 Informazioni che caratterizzano file e directory 8
1.6 Natura, o tipo, dei file 8
1.7 Operazioni permesse sui file 11
1.8 Operazioni che agiscono sul contenuto del file 11
1.9 Operazioni che agiscono sul file come oggetto del *file system* 12

2 File di testo 13

2.1 Codifica dei dati in formato stringa 14
2.2 Codifica dei caratteri 15

3 File binari 17

3.1 Memorizzazione di stringhe nel formato binario 18
3.2 Lettura di un file binario 18

4 Concetto di stream 19

4.1 Lettura da uno stream 20
4.2 Scrittura su uno stream 20
4.3 Posizionamento 21

14 22

Classe "FileStream" 22

1 Introduzione all classe "FileStream" 22

1.1 Connessione con il file: creazione di un oggetto "FileStream" 22
1.2 Lettura e scrittura del file 26
1.3 Posizione corrente e lunghezza dello stream 26
1.4 Impiego della classe "FileStream" 27

15 30

Elaborare file di testo 30

1 Classe "StreamWriter" 30

1.1 Proprietà principali della classe "StreamWriter" 30
1.2 Costruttori della classe "StreamWriter" 31
1.3 Metodi della classe "StreamWriter" 31

2 Classe "StreamReader" 32

2.1 Proprietà principali della classe "StreamReader" 32
2.2 Costruttori della classe "StreamReader" 32
2.3 Metodi della classe "StreamReader" 33

3 Uso delle classi "StreamWriter" e "StreamReader" 33

3.1 Esempio n° 1: Impostazione a maiuscolo del "case" delle lettere 34
3.2 Esempio n° 2: Serializzazione di una classe su un file di testo 35

16 42

Accesso al *file system* 42

1 Introduzione 42
2 Classe "File" 43

2.1 Metodi della classe “File” 43
2.2 Attributi di un file o una directory: tipo “FileAttributes”. 45
3 Classe “Directory” 46
3.1 Metodi della classe “Directory” 46
3.2 Pattern e caratteri jolly 48
4 Classe “Path” 48
4.1 Campi pubblici classe “Path” 48
4.2 Metodi principali della classe “Path” 49
5 Classi FileInfo e DirectoryInfo 50
6 Uso delle classi “File”, “Directory” e “Path” 51
6.1 Requisiti del programma “Elimina” 51
6.2 Struttura generale del programma 52
6.3 Individuazione degli oggetti – file o directory – da eliminare 52
6.4 Eliminazione dei file e delle directory 54

13

Introduzione al “file system”

1 Concetti generali

Un file può essere definito in modo molto generale come un:

un contenitore d’informazioni a carattere persistente.

Il termine “persistente” si contrappone al termine “volatile”, che caratterizza le strutture dati che vengono create nella memoria centrale (RAM) e la cui esistenza, legata alla presenza in memoria dei programmi che le elaborano, non può comunque superare lo spegnimento del computer. I file, al contrario, risiedono su supporti fisici di natura diversa dalla memoria centrale. Essi, definiti “supporti permanenti”, o anche “memorie permanenti di massa”, garantiscono la persistenza dei dati memorizzati sui file anche quando il computer è spento.

1.1 Dispositivi fisici per la memorizzazione dei file

Da una certa prospettiva, la natura del supporto fisico nel quale sono memorizzati i file non è molto rilevante; l’aspetto fondamentale che lo caratterizza è infatti la permanenza delle informazioni. Le memorie di massa più comuni sono:

- dischi magnetici: *hard-disk*, *floppy-disk*, unità-zip;
- dischi ottici: CD-ROM R, CD-ROM RW, DVD, eccetera;
- dischi magneto-ottici;
- nastri (*tapes*);

Il modo in cui ognuno di questi dispositivi realizza la permanenza dei dati dipende dalla specifica tecnologia impiegata e dalla natura del materiale che caratterizza il supporto. La fase di memorizzazione vera e propria viene eseguita dal “controllo del dispositivo” (*device controller*), e cioè l’hardware che comanda la parte meccanica ed elettronica, guidato dal “driver del dispositivo” (*device driver*), e cioè il software specificatamente progettato per gestire il dispositivo.

Questo aspetto della memorizzazione dati non riguarda il programmatore da vicino[1], ma il sistema operativo. Quest’ultimo si interfaccia con i *device driver* che gestiscono i dispositivi di memorizzazione, nascondendo i dettagli della loro effettiva struttura e traducendola in un’altra, altamente organizzata e facilmente accessibile sia per il programmatore che per l’utente finale: il *file system*.

1.2 File system

Il termine *file system* ha un significato piuttosto vasto:

[1] A meno che non sia un programmatore di *device driver*.

- la parte – il software – del sistema operativo che gestisce la memorizzazione dei file in un determinato dispositivo;
- la modalità di organizzazione fisica dei file nel dispositivo;
- l'insieme dei file memorizzati, la loro organizzazione logica, le informazioni accessorie che caratterizzano ogni file e la natura delle operazioni che il sistema operativo consente su di essi.

Di queste tre prospettive, le prime due riguardano lo studio della struttura e del funzionamento dei sistemi operativi e quindi non sono rilevanti per il tipo di trattazione che stiamo facendo. La terza è importante sia per il programmatore che per l'utente finale.

1.3 Organizzazione logica dei *file system*

L'organizzazione logica di un *file system* riguarda la collocazione dei file nella memoria di massa così come la vedono il programmatore e l'utente finale. Il termine "logica" si contrappone a "fisica", poiché da questo punto di vista è del tutto irrilevante dove e come siano realmente memorizzati i file.

La maggior parte dei *file system*, perlomeno quelli utilizzati nei dischi, sono organizzati secondo una struttura ad albero chiamata "**albero delle directory**". Il *file system* è suddiviso in cartelle, o directory, le quali possono contenere file e altre cartelle, e così via, in una struttura arbitrariamente ramificata.

Esattamente come avviene per un albero, esiste una sola radice (o tronco) dalla quale dipartono dei rami, dai quali possono dipartire altri rami e così via. In questa metafora, i file sono le foglie, e soltanto essi memorizzano i dati. Le directory sono i rami che contengono le foglie e/o altri rami. La radice è rappresentata dal dispositivo fisico.

In questa struttura, ogni oggetto, file o directory, è individuato da un "percorso assoluto", che è composto nell'ordine:

- dall'elenco delle directory che occorre "attraversare" per raggiungere un oggetto, partendo dalla radice, eventualmente preceduto dal nome assegnato al dispositivo;
- da un nome che distingue l'oggetto dagli altri oggetti che si trovano nella directory.

Di seguito è schematizzata una tipica struttura ad albero:

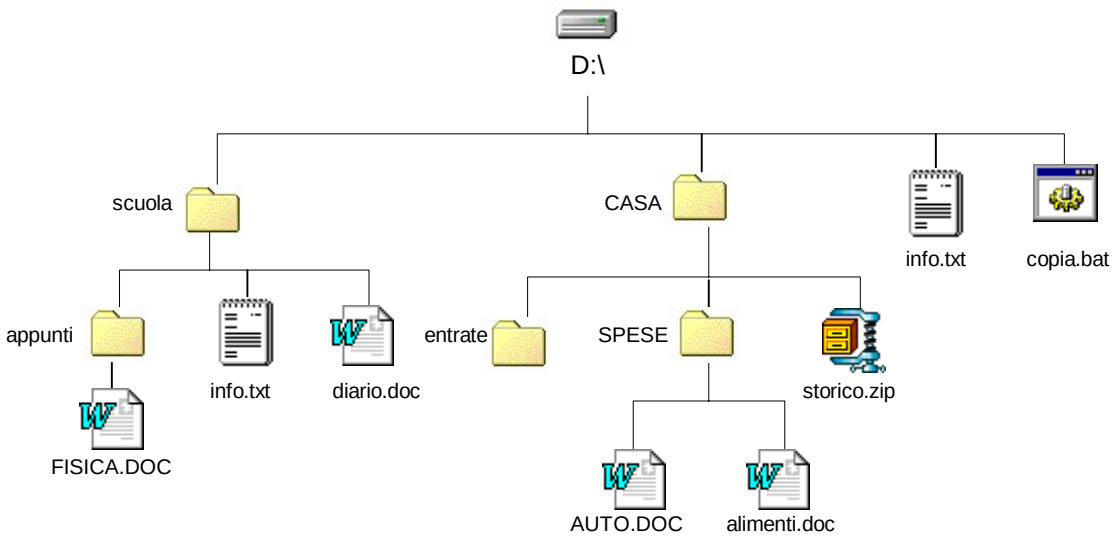

Figura 13-1 Rappresentazione schematica della struttura ad albero di un *file system*

In alcuni testi il nome di un file o una directory viene considerato facente parte del percorso, mentre in altri il percorso è rappresentato dal solo elenco delle directory da attraversare. In questo testo faremo riferimento alla prima definizione.

Ogni sistema operativo ha la propria simbologia e le proprie regole per indicare i dispositivi fisici, i percorsi e i nomi. Windows ha le seguenti:

- **dispositivi**: vengono definiti "unità" e sono indicati mediante una lettera seguita dal carattere "**:**" (due-punti); ad esempio: "**`A:`**", "**`B:`**", "**`C:`**" , "**`D:`**", eccetera. Le prime due lettere sono riservate ai *floppy disk*; le lettere successive agli *hard disk* e alle altre unità installate, nell'ordine in cui vengono riconosciute dal sistema operativo;

- **percorsi**: stringhe contenenti i nomi delle directory da attraversare e dell'oggetto, file o directory, da raggiungere, separati dal carattere "\" (barra retroversa). Opzionalmente, un percorso può contenere all'inizio il simbolo dell'unità.

- **nomi**: stringhe contenenti lettere, numeri, spazi, e altri simboli come "**–**", "**+**", "**_**", eccetera. Un nome può opzionalmente terminare con una particolare sequenza di caratteri, preceduta da un punto, chiamata "estensione". L'estensione, nel caso in cui il nome si riferisca ad un file, viene interpretata dal sistema operativo per identificare la natura del file in base ad un dizionario interno che associa ad ogni estensione un tipo di file.

A titolo di esempio analizziamo l'albero delle directory mostrato in Figura 12-1.

L'unità "`D:`" contiene 6 directory:

- "`scuola`" e "`casa`"; il loro percorso assoluto è "`D:\scuola`" e "`D:\casa`";

- "`appunti`"; il suo percorso assoluto è "`D:\scuola\appunti`";

- "`entrate`" e "`spese`"; il loro percorso è "`D:\casa\entrate`" e "`D:\casa\spese`";

Sono presenti 8 file:

- "`info.txt`" e "`copia.bat`"; il loro percorso assoluto è "`D:\info.txt`" e "`D:\copia.bat`";

- "`info.txt`" e "`diario.doc`"; il loro percorso assoluto è "`D:\scuola\info.txt`" e "`D:\scuola\info.txt`";

- "`fisica.doc`"; il suo percorso assoluto è "`D:\scuola\appunti\fisica.doc`";

- "`storico.zip`"; il suo percorso assoluto è "`D:\casa\storico.zip`";

- "`auto.doc`" e "`alimenti.doc`"; il loro percorso assoluto è: "`D:\casa\spese\auto.doc`" e "`D:\casa\spese\alimenti.doc`".

Inoltre:

- nell'albero delle directory si possono individuare tre "livelli".Due oggetti si trovano allo stesso livello quando, partendo dalla radice, è necessario attraversare lo stesso numero di directory per raggiungerli;

- esistono due file con lo stesso nome, "`info.txt`", ma ciò non viola le regole del *file system*, poiché due o più file o directory possono avere lo stesso nome purché abbiano un percorso assoluto diverso;

- ogni file è contraddistinto da un'estensione. Ciò non rappresenta un requisito, né implica necessariamente che l'estensione, "`doc`", "`txt`", "`zip`", eccetera, sia coerente con l'effettiva

natura del file. Semplicemente, il sistema operativo, attraverso l'estensione, si affida al proprio dizionario interno per tentare di associare il file ad una specifica applicazione e rappresentarlo con una determinata icona;

- sia nei nomi dei file che delle directory è irrilevante il "case" dei caratteri; ad esempio; "`casa`" e "`CASA`" sono lo stesso nome.

1.4 Percorsi assoluti e relativi di un file o una directory

In un percorso assoluto il punto di partenza è sempre rappresentato dalla radice e cioè dalla barra, preceduta opzionalmente dal simbolo di unità. Dunque, esso si presenta sempre nella forma:

`<UNITÀ>:\...`

oppure

"`\...`"

se viene omesso il simbolo di unità.

Un percorso relativo comincia invece dalla "directory corrente". Qualsiasi operazione che coinvolga gli oggetti del *file system* – esecuzione di un programma, modifica, cancellazione, creazione di un file di una directory – si svolge sempre nel contesto di una determinata directory, chiamata appunto directory corrente. In questo contesto, qualsiasi percorso che non sia assoluto (che non cominci con la barra) viene interpretato dal sistema operativo come relativo, e cioè "che parte" dalla directory in questione.

Ad esempio, il percorso assoluto del file "`alimenti.doc`" è "`D:\casa\spese\alimenti.doc`", mentre il suo percorso relativo alla directory "`casa`" è "`spese\alimenti.doc`".

Un aspetto che contraddistingue i percorsi relativi da quelli assoluti è che i primi possono "attraversare" l'albero delle directory anche all'indietro. Per rendere questo possibile sono previsti due simboli speciali "`.`" e "`..`", per indicare, all'interno di un percorso, la directory corrente e la directory precedente a quella corrente. Ciò detto, partendo ad esempio dalla directory "`spese`" (e cioè presupponendo "`spese`" come directory corrente), è possibile raggiungere i file "`fisica.doc`", "`diario.doc`" e "`copia.bat`" con i seguenti percorsi relativi:

- "`..\..\scuola\appunti\fisica.doc`";
- "`..\..\scuola\diario.doc`";
- "`..\..\copia.bat`";

Un percorso relativo che contiene il solo nome di un oggetto, file o directory, sottintende la sua esistenza nella directory corrente. In questo senso, data come directory corrente "`scuola`", i seguenti percorsi individuano gli stessi oggetti:

- "`appunti`" equivale a "`.\appunti`";
- "`info.txt`" equivale a "`.\info.txt`";
- "`diario.doc`" equivale a "`.\diario.doc`".

Percorsi UNC (Universal Naming Convention)

Nell'ambito di una rete locale esiste una convenzione che consente di fare riferimento ad oggetti che risiedono in un computer remoto; si chiama Universal Naming Convention e viene designata dalla sigla UNC.

Un percorso UNC è per definizione un percorso assoluto e inizia sempre con due barre. Dopo di queste, il primo elemento è il nome del computer nel quale si trova l'oggetto da raggiungere. Segue l'elenco delle directory condivise (in generale dei nomi condivisi, considerato che possono essere anche nomi di dispositivi) separate dalla barra come in un percorso qualsiasi.

Ad esempio, supponiamo che l'unità "`D:`" dell'esempio precedente si trovi in un server di nome "`trixie`" e che essa sia condivisa con il nome `D$`; ecco il percorso per accedere al file "`Info.txt`" della cartella scuola:

```
"\\trixie\d$\scuola\info.txt"
```

1.5 Informazioni che caratterizzano file e directory

Ogni oggetto memorizzato nel *file system* è caratterizzato da alcune informazioni la cui natura dipende fondamentalmente dal *file system* stesso, ma non dal contenuto dell'oggetto. Informazioni di base, praticamente comuni a tutti i *file system*, sono:

- dimensione (espressa in numero di byte o multipli di byte);
- data di creazione;
- data ultima modifica e/o data ultimo accesso;
- attributi: "nascosto", "sola lettura".

I *file system* dei dischi rigidi, perlomeno quelli più evoluti (il *file system* NTFS di Windows NT, 2000 e XP, ad esempio), memorizzano ulteriori dati, come i permessi di accesso all'oggetto in questione ("chi" può fare "cosa" con l'oggetto) e alcune informazioni di riepilogo.

Queste ed altre informazioni accessorie eventualmente disponibili prescindono dalla natura dell'oggetto e sono accessibili, e modificabili, (fatta salva l'appropriata autorizzazione) sia da parte del programmatore che dell'utente finale.

1.6 Natura, o tipo, dei file

Interrogarsi sulla natura dei file implica chiedersi se esistano delle caratteristiche – sulla struttura interna, nella modalità di memorizzazione, sulle informazioni accessorie, eccetera – che distinguano un file da un altro; o, da un'altra prospettiva, che accomunino file dello stesso tipo. In questo senso il concetto di "tipo" di file è piuttosto vago, poiché esistono vari criteri che possono essere usati per distinguere i file tra di loro.

Modalità di memorizzazione sul supporto

Perlomeno nei *file system* più comuni, tutti i file sono memorizzati nel supporto fisico esattamente con la stessa modalità, e dunque non c'è niente che caratterizzi un file rispetto a un altro. Nei dischi, ogni file viene suddiviso in blocchi di dimensione fissa (che dipende dal *file system*; in NTFS di Windows NT, 2000 e XP, ogni blocco è di 4096 byte). Poiché nel supporto i blocchi non occupano necessariamente posizioni contigue è compito del *file system* tenere traccia di quanti e quali blocchi sono riservati a un determinato file.

> Questa forma di memorizzazione produce un effetto "collaterale" importante: la dimensione reale di un file (e cioè la sua occupazione effettiva sul supporto fisico) è come minimo equivalente a quella di un blocco. Ad esempio, in NTFS un file di un solo byte occupa comunque 4096 byte su disco.

Modalità di memorizzazione logica

La suddivisione in blocchi di un file viene gestita dal sistema operativo in modo trasparente per il programmatore e l'utente finale. Il *file system* rende del file un'immagine diversa, e cioè quella di una sequenza di byte senza soluzione di continuità, nella quale ogni byte è individuato dalla propria posizione all'interno della sequenza:

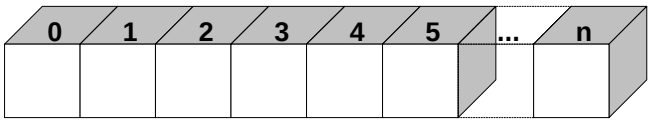

Figura 13-2 Rappresentazione logica della sequenza di byte che costituiscono un file.

Anche secondo questa prospettiva, non c'è niente che caratterizzi un file rispetto agli altri.

Estensione

L'estensione di un file è qualcosa che riguarda soltanto il suo nome ed è indipendente dal suo effettivo contenuto. All'interno dei *file system* più comuni l'estensione rappresenta soltanto un'utilissima convenzione che consente al sistema operativo di associare un file ad una determinata applicazione in grado di elaborarlo. Ad esempio, nei sistemi operativi Windows l'esecuzione di un doppio clic su un file determina le seguenti operazioni:

1) Windows verifica l'estensione del file: se questa è "EXE" (programma eseguibile), "BAT" (file di comandi) o "WSH" (file di script) tenta di eseguirlo,

2) altrimenti ne effettua la ricerca in un dizionario interno, memorizzato nel registro di sistema, che tiene traccia delle estensioni registrate. Alcune applicazioni registrano in questo dizionario una o più estensioni di file, i quali saranno dunque riconosciuti da Windows come prodotti e dunque processabili dalle applicazioni in questione. Ad esempio, se nel computer è istallato Microsoft Office, l'applicazione registrata per i file di estensione "XLS" è Microsoft Excel;

3) se l'estensione non viene trovata, Windows chiede all'utente di indicare una particolare applicazione, tre quelle installate, da utilizzare per processare il file.

In tutto questo, il contenuto, la dimensione o altre informazioni che caratterizzano il file non vengono prese in considerazione. Ad esempio, nulla impedisce di creare un file di testo, ad esempio con il programma Blocco note, e di nominarlo con l'estensione "EXE". Windows riconoscerà il file come un programma eseguibile, eccetto poi visualizzare un messaggio di errore se si tenta di eseguirlo. Per contro, nominare un file creato con Word con l'estensione "EXE", non lo rende affatto incompatibile per una successiva rielaborazione con il programma di video scrittura.

> Gli esempi presentati hanno il solo scopo di chiarire il concetto. Non esiste alcun motivo sensato per modificare l'estensione di un file rispetto a quella impostata dall'applicazione con la quale viene processato.

Formato

Con il termine "formato" ci si riferisce a una particolare strutturazione dei dati memorizzati nel file. Il formato di un file è strettamente connesso con l'applicazione, o le applicazioni, in grado di processarlo. Ad esempio, un file che contiene un'immagine prodotta con il programma MS Paint (formato Bitmap) è strutturato molto approssimativamente nel seguente modo:

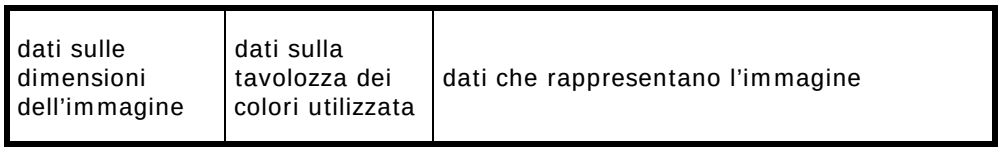

Figura 13-3 Schema (semplificato) del formato di una immagine Bitmap

Il formato, esattamente come l'estensione, è una convenzione, che però in questo caso è relativa al contenuto del file e non al suo nome. Ritornando al precedente esempio, non c'è niente nei byte memorizzati in un file in formato Bitmap che li caratterizzi come riservati alla parte che definisce le dimensioni dell'immagine, piuttosto che a quella che definisce l'immagine vera e propria. Semplicemente, qualsiasi applicazione in grado di leggere un file in questo formato "sa" che, ad esempio, i primi 16 byte definiscono le dimensioni, i successivi 256 definiscono la tavolozza dei colori e tutti i byte restanti definiscono l'immagine[2].

Per vari motivi alcuni formati, largamente diffusi, si affermano come degli standard. Un "formato standard" non è legato a nessuna specifica applicazione e per questo motivo può essere riconosciuto da un numero di applicazioni molto vasto. Esempi di formati standard per la memorizzazione delle immagini: PNG, JPEG, GIF, eccetera. Nella memorizzazione dei testi un formato standard molto diffuso è RTF (Rich Text Format).

Esistono poi i "formati proprietari", i quali sono strettamente legati a un'applicazione e dunque alla casa produttrice della stessa. Un formato proprietario può essere ancora più diffuso e utilizzato di un formato standard, ma la sua diffusione è collegata alla diffusione dell'applicazione che lo ha introdotto. Un esempio di formato proprietario diffusissimo (nelle sue varie versioni) è quello utilizzato dai documenti prodotti con Microsoft Word.

File eseguibili e file di dati

Con il termine "file eseguibili" s'intende qui classificare tutti quei file che in una forma o in un'altra contengono delle istruzioni che possono essere eseguite, direttamente o indirettamente, dal computer. Questi file sono contraddistinti da specifiche estensioni – "EXE", "DLL", "BAT", "BIN", "WSH", "COM", eccetera –, ma non è questo che li rende di per sé eseguibili, ma il loro effettivo contenuto e il formato che questo assume, formato che dev'essere riconosciuto dal sistema operativo perché il file possa essere eseguito. Al di là di questo, ed eventuali informazioni accessorie specifiche, non c'è niente che distingua un file eseguibile da qualsiasi altro file.

Un "file di dati", come suggerisce il termine, ha lo scopo di memorizzare informazioni – testi, immagini, suoni, video, eccetera – e di norma è processato da un qualche tipo di applicazione. In un certo senso, se un file non è eseguibile significa che è un file di dati; resta il fatto che il termine "file di dati" è piuttosto vago: non c'è niente nel contenuto di un file di dati che lo qualifichi automaticamente come tale.

Ad esempio, esistono applicazioni che sono in grado di effettuare determinate elaborazioni su programmi – e quindi file – eseguibili. In questo senso, i programmi in questione assumono il ruolo di file di dati, poiché non vengono eseguiti, ma processati da un'applicazione.

File di testo e file binari

I concetti di file di testo e file binario, e dunque di "formato testo" e "formato binario", rappresentano degli argomenti centrali per la comprensione degli altri capitoli e quindi ad essi sono riservati dei paragrafi a parte.

[2] Nella realtà le cose sono un po' più complicate, ma concettualmente equivalenti.

1.7 Operazioni permesse sui file

In relazione ai file si possono individuare quattro classi di operazioni:

- operazioni che agiscono sul contenuto: creazione, scrittura e lettura;
- operazioni che agiscono sul file inteso come oggetto del *file system*: copia, spostamento, cancellazione del file, modifica del nome;
- operazioni che agiscono sulle informazioni accessorie del file;
- operazioni che agiscono sui permessi associati al file (ammissibili solo su alcuni *file system*, come ad esempio NTFS di Windows NT, 2000, XP).

Di seguito, ma soprattutto nei capitoli successivi, sono trattate soltanto le prime due classi di operazioni.

1.8 Operazioni che agiscono sul contenuto del file

Rappresenta la classe di operazioni più importanti per il programmatore, poiché la maggior parte dei programmi esegue una qualche forma di elaborazione sui dati contenuti in uno o più file. In generale, l'elaborazione del contenuto di un file prevede sempre le seguenti 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.

"Apertura" di un file esistente

Nell'operazione di apertura viene a crearsi una connessione tra il file e il programma che lo elabora. In un certo senso, il programma si "appropria" temporaneamente del file, per poterne elaborare il contenuto. Tale connessione può essere:

- di tipo esclusivo, nel qual caso, fintantoché il file non viene chiuso nessun altro programma può accedere ad esso;
- caratterizzata da una condivisione più o meno restrittiva, nel qual caso altri programmi possono avere il permesso di leggere, scrivere o leggere/scrivere sul file.

Nell'operazione di apertura è inoltre possibile specificare il tipo di elaborazione – lettura, scrittura o lettura/scrittura – permessa sul suo contenuto.

"Creazione" di un nuovo file

Laddove l'operazione di apertura presuppone l'elaborazione di un file esistente, questa operazione implica la creazione di un nuovo oggetto – inizialmente vuoto – nel *file system* e la conseguente connessione tra questo e il programma. E' anche possibile creare un nuovo file al posto di uno già esistente, operazione che si traduce nella cancellazione di ogni informazione contenuta nel file originale.

“Lettura” di un file

Questa operazione implica che il file sia stato precedentemente aperto o creato e si traduce nel caricamento in memoria delle informazioni in esso contenute. La sola lettura non determina alcuna modifica del file.

“Scrittura” in un file

Questa operazione implica che il file sia stato precedentemente aperto o creato. La scrittura su file assume il ruolo di:

- modifica, se i dati vengono sovrascritti a quelli già esistenti;
- aggiunta, se i dati vengono accodati a quelli già esistenti.

L’operazione di scrittura può essere “bufferizzata”; ciò significa che i dati non vengono immediatamente scritti nel supporto fisico, ma accumulati in un apposita area di memoria, chiamata “buffer”. Quando il buffer è pieno il suo contenuto viene effettivamente memorizzato sul file e il buffer viene svuotato. Ciò velocizza le operazioni di scrittura, riducendo il numero di operazioni effettive sul supporto fisico che, qualunque sia la sua natura, è comunque molto più lento della memoria centrale.

“Chiusura” del file

Questa operazione implica che il file sia stato precedentemente aperto o creato. L’operazione di chiusura “rilascia” la connessione precedentemente creata tra il programma e il file, il quale non può dunque più essere elaborato, a meno che non venga riaperto. Qualsiasi forma di elaborazione su file deve prevedere l’operazione finale di chiusura, per due motivi:

- se il file è bufferizzato ed è stato aperto in modalità scrittura o lettura/scrittura, essa produce lo svuotamento del buffer, memorizzando il suo contenuto sul file;
- se il file non venisse chiuso, la connessione tra di esso e il programma resterebbe attiva e dunque altri programmi che volessero elaborarlo potrebbero non riuscire a farlo;
- una connessione consuma risorse del sistema operativo e dunque non c’è alcun motivo per lasciarla aperta se non è più necessario.

1.9 Operazioni che agiscono sul file come oggetto del *file system*

Tali operazioni sono completamente indipendenti dal contenuto e dalla natura del file.

“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.

Oltre al nome, vi sono altre informazioni che possono risultare differenti tra il file originale e quello duplicato:

- la data di creazione e quella di modifica;
- gli attributi;
- le informazioni – se presenti – relative ai permessi associati al file.

Un file può essere copiato:

- nella stessa directory;
- in una directory diversa, ma nello stesso dispositivo fisico;
- in un altro dispositivo fisico, il quale può essere di natura diversa dal primo e gestito da un *file system* diverso. In questo caso, comunque, i due *file system* devono essere in qualche modo compatibili.

"Modifica del nome" 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.

2 File di testo

Innanzitutto, il termine "file di testo" è convenzionale e non significa che i file di questo tipo debbano contenere solo dati di natura testuale, oppure che siano prodotti da programmi di video scrittura. Esso fa riferimento al formato, "formato testo" appunto, mediante il quale sono memorizzate le informazioni nel file.

Il formato testo di un file implica che:

- il suo contenuto, qualunque sia la sua natura, è comunque espresso in forma testuale;
- esso è di norma (ma non necessariamente) suddiviso in righe, e cioè stringhe di caratteri di lunghezza arbitraria il cui termine è indicato dalla sequenza di caratteri "`\r\n`" (ritorno carrello e nuova linea) oppure dal solo carattere '`\n`'.

Ciò che caratterizza un file di testo è dunque il formato che adotta; non esiste niente d'intrinseco che qualifichi un file di testo come tale, sia nella modalità di memorizzazione fisica che logica, né nelle informazioni accessorie, e nulla impedisce di processare un file di questo tipo come se fosse di natura diversa.

Il formato testo è universalmente riconosciuto da qualsiasi tipo di software e sistema operativo, e infatti viene impiegato per memorizzare contenuti di svariata natura, come ad esempio:

- il codice sorgente di tutti i linguaggi di programmazione;
- pagine HTML e documenti XML;
- file di comandi, di estensione "BAT" e file di script, di estensione "WSH";
- semplici testi privi di formattazione, sia nei paragrafi che nel tipo di carattere, prodotti ad esempio dall'editor Blocco note;
- file di configurazione di programmi e/o sistemi operativi; file di log e di riepilogo prodotti da alcuni software, come ad esempio gli antivirus, o sintesi del risultato di alcune operazioni del sistema operativo.

Memorizzare i dati in forma testuale significa convertirli in stringhe e quindi scrivere queste ultime nel file. Ciò introduce due questioni, inerenti l'aspetto della codifica delle informazioni all'interno del file:

- la modalità di rappresentazione dei vari tipi di dati in forma di stringa: codifica dei dati;
- la modalità di codifica dei singoli caratteri della stringa: codifica dei caratteri.

2.1 Codifica dei dati in formato stringa

La questione riguarda la codifica delle informazioni che non sono di natura testuale, poiché le stringhe non richiedono ovviamente nessuna forma di conversione. Si consideri ad esempio di voler memorizzare in un file di testo tre numeri, uno per ogni riga del file: `10`, `1.5` e `-2.13`. Il file risultante può essere così schematizzato:

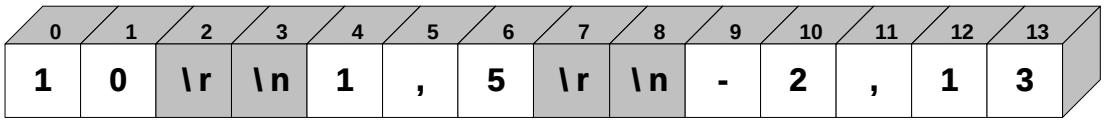

Figura 13-4 Rappresentazione di un file di testo contenente tre numeri[3]

Come si vede, il separatore decimale impiegato è la virgola, poiché così impone la convenzione usata nel nostro paese, mentre nei paesi anglosassoni viene usato il punto:

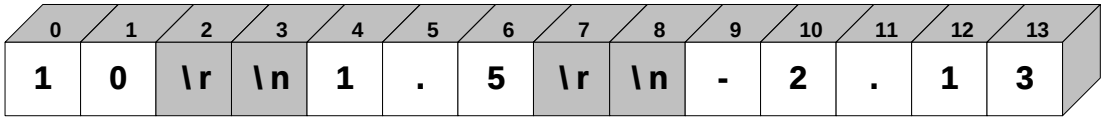

Figura 13-5 Rappresentazione di un file di testo contenente tre numeri; formato anglosassone

Ciò significa che lo stesso file (contenente gli stessi tre numeri) scritto in un paese anglosassone pone dei problemi d'interpretazione se elaborato in un computer sul quale gira un sistema operativo impostato per la lingua italiana. Discorso analogo vale per la rappresentazione del separatore delle

[3] Le sequenze di simboli "\r\n", evidenziati in grigio, rappresentano i terminatori di riga nel formato testo adottato nei sistemi operativi Windows.

migliaia, delle date, di valori che esprimono valute (e che dunque richiedono il simbolo di valuta appropriato), delle date, eccetera.

Esistono quindi dei dati la cui rappresentazione in formato stringa è sensibile alla "cultura" di un determinato paese. Quando viene processato un file di testo, la modalità d'interpretazione del suo contenuto deve rispettare la cultura applicata nella fase di scrittura, pena un'interpretazione errata. Ad esempio, il file schematizzato in `Figura 12-5`, se interpretato attraverso la cultura italiana, produce i numeri `10`, `15`, `-213`.

2.2 Codifica dei caratteri

Negli esempi precedenti viene dato per scontato che la fase di scrittura vera e propria di un certo valore si traduca nella memorizzazione nel file, carattere dopo carattere, della stringa che rappresenta il valore in questione. Ovviamente, nel file non vengono memorizzati caratteri ma numeri, più precisamente codici numerici che hanno una corrispondenza con un determinato insieme di caratteri.

Ciò non produrrebbe problemi se esistesse un solo insieme di caratteri utilizzabile; in questo caso infatti, la forma di codifica (quale numero corrisponde a quale carattere) impiegata durante la scrittura sarebbe per la stessa di quella impiegata durante la lettura. Nel mondo Windows, prima dell'avvento del sistema operativo Windows 2000, la situazione era appunto questa.

> Qui si sorvola sul fatto che in senso stretto non esiste, né è mai esistito un solo set di caratteri, poiché moltissime lingue contengono caratteri – simboli fonetici – che non hanno un equivalente in altre lingue, o comunque nella lingua inglese, che tradizionalmente è la lingua di riferimento nel mondo dell'informatica.

Nei sistemi operativi che vanno della serie `Windows 3.1` a `Windows Millenium` (passando per `Windows 95` e `98`), l'insieme di caratteri prevalentemente impiegato è quello ASCII, nel quale ad ogni carattere corrisponde un valore numerico che varia nell'intervallo 0 – 127, rappresentabile mediante un solo byte. Ad esempio, un file contenente le parole "`QUI`", "`QUO`", "`QUA`", una per ogni riga, viene codificato come segue:

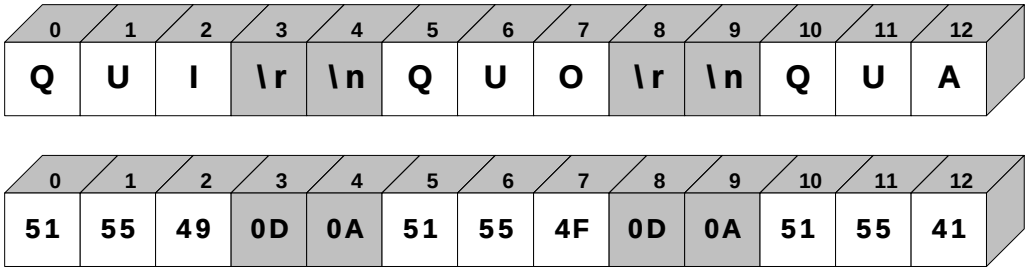

Figura 13-6 Rappresentazione del contenuto effettivo di un file ASCII[4]

Un file così codificato viene anche chiamato file ASCII.

Contestualmente all'avvento di Windows 2000 è nato un insieme di caratteri molto più vasto, chiamato Unicode, il quale comprende i sotto insiemi:

- ASCII;
- UTF-8;
- UTF-7.

[4] La rappresentazione numerica è espressa in formato esadecimale

Il contenuto di un testo può essere dunque codificato mediante uno di quattro insiemi di caratteri e, come si intuisce, perché la sua interpretazione in fase di lettura abbia successo è necessario che avvenga mediante lo stessa codifica usata durante la scrittura.

Non è solo una questione di codici ma anche di quanti byte sono impiegati per memorizzare un carattere. L'Unicode rappresenta ad esempio un formato di codifica a 16 bit, il quale impiega due byte per memorizzare un carattere.

> Per la precisione, l'insieme Unicode esiste in due forme: "little-endian" e "big-endian", le quali si distinguono per l'ordine con il quale vengono memorizzati i due byte che codificano un carattere. Nel testo faremo implicitamente riferimento alla forma "little-endian".

Il contenuto del file schematizzato in precedenza espresso mediante il codice Unicode viene così rappresentato:

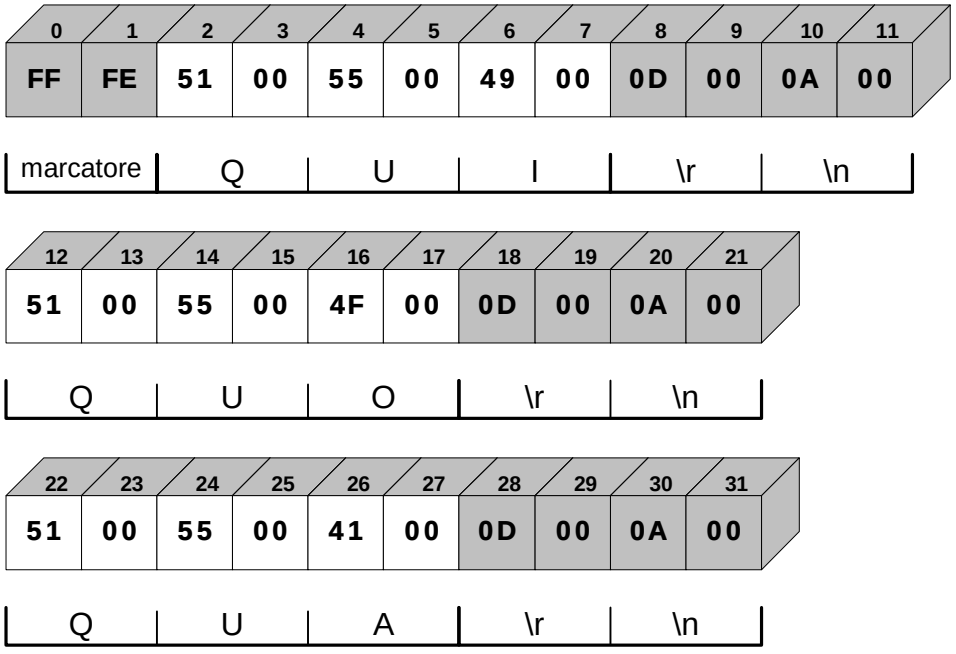

Figura 13-7 Rappresentazione del contenuto effettivo di un file Unicode[5]

Ogni carattere, compresi i due che fungono da terminare di linea, è rappresentato da una coppia di byte, dei quali il primo (quello che in memoria ha l'ordine più basso) coincide con l'equivalente in ASCII. Inoltre, i primi due byte fungono da "marcatore", e cioè da sequenza che qualifica il tipo di codifica Unicode.

Nella codifica UTF-8, ogni carattere occupa un byte. D'altra parte, all'inizio del file viene scritto un prologo di tre byte che identifica appunto questo tipo di codifica. Di seguito è mostrato lo stesso contenuto codificato in UTF-8.

[5] La rappresentazione numerica è espressa in formato esadecimale

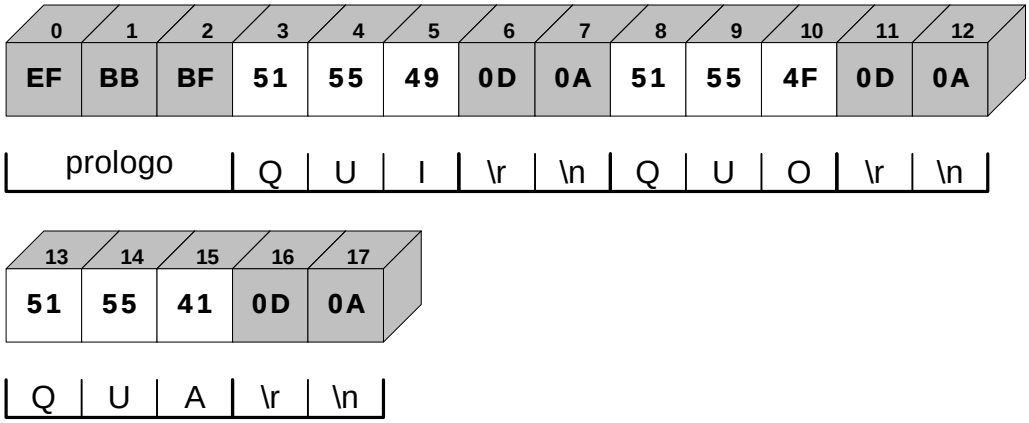

Figura 13-8 Rappresentazione del contenuto effettivo di un file UTF-8[6]

3 File binari

Con il termine "file binario" o "formato binario" ci si riferisce ai file che non adottano il "formato testo" per la memorizzazione dei dati, i quali non vengono dunque convertiti in forma testuale prima di essere memorizzati. In estrema sintesi: è formato binario tutto ciò che non è formato testo. In base a questa definizione, rientrano ad esempio nella categoria dei file binari:

- i file contenenti codice eseguibile (estensioni EXE, DLL, BIN, COM);
- i file contenenti immagini (estensioni: BMP, GIF, JPG, PNG, eccetera);
- i file contenenti video e audio (estensioni AVI, MPG, MP3, WAV, AIF).

Nel formato binario i dati vengono scritti nel file così come sono rappresentati in memoria, effettuando una copia diretta, partendo dal byte di indirizzo minore fino a quello di indirizzo maggiore. Ad esempio, la scrittura in formato binario dei valori interi `10`, `2`, `-2`, produce:

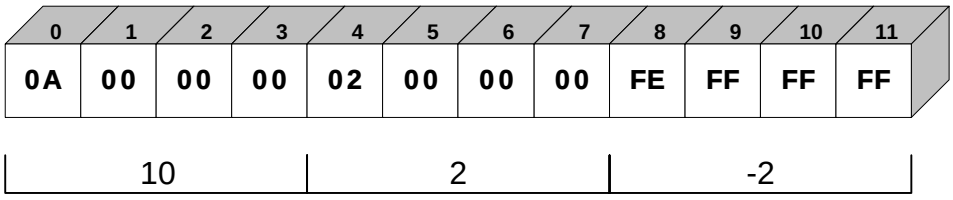

Figura 13-9 Rappresentazione di un file in formato binario contenente tre numeri interi

I valori interi negativi vengono rappresentati in memoria nella notazione in complemento a due, che equivale a FFFFFFFF – "valore assoluto del numero" + 1. Il valore –2 diventa infatti: FFFFFFFF – 00000002 + 1 = FFFFFFFE, che viene scritto come FE FF FF FF, poiché la scrittura parte dal byte di ordine inferiore, che cioè si trova all'indirizzo di memoria minore.

Nel file, ogni valore di un certo tipo occupa la stessa quantità di byte ed è scritto mediante la stessa configurazione impiegata in memoria. La figura seguente mostra la configurazione di un file binario contenente i valori `double` `1500.0` e `-0.5`.

[6] Idem

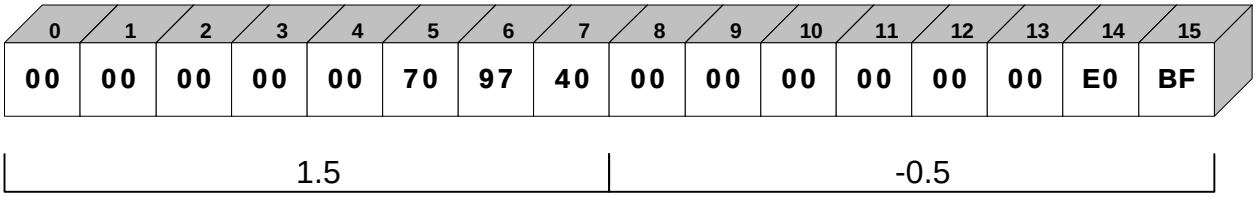

Figura 13-10 Rappresentazione di un file in formato binario contenente due valori `double`

Ogni numero, al di là del suo valore effettivo, produce la scrittura su file di 8 byte, che è l'occupazione in memoria di un valore `double`.

3.1 Memorizzazione di stringhe nel formato binario

Nei file binari le stringhe sono rappresentate analogamente a quanto avviene nei file di testo, poiché in questo caso non si pone alcun problema di conversione in quanto la scrittura di una stringa si traduce, in entrambi i formati, nella memorizzazione dei singoli caratteri che la compongono, partendo dal primo. Esiste comunque una differenza:

nel formato binario una stringa è preceduta dalla propria lunghezza.

Ad esempio, un file contenente le parole "`QUI`", "`QUO`", "`QUA`" viene codificato come segue:

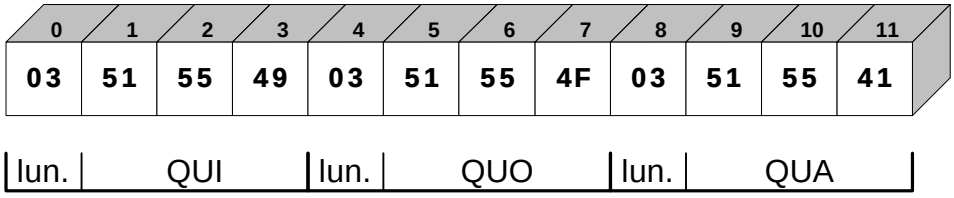

Figura 13-11 Rappresentazione di un file binario contenente tre stringhe

Anche in questo caso l'insieme di caratteri usato – Unicode, UTF-8, UTF-7, ASCII – determina l'effettiva modalità di codifica dei caratteri nel file. Ciò si riflette anche nel valore che indica la lunghezza della stringa, esso riflette infatti il numero di byte effettivamente necessari per memorizzarla.

Ad esempio, di seguito è mostrato il contenuto di un file che memorizza la parola "`CIAO`" codificata in Unicode.

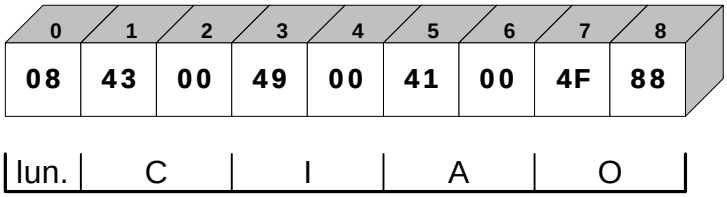

Figura 13-12 Rappresentazione di un file binario contenente la stringa "`CIAO`"

Come si vede, la lunghezza è 8, e cioè il numero di byte impiegati per rappresentare la stringa e non il numero di caratteri che la compongono.

E' importante sottolineare che diversamente dai file di testo, la codifica dei caratteri viene applicata soltanto ai tipi di dati di natura testuale e cioè ai valori `string` e `char`.

3.2 Lettura di un file binario

Nel formato binario i dati occupano posizioni contigue all'interno della sequenza (logica) dei byte contenuti nel file. E, contrariamente a quanto avviene per i terminatori di riga dei file di testo, non

esiste alcuna convenzione o particolare sequenza di byte che stabilisca un confine tra i dati, i quali possono essere interpretati in molti modi diversi.

Si consideri ad esempio l'ipotesi di un file binario contenente i seguenti byte:

```
08   41   41   41   04   4F   4B   45   59
```

Non esiste assolutamente niente, nei byte contenuti nel file, che possa suggerire il dato o i dati che vi sono stati memorizzati. Potrebbe essere ad esempio un singolo dato, rappresentato dalla stringa "AAA♦OKEY":

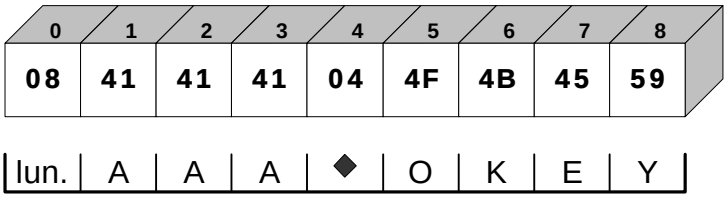

Figura 13-13 Interpretazione del contenuto del file come stringa lunga otto caratteri

Ma anche il valore intero 1094795528 seguito dalla stringa "OKEY":

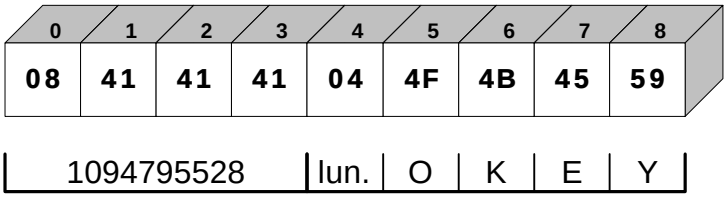

Figura 13-14 Interpretazione del contenuto del file come valore intero seguito da stringa

Come si comprende, l'effettiva disposizione dei dati nel file sta tutta nella logica del codice che lo scrive, e di conseguenza anche nella logica del codice che lo interpreta. Nel caso precedente, se prima viene scritto un valore intero e successivamente una stringa, in fase di lettura dovrà prima essere letto un valore intero e quindi una stringa, in caso contrario i dati sarebbero interpretati in modo incoerente rispetto al modo in cui sono stati scritti, con esiti imprevedibili.

4 Concetto di stream

Perché un programma possa processare un file è necessario che prima stabilisca un collegamento con esso. Ciò 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. Tale oggetto è uno "stream". In realtà il concetto di stream e indipendente da quello di file; uno stream è infatti un:

oggetto astratto che fornisce l'accesso a una generica sequenza di byte indipendentemente dalla sua reale natura e provenienza.

Tale sequenza di byte, definita "sorgente dati", può essere appunto il contenuto di file, ma anche un'area di memoria, oppure il flusso di dati proveniente da un dispositivo di input (la tastiera o la scheda seriale ad esempio) o da una scheda di rete.

Schematicamente, uno stream può essere così rappresentato:

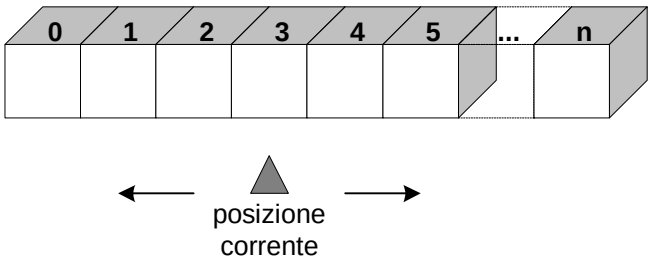

Figura 13-15 Rappresentazione schematica di uno stream

Lo stream mantiene internamente un indicatore “della posizione corrente” che inizialmente punterà al primo byte (nello schema il byte zero) della sorgente di dati.

Fondamentalmente, uno stream può supportare una o più delle seguenti operazioni:

- lettura: i dati vengono letti dallo stream e caricati in memoria;
- scrittura: i dati vengono memorizzati nello stream;
- posizionamento: viene modificata la “posizione corrente” senza che siano letti o scritti dei dati.

Le operazioni effettivamente supportate dipendono dall’effettiva natura della sorgente di dati. Ad esempio, uno stream associato alla tastiera non può che operare in sola lettura. Al contrario, uno stream associato allo schermo non può che operare in sola scrittura.

4.1 Lettura da uno stream

Leggere da uno stream significa caricare in memoria un certo numero di byte provenienti dalla sorgente dati. La lettura inizia dal byte che si trova nella “posizione corrente”, la quale viene aggiornata – aumentata – in base al numero di byte letti. Se la sorgente dati sottostante allo stream supporta la sola lettura, la “posizione corrente” può essere solo aumentata, e solo come conseguenza di un’operazione di lettura.

In figura è mostrato un esempio di lettura di 5 byte da un ipotetico stream:

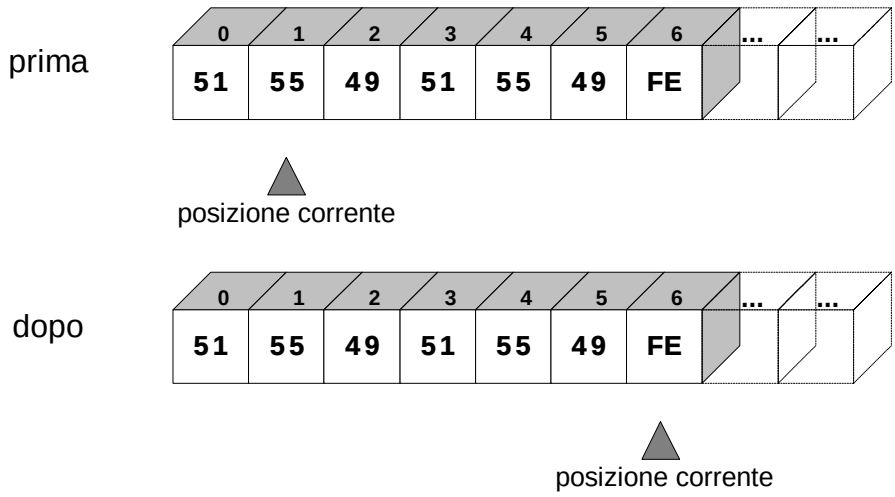

Figura 13-16 Esempio di lettura di 5 byte da uno stream

4.2 Scrittura su uno stream

Scrivere su uno stream significa memorizzare nella sorgente dati un certo numero di byte. La scrittura inizia a partire dalla “posizione corrente”, la quale viene aggiornata – aumentata – in base

al numero di byte scritti. Se la sorgente dati sottostante allo stream supporta la sola scrittura, la "posizione corrente" può essere solo aumentata, e solo come conseguenza di un'operazione di scrittura.

Di seguito viene schematizzata la scrittura di 4 byte – `04`, `4F`, `4B`, `45`, `59` –su un ipotetico stream inizialmente vuoto.

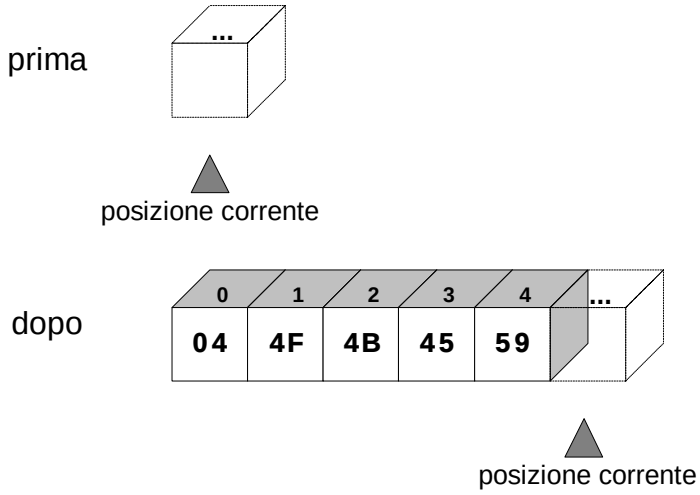

Figura 13-17 Esempio di scrittura di 5 byte da uno stream

4.3 Posizionamento

L'operazione di posizionamento (*seeking*) si traduce nel modificare – aumentare o diminuire – l'indicatore di "posizione corrente" senza leggere o scrivere byte. Soltanto alcune sorgenti dati supportano il posizionamento, tra le quali i file. Tale operazione non viene invece supportata da sorgenti dati come tastiera, schermo, seriale, eccetera.

> In linea di massima supportano il posizionamento le sorgenti di dati persistenti, come appunto i file. (Anche se ciò dipende in realtà dalla natura del *file system*).

14

Classe “FileStream”

Il .NET Framework fornisce un modello ad oggetti appositamente realizzato per accedere ad un flusso di dati rappresentato da uno “stream”. In tale modello esiste una netta separazione della logica di accesso di basso livello da quella che dipende dal formato dei dati, sia esso semplice testo, binario, XML, eccetera.

La prima è implementata mediante oggetti di tipo `Stream`, i quali supportano l’accesso al flusso di dati come se fosse una sequenza di byte. La logica dipendente dal formato è invece implementata mediante oggetti specializzati nell’accesso a flussi in formato testo, binario, XML, eccetera. Questi ultimi si appoggiano agli oggetti del primo tipo per accedere ai byte dello stream, in molti casi ignorando l’effettiva origine dei dati (file, tastiera, rete, eccetera). In questo senso essi sono convenzionalmente chiamati “*reader*” e “*writer*”, poiché il svolgono il compito di leggere e/o scrivere su uno stream, indipendentemente dalla sua natura.

In questo capitolo analizzeremo le caratteristiche e il funzionamento della classe `FileStream`, i cui oggetti forniscono l’accesso di basso livello ad un file. Nei due capitoli successivi esamineremo quattro classi specializzate nella lettura/scrittura di dati in formato binario e in formato testo.

Tutte le classi esaminate, in questo e nei successivi capitoli, sono definite all’interno del *namespace* `System.IO`.

1 Introduzione all classe “FileStream”

La classe `FileStream` deriva dalla classe astratta `Stream` ed è specializzata per fornire l’accesso al contenuto di un file. Essa definisce le operazioni di: creazione, apertura, lettura, scrittura, chiusura, posizionamento su un determinato byte della sequenza logica rappresentata dal file.

1.1 Connessione con il file: creazione di un oggetto “FileStream”

La connessione con il file viene stabilita contestualmente alla creazione dell’oggetto `FileStream` mediante l’invocazione del costruttore. Passando gli appropriati argomenti al costruttore è possibile specificare alcune informazioni sulla modalità di accesso richiesta:

- percorso del file;
- modo di connessione;
- tipo di accesso;
- livello di condivisione.

L’invocazione del costruttore assume la seguente forma:

```
oggetto-stream = new FileStream(string percorso,
                                FileMode modo,
                                FileAccess accesso_opz,
                                FileShare condivisione_opz);
```

Ad esempio:

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

Da notare che il terzo e il quarto argomento sono opzionali; comunque, per specificare l'argomento `FileShare` è necessario specificare anche l'argomento `FileAccess`.

> In realtà, il percorso non è l'unico modo per specificare il file da processare. E' però il più comune e in questo testo sarà fatto riferimento unicamente ad esso.

Percorso del file

Questo argomento è rappresentato da una stringa contenente il percorso – assoluto o relativo – del file con il quale stabilire la connessione. Tale argomento e quello che specifica la modalità di connessione sono gli unici obbligatori.

Modo di connessione

Consente di specificare se il file dev'essere creato o aperto, e in quest'ultimo caso se le operazioni di scrittura devono essere interpretate come l'aggiunta di nuovi dati o la sovrascrittura di quelli esistenti. Insieme al percorso è l'unico argomento obbligatorio.

Tabella 14-1 Elenco delle modalità di connessione ammesse

MODO CONNESSIONE FILEMODE	DESCRIZIONE
Append	Se il file esiste lo apre e si posiziona alla fine, impostando la proprietà Position ("posizione corrente" dello stream). Se il file non esiste lo crea. Questa modalità di connessione consente la sola scrittura e quindi dev'essere usata in congiunzione con FileAccess.Write, in caso contrario viene sollevata l'eccezione ArgumentException.
Create	Crea il file. Se il percorso specificato fa riferimento a un file già esistente il suo contenuto viene cancellato.
CreateNew	Crea il file. Se il percorso specificato fa riferimento a un file già esistente viene sollevata l'eccezione IOException.
Open	Apre il file. Se il file non esiste viene sollevata l'eccezione FileNotFoundException.
OpenOrCreate	Apre il file se questo esiste, altrimenti lo crea.
Truncate	Apre il file, ne azzera il contenuto e imposta Position a 0. Se il file non esiste viene sollevata l'eccezione FileNotFoundException.

Tipo di accesso

Mediante questo argomento si specifica se l'accesso al file è in lettura, scrittura o lettura/scrittura:

Tabella 14-2 Elenco dei tipi di accesso ammessi

TIPO ACCESSO FILEACCESS	DESCRIZIONE
Read	E' ammessa la sola operazione di lettura. Un eventuale tentativo di scrittura produce l'eccezione NotSupportedException.
Write	E' ammessa la sola operazione di scrittura. Un eventuale tentativo di lettura produce l'eccezione NotSupportedException.
ReadWrite	Sono ammesse entrambe le operazioni.

Il valore specificato per l'argomento `FileAccess` dev'essere coerente con la modalità di connessione richiesta (argomento `FileMode`). Inoltre, se esso non è specificato viene assunto come valore di default `FileAccess.ReadWrite`.

Seguono quattro esempi che si riferiscono a tentativi di connessione a due file, dei quali soltanto il primo, `"esiste.txt"`, si suppone già esistere.

Tentativo di apertura in lettura di un file già esistente con accesso in sola lettura:

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

Tentativo di apertura in modalità "append" di un file già esistente:

```
FileStream fs = new FileStream("esiste.txt", FileMode.Append);  // errore!
```

La precedente istruzione produce un'eccezione, poiché non essendo stato specificato il tipo di accesso, questo viene assunto per default `FileAccess.ReadWrite`, valore che è incompatibile con la modalità di connessione `FileMode.Append`.

Tentativo di creazione di un nuovo file in sola scrittura:

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

Tentativo di creazione un file esistente:

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

Il contenuto del file viene azzerato.

Livello di condivisione

Con questo argomento è possibile specificare se e a che livello altri processi possono elaborare il file dopo che è stata stabilita una connessione con esso.

Tabella 14-3 Elenco dei livelli di condivisione ammessi

LIVELLO CONDIVISIONE FILESHARE	DESCRIZIONE
None	Vieta qualsiasi forma di condivisione. Qualsiasi tentativo di connessione al file da altri processi viene rifiutato, fintantoché lo stream non viene chiuso.

LIVELLO CONDIVISIONE FILESHARE	DESCRIZIONE
None	Vieta qualsiasi forma di condivisione. Qualsiasi tentativo di connessione al file da altri processi viene rifiutato, fintantoché lo stream non viene chiuso.

Read	Consente a un altro processo di leggere il file (e quindi di aprirlo in sola lettura). Qualsiasi altro tentativo di operazione vengono rifiutati, fintantoché lo stream non viene chiuso.
ReadWrite	Consente a un altro processo di leggere e di scrivere il file.
Write	Consente a un altro processo di scrivere il file (e quindi di aprirlo in sola scrittura). Qualsiasi tentativo di leggere il file viene rifiutato, fintantoché lo stream non viene chiuso.

La gestione del livello di condivisione nell'accesso ai file è un aspetto avanzato e importante della programmazione, che riguarda qualsiasi applicazione realistica e che comunque non sarà preso in considerazione in questo testo. In ogni caso, per mettere alla prova questa funzionalità è sufficiente eseguire il seguente codice:

```
...
FileStream fs = new FileStream("prova.txt", FileMode.Open,
                                            FileAccess.Read,
                                            FileShare.Read);
...
Console.ReadLine();
fs.Close();
...
```

e mentre il programma è in esecuzione – e dunque il file è connesso al programma – tentare con Blocco note di aprirlo, modificarlo e successivamente salvarlo con lo stesso nome. Si nota che:

- il file viene aperto senza problemi, poiché il livello di condivisione FileShare.Read consente ad altri programmi di leggerlo;
- all'atto del salvataggio, Blocco note chiede di specificare un nome diverso, poiché ha riconosciuto che il file è già connesso a un altro programma;
- se viene comunque specificato lo stesso nome e confermata l'operazione di salvataggio, Blocco note visualizza un messaggio – un po' fuorviante – che avverte dell'impossibilità di salvare il file.

Senza chiudere Blocco note e dopo essere ritornati al nostro programma si prema invio; ciò determina l'esecuzione del metodo `fs.Close()`, il quale chiude la connessione con il file. A questo punto si ritorni a Blocco note e si riprovi a salvare il file. L'operazione viene eseguita con successo.

Nota bene: in questo come nei precedenti esempi l'accesso al file avviene sempre mediante un percorso relativo che contiene soltanto il nome; ciò presuppone che il file risieda nella directory corrente. Se non specificato diversamente mediante opportune impostazioni, la directory corrente è la directory nella quale è memorizzato il programma eseguibile. Ciò significa che gli esempi precedenti, così come sono stati scritti, funzionano soltanto se i file processati risiedono nella stessa directory che contiene il programma eseguibile.

1.2 Lettura e scrittura del file

La classe `FileStream` mette a disposizione funzionalità poco sofisticate per quanto riguarda le operazioni di lettura e scrittura del file; quando è necessario un maggior controllo su questo tipo operazioni è necessario affidarsi ad altre classi, che saranno esaminate successivamente.

Per processare il file esistono fondamentalmente due metodi, `Read()` e `Write()`, i quali consentono di leggere o scrivere una sequenza di byte.

Metodo "Read()"

Il prototipo del metodo è:

```
int Read(byte[] vettore, int indice, int lunghezza)
```

Il metodo legge dal file una sequenza di byte pari a `lunghezza`, memorizzandoli in `vettore` a partire dalla posizione specificata da `indice`. L'array `vettore` dev'essere già stato creato prima dell'invocazione del metodo. `Read()` ritorna come valore il numero di byte effettivamente letti, che è minore di `lunghezza` se viene raggiunta la fine del file[7]. Dopo l'operazione, la posizione corrente dello stream, definita dalla proprietà `Position`, viene incrementata del numero di byte letti.

Metodo "Write()"

Il prototipo del metodo è:

```
void Write(byte[] vettore, int indice, int lunghezza)
```

Il metodo scrive sul file i byte memorizzati in `vettore`, partendo da `indice` e per un numero di byte pari a `lunghezza`. Dopo l'operazione, `Position` viene incrementata del numero di byte scritti.

1.3 Posizione corrente e lunghezza dello stream

Queste informazioni sono definite rispettivamente dalle proprietà `Position` e `Length`.

Posizione corrente dello stream: metodo "Seek()"

La proprietà `Position` consente di conoscere, nonché di modificare, la posizione corrente dello stream. In questo senso, lo stream, e dunque il file connesso, può essere visto come un vettore di byte il cui indice del primo elemento è zero. In sostanza, la proprietà `Position` consente un accesso casuale al file esattamente come avviene per l'indice di un array.

La posizione corrente viene aggiornata in modo automatico conseguentemente alle operazioni di lettura e scrittura, ma può essere impostata direttamente sia assegnando un valore alla proprietà `Position`, scrivendo ad esempio:

```
fs.Position = 0;      // imposta posizione corrente all'inizio del file
```

sia invocando il metodo `Seek()`, il cui prototipo è:

[7] A questo proposito il testo di riferimento su .NET afferma che ciò non è garantito, e che dunque il metodo `Read()` "potrebbe" ritornare un valore minore di `lunghezza` anche se non è stata raggiunta la fine del file.

```
long Seek(long distanza, SeekOrigin origine)
```

Mediante `Seek()` è possibile spostare la posizione corrente di un valore pari a `distanza`, relativamente:

- all'inizio: argomento origine uguale a SeekOrigin.Begin;
- alla fine: argomento `origine` uguale a `SeekOrigin.End`;
- all'attuale posizione: argomento `origine` uguale a `SeekOrigin.Current`;

Il metodo ritorna la nuova posizione corrente. Ecco un esempio d'uso:

```
Seek(0, SeekOrigin.Begin);    // imposta posizione corrente a inizio file
```

Lunghezza dello stream e metodo "SetLength()"

La proprietà `Length` è a sola lettura e ritorna il numero dei byte memorizzati nel file. Da notare che il tipo della proprietà è `long` e dunque è necessario usare il cast se si vuole utilizzarla come un intero. Ad esempio:

```
int lunghezzaFile = (int) fs.Length;
```

Esiste un metodo, `SetLength()`, che consente di impostare la lunghezza del file. Il suo prototipo è:

```
void SetLength(long lunghezza)
```

Invocando `SetLength()`, è dunque possibile modificare la dimensione del file in accordo alle seguenti regole:

- se `lunghezza` è minore dell'attuale dimensione del file, questo viene troncato;
- se `lunghezza` è maggiore dell'attuale dimensione de file, il file viene espanso. Il contenuto di questo, a partire dalla vecchia posizione fino alla nuova, è indefinito.

1.4 Impiego della classe "FileStream"

La classe `FileStream` fornisce un accesso al file che potremmo definire di "basso livello", poiché esso viene processato come una sequenza indistinta di byte. Per questo motivo i metodi `Read()` e `Write()` di tale classe vengono raramente impiegati per processare file dotati, dal punto di vista logico, di una determina struttura interna. Diverso è il caso in cui il file non dev'essere elaborato in base alla natura del suo contenuto.

Segue un programma di esempio che, dato il nome di un file esistente, lo "divide" in N parti, memorizzandole in file con lo stesso nome e con estensione "001", "002", "003", eccetera. Per semplicità si presuppone che il nome del file da processare sia privo di estensione.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Nome del file da suddividere:");
        string nomeFile = Console.ReadLine();
        Console.WriteLine("Numero di parti:");
        int numParti = Convert.ToInt32(Console.ReadLine());
        FileStream fsOut;
        FileStream fsIn = new FileStream(nomeFile, FileMode.Open,
        FileAccess.Read);
```

```
        int dimParte = (int) (fsIn.Length / numParti);
        byte[] buffer = new byte[dimParte + 1];
        string nomeParte = null;
        for (int i = 1; i <= numParti; i++)
        {
            nomeParte = string.Format("{0}.{1:000}", nomeFile, i);
            fsOut = new FileStream(nomeParte,
            FileMode.Create, FileAccess.Write);
            fsIn.Read(buffer, 0, dimParte);
            fsOut.Write(buffer, 0, dimParte);
            fsOut.Close();
        }

        // aggiunge all'ultima parte gli eventuali byte restanti
        int byteResidui = (int) (fsIn.Length - fsIn.Position);
        if (byteResidui > 0)
        {
            fsIn.Read(buffer, 0, byteResidui);
            fsOut = new FileStream(nomeParte, FileMode.Append);
            fsOut.Write(buffer, 0, byteResidui);
            fsOut.Close();
        }
        fsIn.Close();
    }
}
```

Come si nota, il programma processa sia il file in ingresso che quelli in uscita in modo indipendente dalla natura del loro contenuto; per questo motivo può essere impiegato per dividere file di qualsiasi tipo, da un documento redatto con Microsoft Word a un programma eseguibile a un file immagine Bitmap, eccetera.

15

Elaborare file di testo

`StreamWriter` e `StreamReader` rientrano nella categoria dei *writer* e dei *reade*r, e sono dunque oggetti in grado di scrivere o leggere in uno stream. `StreamWriter` e `StreamReader` adottano il formato testo per la rappresentazione dei dati, convertendoli quando necessario in forma di stringa. Inoltre `StramWriter` e `StreamReader` possono connettersi direttamente al file senza che sia esplicitamente creato un oggetto `FileStream`. Saranno loro stessi a crearlo automaticamente.

1 Classe “StreamWriter”

La classe `StreamWriter` è un *writer* che collegandosi a uno stream consente di scrivere su di esso in formato testo. Essa mette a disposizione un nutrito insieme di metodi sia per la scrittura dei tipi di dati predefiniti che per la scrittura di una generica sequenza di caratteri, memorizzata su array. Ogni dato viene convertito in stringa prima di essere scritto nello stream; la stringa risultante viene scritta in base alla codifica – UTF-8, ASCII, eccetera – adottata.

Segue un elenco delle proprietà e dei metodi d’uso più comune.

1.1 Proprietà principali della classe “StreamWriter”

Tabella 15-1 Proprietà principali della classe `StreamWriter`

PROPRIETÀ	DESCRIZIONE

`AutoFlush`
 	
`bool`
	Se true fa sì che il writer svuoti il buffer interno dopo ogni istruzione di scrittura, altrimenti i dati saranno effettivamente scritti su disco soltanto quando il buffer è pieno. (get)

`BaseStream`
 	
`stream`
	Riferimento allo stream sottostante allo StreamWriter. (get)

`Encoding`
 	
`encoding`
	Consente di stabilire il tipo di codifica dei caratteri (Unicode, UTF-8, ASCII, eccetera). Il tipo predefinito è UTF-8. (get)

`NewLine`
 	
`string`
	Definisce il terminatore di linea utilizzato. Il terminatore di default è la combinazione "\r\n" (ritorno carrello + nuova linea). (get e set)

1.2 Costruttori della classe “StreamWriter”

Tabella 15-2 Costruttori della classe `StreamWriter`

PROTOTIPO / DESCRIZIONE
`StreamWriter(Stream s)`
Crea uno StreamWriter che scrive sullo stream specificato. Per default viene impostata il tipo di codifica UTF-8. Lo stream dev'essere abilitato alla scrittura (vedi FileAccess).
`StreamWriter (string nomeFile)`
Crea uno StreamWriter che scrive su uno stream creato automaticamente e associato al file specificato. Se nomeFile fa riferimento a un file già esistente il suo contenuto viene cancellato, altrimenti il file viene creato. Per default viene impostata il tipo di codifica UTF-8.
`StreamWriter (string nomeFile, Encoding codifica)`
Crea uno StreamWriter che scrive su uno stream creato automaticamente e associato al file specificato. Se nomeFile fa riferimento a un file già esistente il suo contenuto viene cancellato, altrimenti il file viene creato. Viene applicata la codifica specificata (Unicode, ASCII, UTF-8, UTF-7).
`StreamWriter (string nomeFile, bool aggiungi)`
Crea uno StreamWriter che scrive su uno stream associato automaticamente al file specificato. Se nomeFile fa riferimento a un file già esistente: - se aggiungi è true: i dati vengono aggiunti al precedente contenuto del file; - se aggiungi è false: il precedente contenuto del file viene cancellato; Per default viene impostata il tipo di codifica UTF-8.

Il costruttore della classe si presenta in molte forme (ne sono elencate soltanto alcune), la più semplice delle quali richiede come argomento il nome del file, che sarà creato (o azzerato se già esistente) e aperto in sola scrittura. Internamente, il costruttore crea un oggetto di tipo `FileStream` e lo usa per connettersi al file.

1.3 Metodi della classe “StreamWriter”

Tabella 15-3 Metodi della classe `StreamWriter`

PROTOTIPO / DESCRIZIONE
`void Close()`
Chiude il writer e lo stream sottostante.
`void Write(tipo valore)`
Scrive il valore sullo stream dopo averlo convertito in stringa. La sequenza di caratteri risultante dipende oltre che dal valore anche dal tipo di codifica impiegata.
`void Write(char[] buffer, int indice, int lunghezza)`
Scrive sullo stream i caratteri contenuti in buffer, partendo da indice e per un numero di caratteri pari a lunghezza. La sequenza di byte effettivamente scritti dipende dalla codifica impiegata.

void Write(string formattazione, lista-parametri)
Scrive sullo stream i valori presenti nella lista-parametri dopo averli convertiti in accordo alla stringa formattazione.
WriteLine()
Questo metodo presenta le stesse forme e produce lo stesso effetto del metodo Write(). In più, dopo aver scritto gli argomenti sullo stream, scrive un terminatore di linea.

2 Classe "StreamReader"

La classe `StreamReader` è un *reader* che collegandosi a uno stream consente di leggere da esso in formato testo. Il contenuto del file viene interpretato in base a una determinata codifica, (UTF-8 di default oppure specificata come argomento nel costruttore) la quale deve coincidere con quella effettivamente impiegata per scrivere il file; in caso contrario il risultato delle successive operazioni di lettura sarà imprevedibile.

2.1 Proprietà principali della classe "StreamReader"

Tabella 15-4 Proprietà principali della classe `StreamReader`

PROPRIETÀ	DESCRIZIONE

`BaseStream`
 	
`Stream`
	Riferimento allo stream sottostante allo StreamWriter. (get)

`CurrentEncoding`
 	
`Encoding`
	Restituisce il tipo di codifica dei caratteri utilizzato per leggere dallo stream. (get)

2.2 Costruttori della classe "StreamReader"

Tabella 15-5 Costruttori della classe `StreamReader`

PROTOTIPO / DESCRIZIONE
StreamReader(Stream s)
Crea uno StreamReader che legge dallo stream specificato.
StreamReader(string nomeFile)
Crea uno StreamReader che legge da uno stream creato automaticamente e associato al file specificato. Se nomeFile non fa riferimento a un file già esistente viene sollevata un'eccezione.
StreamReader (string nomeFile, Encoding codifica)
Crea uno StreamReader che legge da uno stream creato automaticamente e associato al file specificato. Se nomeFile non fa riferimento a un file già esistente viene sollevata un'eccezione. Durante la lettura viene applicata la codifica specificata (Unicode, ASCII, UTF-8, UTF-7).

StreamReader (string nomeFile, bool rilevaCodifica)
Crea uno StreamReader che legge da uno stream creato automaticamente e associato al file specificato. Se nomeFile non fa riferimento a un file già esistente viene sollevata un'eccezione. Se rilevaCodifica vale true, il reader usa l'eventuale prologo memorizzato nel file per determinare in modo automatico la codifica impiegata per scriverlo.

2.3 Metodi della classe "StreamReader"

Tabella 15-6 Metodi della classe `StreamReader`

METODO / PROTOTIPO / DESCRIZIONE
void Close()
Chiude il reader e lo stream sottostante.
int Peek ()
Legge e ritorna un carattere dallo stream, o ritorna –1 se è stata raggiunta la fine dello stream. Non avanza l'indicatore di posizione (BaseStream.Position).
int Read()
Legge e ritorna un carattere dallo stream, o ritorna –1 se è stata raggiunta la fine dello stream. Avanza di 1 l'indicatore di posizione (BaseStream.Position).
int Read(char[] buffer, int indice, int lunghezza)
Legge dallo stream un numero di caratteri pari a lunghezza e li memorizza in buffer a partire da indice. Ritorna il numero di caratteri effettivamente letti (0 se era stata raggiunta la fine dello stream).
string ReadLine()
Legge una linea dallo stream e la ritorna come stringa. Se era stata raggiunta la fine dello stream ritorna una stringa nulla. (null). Una linea è definita come una sequenza di caratteri seguita da un terminatore di linea; questo è per default definito dalla sequenza di caratteri "\r\n" (ritorno carrello + nuova linea), la quale non viene comunque memorizzata nella stringa.
string ReadToEnd()
Legge dalla posizione corrente fino alla fine dello stream. Ritorna i caratteri letti sotto forma di stringa. Nella stringa vengono memorizzati anche gli eventuali terminatori di linea.

Il metodo più comune – ma non necessariamente l'unico – impiegato per la lettura è senz'altro `ReadLine()`, il quale legge una riga di testo e cioè una sequenza di caratteri seguita da un terminatore di linea.

3 Uso delle classi "StreamWriter" e "StreamReader"

I file di testo, e dunque le classi `StreamReader` e `StreamWriter`, hanno sostanzialmente due campi di applicazione:

- memorizzazione di informazioni in una forma che sia leggibile dall'utente finale;

- memorizzazione di informazioni in una forma che sia indipendente dalla particolare configurazione binaria usata per rappresentarle in memoria centrale.

Il primo aspetto riguarda la forma testuale in sé nella quale vengono rappresentati i dati, forma che ad esempio consente all'utente di aprire un file di testo con `Blocco note` o qualsiasi altro editor e di interpretarne agilmente il contenuto indipendentemente dalla sua natura (numeri, date, valori monetari, eccetera).

Il secondo aspetto, più generale, riguarda la possibilità di rappresentare i dati in una forma che possa essere facilmente condivisa tra programmi e sistemi operativi diversi. Ciò è reso possibile dal fatto che mentre la rappresentazione di determinati valori in formato binario è strettamente dipendente dall'ambiente (applicazione + sistema operativo + computer) utilizzato, la sua rappresentazione in formato testo può essere universalmente condivisa, purché aderisca ad uno standard comune.

Di seguito sono mostrati due programmi di esempio. Il primo elabora un file di testo impostando a maiuscolo le lettere di ogni parola in esso contenuta. Il secondo affronta il problema della archiviazione delle informazioni relative alla rotta di una nave, mostrando che il formato testo può essere utilizzato per memorizzare informazioni di qualsiasi natura, purché esse siano formattate in modo da poter essere interpretate correttamente nella lettura.

3.1 Esempio n° 1: Impostazione a maiuscolo del "case" delle lettere

Ipotizziamo di avere un file di testo e di voler impostare a maiuscolo il case di tutte le lettere.

Le fasi di lettura e scrittura del file sono estremamente semplici. In lettura, l'intero file viene caricato una riga per volta in una lista tipizzata. In scrittura, il contenuto della lista viene nuovamente riversato nel file, sovrascrivendo il contenuto precedente.

Anche la fase di elaborazione delle righe di testo è molto semplice, poiché si limita a iterare nella collezione di stringhe invocando il metodo `ToUpper()` per ognuna di esse. In questo senso occorre dire che processare un file di testo implica quasi sempre operazioni di lettura e scrittura di facile realizzazione e fasi di elaborazione del contenuto che, al contrario, possono essere anche piuttosto complesse.

```
class Program
{
    static List<string> righe;

    static void LeggiFile(string percorso)
    {
        StreamReader sr = new StreamReader(percorso);
        righe = new List<string>();
        string linea = sr.ReadLine();
        while (linea != null)
        {
            righe.Add(linea);
            linea = sr.ReadLine();
        }
        sr.Close();
    }   // fine lettura file

    static void ScriviFile(string percorso)
    {
        StreamWriter sw = new StreamWriter(percorso);
```

```
        foreach (string linea in righe)
        sw.WriteLine(linea);
        sw.Close();

    }   // fine scrittura file

    static void ElaboraRighe()
    {
        for (int i = 0; i < righe.Count; i++)
        {
            string linea = righe[i];
            righe[i] = linea.ToUpper();
        }
    }

    static void Main()
    {
        Console.WriteLine("Nome del file:");
        string percorso = Console.ReadLine();
        // ...
        LeggiFile(percorso);
        // ...
        Console.WriteLine("File originale:\n");
        foreach (string linea in righe)
            Console.WriteLine(linea);
        // ...
        ElaboraRighe();
        // ...
        ScriviFile(percorso);
        //...
        LeggiFile(percorso);
        Console.WriteLine("\n\nFile modificato:\n");
        foreach (string linea in righe)
            Console.WriteLine(linea);

        Console.ReadKey();
    }
}
```

Due righe di codice sono degne di nota, collocate nel metodo di lettura del file. Quando si tenta di leggere una riga di testo con il metodo `ReadLine()`, questo ritorna un valore `null` se è stata incontrata la fine del file. La lettura delle righe di un file di testo implica dunque un ciclo la cui condizione verifica se l'ultima riga letta vale `null`. Ovviamente, è necessario provare a leggere almeno una riga (ed è questo il motivo dell'invocazione di `ReadLine()` fuori dal ciclo).

3.2 Esempio n° 2: Serializzazione di una classe su un file di testo

Innanzitutto, occorre sottolineare che il formato testo risulta meno appropriato rispetto al formato binario per l'archiviazione dati, poiché:

- le operazioni di lettura e scrittura sono meno efficienti rispetto a quelle formato binario, poiché richiedono una conversione dei dati;
- un file di testo tende ad occupare più memoria di un file binario contenente gli stessi dati;
- benché ammissibile in linea teorica, l'accesso casuale ai dati – operazione che presuppone la memorizzazione degli stessi in record a lunghezza fissa – può risultare complicato da implementare.

Nonostante ciò esistono situazioni nelle quali la rappresentazione di un archivio in formato testo risulta utile, se non addirittura necessaria, ad esempio per la condivisione dei dati tra applicazioni diverse. In questo caso esiste comunque un problema da risolvere, che è quello di stabilire una formattazione dei dati che ne consenta la corretta interpretazione in fase di lettura.

Per chiarire il concetto si consideri l'ipotesi di aver scritto nel file i primi due campi, "codice" e "descrizione", di un articolo. L'esecuzione del seguente codice, collocato in un ipotetico metodo di scrittura dei dati:

```
Articolo art;
// ... si presuppone che la variabile art contenga i dati dell'articolo
// ... e che i primi due campi valgano "10001" e "CD-ROM R/W"
StreamWriter sw = new StreamWriter(percorso);
sw.Write(art.Codice);
sw.Write(art.Descrizione);
```

produce la scrittura della sequenza di caratteri mostrata sotto:

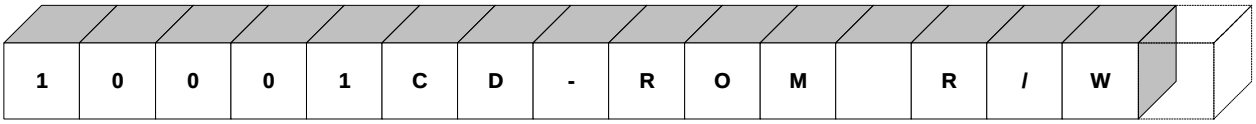

Figura 15-1. Sequenza di caratteri risultante dalla scrittura dei campi "codice" e "descrizione".

Dov'è il confine tra il campo "codice" e il campo "descrizione"? Nel formato binario questo confine è definito implicitamente nell'occupazione di memoria, fissa e conosciuta, dei tipi di dati predefiniti e dal fatto che la memorizzazione dei caratteri di una stringa è preceduta dalla memorizzazione della sua lunghezza. Ma nel formato testo i dati non hanno una lunghezza fissa, poiché vengono convertiti in stringa e dunque la loro dimensione dipende unicamente dal numero di caratteri – e quindi di byte – necessari per rappresentare la stringa risultante; inoltre, le stringhe non vengono precedute da uno o più byte che ne indichino la lunghezza.

Il problema può essere affrontato mediante due approcci distinti, dei quali il secondo è senz'altro più generale e flessibile.

- convertire i dati in stringhe di lunghezza fissa;
- usare dei "delimitatori" e cioè dei caratteri speciali che in fase di lettura vengano riconosciuti non come dati, ma come simboli di confine tra i dati.

Convertire i dati in stringhe di lunghezza fissa

E' senz'altro un approccio molto semplice da implementare. Per ogni campo si stabilisce il numero massimo di caratteri da riservare alla sua rappresentazione in forma di stringa. Se un dato relativo al campo in questione occupa meno caratteri, alla stringa risultante si aggiungono tanti spazi a destra quanti ne servono per raggiungere il numero di caratteri stabilito.

Segue il codice del metodo `ScriviFile()` implementato sulla base di questo approccio; si presuppone il resto del programma simile a quello presentato nel capitolo precedente.

```
static void ScriviFile()
{
    StreamWriter sw = new StreamWriter(percorso);
    for (int i = 0; i < articoli.Count; i++)
    {
        Articolo art = (Articolo) articoli[i];
        sw.WriteLine("{0,-10}{1,-20}{2,-10}{3,-4}", art.Codice,
                                                    art.Descrizione,
                                                    art.Prezzo,
                                                    art.Sconto);
    }
    sw.Close();
}
```

Il file prodotto è rappresentato in figura, aperto con Blocco note:

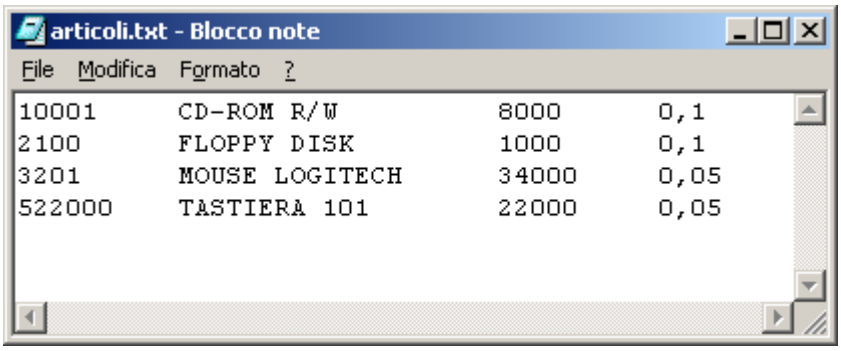

Figura 15-1 File prodotto mediante scrittura in formato testo a campi fissi

In pratica tutto il lavoro di formattazione dei dati è svolto dal metodo `WriteLine()` nella forma in cui accetta come primo argomento una stringa di formattazione, evidenziata in neretto nel codice. Un file così formattato può essere interpretato leggendo una riga per volta ed estraendo da essa i singoli campi:

```
static void LeggiFile()
{
    StreamReader sr = new StreamReader(percorso);
    Articolo art;
    string linea = "";
    do
    {
        linea = sr.ReadLine();
        if (linea != null)
        {
            art.Codice = int.Parse(linea.Substring(0, 10));
            art.Descrizione = linea.Substring(10, 20);
            art.Prezzo = double.Parse(linea.Substring(30, 10));
            art.Sconto = double.Parse(linea.Substring(40, 4));
            articoli.Add(art);
        }
```

```
    }
    while (linea != null);
    sr.Close();
}
```

Il tutto funziona, ma il tipo di formattazione prodotto da questa tecnica presenta degli svantaggi:

- tende a produrre una maggiore occupazione di memoria. Infatti, il limite massimo di caratteri stabilito per un campo può essere anche di molto superiore all'effettivo numero di caratteri richiesto mediamente dei dati relativi al campo in questione. D'altra parte, se anche un solo dato occupa più caratteri del numero massimo stabilito, la formattazione risultante produrrà sicuramente dei problemi in fase di lettura;
- è rigida, poiché è strettamente dipendente al codice che la produce (*hard coded*). Nel file di testo non esiste niente che renda espliciti i confini tra i dati, e dunque è possibile interpretare il contenuto del file soltanto mediante un codice che utilizzi esattamente gli stessi valori per le dimensioni massime dei dati;
- un file di testo scritto mediante questa tecnica può essere interpretato correttamente soltanto dalla stessa applicazione che lo ha prodotto.

Formattazione del testo mediante "delimitatori"

E' l'approccio più comune e consiste nel separare i dati da simboli particolari, chiamati delimitatori, che possono essere riconosciuti come tali in fase di lettura. Un file così formattato si chiama "file di testo delimitato". Requisito fondamentale di questa tecnica è che:

i simboli usati come delimitatori non facciano parte dell'insieme di caratteri potenzialmente presenti nei dati.

Ciò è abbastanza ovvio: in fase di lettura è necessario poter distinguere i caratteri appartenenti ai dati da quelli appartenenti ai delimitatori, altrimenti non sarà possibile estrarre i dati stessi dalla riga di testo.

> In realtà questo non è un requisito così stringente, poiché esistono vari modi per superarlo. Per semplicità, comunque, si farà l'ipotesi che il carattere usato come delimitatore non possa fare parte della rappresentazione in forma di stringa dei dati.

Per semplificare il codice, qui decidiamo di usare un formato delimitato piuttosto primitivo, che fa uso di un singolo carattere delimitatore, e cioè il carattere '|'.

```
static void ScriviFile()
{
    StreamWriter sw = new StreamWriter(percorso);
    for (int i = 0; i < articoli.Count; i++)
    {
        Articolo art = (Articolo) articoli[i];
        sw.WriteLine("{0}|{1}|{2}|{3}", art.Codice,
                                        art.Descrizione,
                                        art.Prezzo,
                                        art.Sconto);
    }
    sw.Close();
}
```

Il file prodotto è rappresentato in figura, aperto con Blocco note:

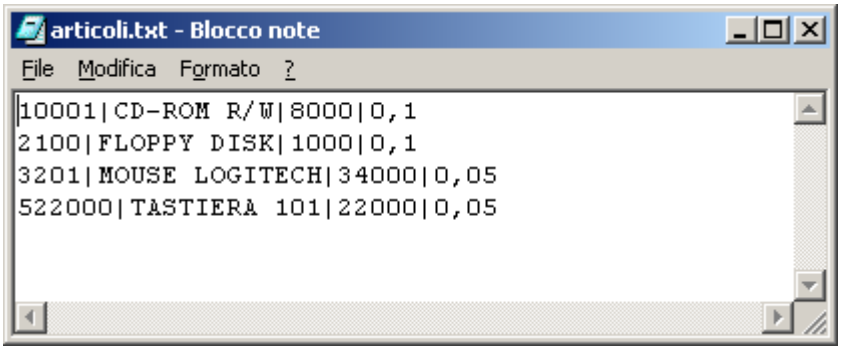

Figura 15-2 File prodotto mediante scrittura in formato testo delimitato

In fase di lettura è possibile estrarre i dati da una riga servendosi appunto del delimitatore inserito tra di essi:

```
static void LeggiFile()
{
    StreamReader sr = new StreamReader(percorso);
    Articolo art;
    string linea = "";
    do
    {
        linea = sr.ReadLine();
        if (linea != null)
        {
            string[] campi = linea.Split('|');
            art.Codice = int.Parse(campi[0]);
            art.Descrizione = campi[1];
            art.Prezzo = double.Parse(campi[2]);
            art.Sconto = double.Parse(campi[3]);
            articoli.Add(art);       }
    }
    while (linea != null);
    sr.Close();
}
```

Di particolare interesse l'istruzione:

```
string[] campi = linea.Split('|');
```

L'invocazione del metodo `Split()` produce un vettore delle sotto stringhe delimitate dal carattere (o i caratteri) specificato come argomento.

Rispetto alla formattazione mediante campi fissi, il formato testo delimitato ha i seguenti vantaggi:

- occupa meno memoria;
- è maggiormente indipendente dal codice che lo produce, poiché l'unica assunzione fatta è sul simbolo impiegato come delimitatore e non sulla dimensione dei campi, che può essere variabile.

L’ultimo punto implica che, stabilita una convenzione comune sul simbolo dal usare come delimitatore, due applicazioni diverse sono in grado di processare lo stesso file interpretandolo correttamente.

Il formato testo delimitato impiegato nell’esempio è relativamente banale, ne esistono di più sofisticati, riconoscibili da un vasto numero di applicazioni; uno di essi è il seguente:

"10001", "CD-ROM R/W", "8000", "0,1"

nel quale ogni campo è racchiuso da virgolette e separato dagli altri campi da una virgola.

16

Accesso al *file system*

1 Introduzione

Il *namespace* `System.IO` definisce quattro classi per l'accesso agli oggetti – file e directory – del *filesystem*: `File`, `Directory`, `FileInfo`, `DirectoryInfo`. Pur essendo classi di natura diversa (le prime due espongono solo funzioni statiche, mentre le altre due solo funzioni d'istanza) consentono di eseguire sostanzialmente le stesse forme di elaborazione; per questo motivo saranno trattate soltanto le classi `File` e `Directory`.

Le classi `File` e `Directory` processano gli oggetti del *file system* senza accedere realmente al loro contenuto, consentendo di eseguire su di essi le operazioni tipiche – copia, cancellazione, elencazione, modifica degli attributi, eccetera – necessarie per la gestione di un *file system*.

I metodi esposti dalla classe `File` possono essere raggruppati in due categorie distinte.

- metodi per l'accesso e l'elaborazione degli oggetti del *file system*, prevalentemente file ma in alcuni casi anche directory. Tali metodi operano indipendentemente dal formato interno e dal contenuto degli oggetti processati;

- metodi per la creazione e l'apertura di file con la conseguente creazione di oggetti di accesso al contenuto: `FileStream`, `StreamReader`, `StreamWriter`.

Anche la classe `Directory`, come la classe `File`, espone metodi per l'accesso e l'elaborazione di directory: creazione, copia, spostamento, elencazione del file, eccetera.

Alcuni metodi esposti da entrambe le classi sono perfettamente omologhi e possono processare file o directory indipendentemente dalla classe attraverso la quale vengono invocati. Ad esempio, il metodo `GetCreationTime()`, che ritorna la data e l'ora di creazione di un file o una directory, è esposto da entrambe le classi e opera indifferentemente su file o directory.

Il namespace `System.IO` definisce inoltre la classe `Path`. Essa espone alcuni campi pubblici e metodi statici molti utili per manipolare percorsi di file e directory. La maggior parte di questi metodi non accedono realmente al *file system*, limitandosi a processare la stringa o le stringhe che ricevono come argomenti senza verificare che esse facciano riferimento a oggetti del *file system* realmente esistenti.

2 Classe “File”

2.1 Metodi della classe “File”

Metodi che generano oggetti per la lettura/scrittura di un file

Tabella 16-1 Metodi della classe `File`

METODO / PROTOTIPO / DESCRIZIONE
`FileStream Create(string percorso)`
Crea o apre il file designato da percorso e lo associa a un FileStream. L’eventuale precedente contenuto del file viene sovrascritto.
`StreamWriter CreateText(string percorso)`
Crea o apre il file designato da percorso e lo associa a uno StreamWriter. L’eventuale precedente contenuto del file viene sovrascritto. (Il testo è codificato in UTF-8)
`FileStream Open(string percorso, FileMode modo)`
Apre il file designato da percorso e lo associa a un FileStream. La modalità di creazione o apertura è disegnata da modo (vedi classe FileStream). Se il file non esiste viene sollevata un’eccezione.
`FileStream OpenRead(string percorso)`
Apre in lettura il file designato da percorso e lo associa a un FileStream. Se il file non esiste viene sollevata un’eccezione.
`StreamReader OpenText(string percorso)`
Apre in lettura il file designato da percorso e lo associa a uno StreamReader. Se il file non esiste viene sollevata un’eccezione.
`FileStream OpenWrite(string percorso)`
Apre in lettura/scrittura il file designato da percorso e lo associa a un FileStream. Se il file non esiste viene sollevata un’eccezione.

Tabella 16-2 Metodi della classe `File (continua)`

METODO / PROTOTIPO / DESCRIZIONE
`StreamWriter AppendText(string percorso)`
Crea o apre il file designato da percorso e lo associa a un StreamWriter. Il testo scritto mediante lo StreamWriter è aggiunto al precedente contenuto del file. (Il testo è codificato in UTF-8)
Apre in lettura/scrittura il file designato da percorso e lo associa a un FileStream. Se il file non esiste viene sollevata un'eccezione.
`string[] ReadAllLines(string percorso)`
Apre un file di testo, legge tutte le linee e chiude il file
`string ReadAllText(string percorso)`
Apre un file di testo, legge tutte le linee e chiude il file
`void WriteAllLines(string percorso, string[] linee)`
Crea un nuovo file scrivendo tutte le line contenute nel vettore. Se il file già esiste, lo sovrascrive
`void WriteAllText(string percorso, string stringa)`
Crea un nuovo file scrivendo il contenuto della stringa. Se il file già esiste, lo sovrascrive

Metodi per copiare/cancellare/spostare/ottenere informazioni su un file

Tabella 16-3 Metodi della classe `File`

METODO / PROTOTIPO / DESCRIZIONE
`void Copy(string sorg, string dest)`
Crea una copia del file designato da sorg e la nomina come dest. Se dest fa riferimento a un file esistente viene sollevata un'eccezione.
`void Copy(string sorg, string dest, bool sovrascrive)`
Crea una copia del file designato da sorg e la nomina come dest. Se dest fa riferimento a un file esistente e sovrascrive è true il contenuto del file viene sovrascritto, altrimenti viene sollevata un'eccezione.
`void Delete(string percorso)`
Elimina il file designato da percorso.
`bool Exists(string percorso)`
Ritorna true se il file designato da percorso esiste, false altrimenti.

FileAttributes GetAttributes(string percorso)
Ritorna gli attributi associati al file o alla directory designati da percorso.
DateTime GetCreationTime(string percorso)
Ritorna data e ora di creazione del file o della directory designati da percorso.
DateTime GetLastAccessTime(string percorso)
Ritorna data e ora dell'ultimo accesso al file o alla directory designati da percorso.
DateTime GetLastWriteTime(string percorso)
Ritorna data e ora dell'ultimo accesso in scrittura al file o alla directory designati da percorso.
void Move(string sorg, string dest)
Muove il file designato da sorg in dest. Se il file designato da dest esiste già viene sollevata un'eccezione. Inoltre dest non può fare riferimento a una directory.

Tabella 16-4 Metodi della classe `File` (continua)

METODO / PROTOTIPO / DESCRIZIONE
void SetAttributes(string percorso, FileAttributes attributi)
Imposta gli attributi specificati sul file o sulla directory designati da percorso.
void GetCreationTime(string percorso, DateTime data)
Imposta data e ora di creazione sul file designato da percorso.
void SetLastAccessTime(string percorso, DateTime data)
Imposta data e ora dell'ultimo accesso al file designato da percorso.
void SetLastWriteTime(string percorso, DateTime data)
Imposta data e ora di dell'ultimo accesso in scrittura al file designato da percorso.

2.2 Attributi di un file o una directory: tipo "FileAttributes".

Attraverso l'enumeratore `FileAttributes` è possibile conoscere e modificare gli attributi di un file, attributi che a loro volta possono essere impostati mediante funzionalità del sistema operativo.

Tabella 16-5 Attributi di base associati a un file o a una directory: tipo `FileAttributes`

ATTRIBUTO	DESCRIZIONE
Archive	Molte applicazioni usano questo attributo per marcare i file per operazioni di backup.
Compressed	Il file o la directory è compresso.
Directory	L'oggetto è una directory.
Hidden	L'oggetto –file o directory – è nascosto e dunque non è incluso nelle normali operazioni di elencazione dei file della directory.
ReadOnly	Il file o la directory è di sola lettura.
System	Il file è parte del sistema operativo o viene usato esclusivamente da esso.
Temporary	Il file è temporaneo e dunque la maggior parte del suo contenuto viene mantenuto in memoria per una accesso più veloce. Un file temporaneo dovrebbe essere eliminato appena non è più necessario per l'applicazione.

3 Classe "Directory"

3.1 Metodi della classe "Directory"

Tabella 16-6 Metodi della classe `Directory`

METODO / PROTOTIPO / DESCRIZIONE
`DirectoryInfo CreateDirectory(string percorso)`
Crea la directory designata da percorso. Se percorso contiene delle directory intermedie, anch'esse vengono create, a meno che non siano già esistenti. Ad esempio l'istruzione: `Directory.CreateDirectory(@"Scuola\Compiti\Mat")` crea le directory "Scuola", "Compiti" e "Mat" a partire dalla directory corrente. Ma se "Scuola" e "Compiti" già esistono crea soltanto "Mat". Il metodo ritorna un oggetto di tipo DirectoryInfo associato alla directory creata.
`void Delete(string percorso)`
Elimina la directory designata da percorso, ma soltanto se è vuota.
`void Delete(string percorso, bool sottodir)`
Elimina la directory designata da percorso e tutte le directory e i file in essa contenuti se sottodir è true.
`bool Exists(string percorso)`

Ritorna true se esiste la directory designata da percorso, false altrimenti
GetCreationTime() Vedi metodo equivalente della classe `File`.
string GetCurrentDirectory()
Ritorna il percorso assoluto della directory corrente.
string[] GetDirectories(string percorso)
Ritorna un vettore dei percorsi assoluti delle sotto directory contenute nella directory designata da percorso.
string[] GetDirectories(string percorso, string pattern)
Ritorna un vettore dei percorsi assoluti delle sotto directory contenute nella directory designata da percorso che verificano il pattern specificato.

Tabella 16-7 Metodi della classe `Directory`

METODO / PROTOTIPO / DESCRIZIONE
string GetDirectoryRoot(string percorso)
Ritorna la directory radice, compresa l'unità di volume (se esiste), del percorso specificato. Il percorso non deve obbligatoriamente fare riferimento a un file o a una directory esistenti.
string[] GetFiles(string percorso)
Ritorna un vettore dei percorsi assoluti dei file contenuti nella directory designata da percorso. (Esiste una seconda versione del metodo che accetta come argomento un pattern).
string[] GetFileSystemEntries(string percorso)
Ritorna un vettore dei percorsi assoluti dei file contenuti e delle sotto directory contenute nella directory designata da percorso. (Esiste una seconda versione del metodo che accetta come argomento un pattern).
GetLastAccesTime() Vedi metodo equivalente della classe `File`.
GetLastWriteTime() Vedi metodo equivalente della classe `File`.
string[] GetLogicalDrives()
Ritorna un vettore dei drive logici presenti nel computer. Ogni stringa presenta la forma: "<nome unita>:\"
DirectoryInfo GetParent(string percorso)
Ritorna un oggetto DirectoryInfo associato alla directory che contiene quella designata da percorso, o null se percorso è la directory radice
void Move(string sorg, string dest)
Muove il file o la directory designata da sorg in dest. Se sorg fa riferimento a una directory, tutto il suo contenuto viene spostato. Se dest fa riferimento a una directory esistente viene sollevata un'eccezione.

SetCreationTime()	Vedi metodo equivalente della classe `File`.
void SetCurrentDirectory (string percorso)	
Imposta percorso come directory corrente.	
SetLastAccessTime()	Vedi metodo equivalente della classe `File`.
SetLastWriteTime()	Vedi metodo equivalente della classe `File`.

3.2 Pattern e caratteri jolly

I metodi sovraccaricati `GetDirectories()`, `GetFiles()` e `GetFileSystemEntries()` accettano come secondo argomento una stringa che consente di restringere i nomi di file o directory che saranno memorizzati nel vettore a quelli il cui nome rispecchia un determinato modello, o *pattern*. Questo è caratterizzato dall'impiego dei cosiddetti caratteri jolly, '*****' e '**?**', i quali hanno il seguente significato:

Tabella 16-8 Caratteri jolly (*wildcards*)

CARATTERE JOLLY	DESCRIZIONE (EQUIVALE A:)
*****	Qualsiasi configurazione di zero o più caratteri. Esempio: "*t" corrisponde a qualsiasi stringa che termina con la lettera 't'; "s*" corrisponde a qualsiasi stringa che inizia con la lettere 's'.
?	Un carattere qualsiasi. Esempio: "??t" corrisponde a qualsiasi stringa che termina con la lettera 't' e abbia esattamente tre caratteri; "mat??0*" corrisponde a qualsiasi stringa che inizia con "mat" e termina con '0' e abbia almeno 6 caratteri.

Ad esempio, il seguente codice visualizza l'elenco dei file con estensione ".DOC" contenuti nella directory corrente.:

```
string curDir = Directory.GetCurrentDirectory();
string[] documenti = Directory.GetFiles(curDir, "*.DOC");
foreach(string nomeDoc in documenti)
    Console.WriteLine(nomeDoc);
```

4 Classe "Path"

4.1 Campi pubblici classe "Path"

Tabella 16-9 Campi pubblici (readonly della classe `Path`

NOME CAMPO - TIPO	DESCRIZIONE – EQUIVALE A:

AltDirectorySeparatorChar char	Fornisce un carattere alternativo a quello memorizzato nel campo DirectorySeparatorChar. Nei sistemi Windows è il carattere '/'.
DirectorySeparatorChar char	Carattere usato per la separazione delle directory all'interno di un percorso. E' dipendente dal sistema operativo. In Windows è il carattere '\' (barra retroversa, o *back slash*).
InvalidPathChars char[]	Elenco di caratteri che non possono essere usati all'interno di un percorso.
PathSeparator char	Carattere usato per separare i percorsi specificati nelle variabili di ambiente. E' dipendente dal sistema operativo. In Windows è il carattere ';' (punto-e-virgola).
VolumeSeparatorChar char	Carattere usato per separare un percorso dal nome dell'unita logica. E' dipendente dal sistema operativo. In Windows è il carattere ':' (due-punti).

4.2 Metodi principali della classe "Path"

Tabella 16-10 Metodi principali della classe `Path`

METODO / PROTOTIPO / DESCRIZIONE	
ChangeExtension()	**string** ChangeExtension(**string** percorso, **string** ext)
Modifica l'estensione presente in `percorso` in base al parametro `ext`. Se `percorso` non contiene un'estensione, questa viene aggiunta.	
Combine()	**string** Combine(**string** percorso1, **string** percorso2)
Concatena i due percorsi ritornando il percorso risultante. Se `percorso2` contiene un nome di unità oppure è un percorso UNC viene sollevata un'eccezione. Se `percorso2` è un percorso assoluto viene ritornato direttamente dal metodo senza che vi sia alcuna concatenazione. I due argomenti possono contenere carattery jolli. Ad esempio, la seguente invocazione del metodo è corretta: `string percorso = Combine("Documenti", "*.doc");` e produce come risultato: `"Documenti\*.doc";`	
GetDirectoryName()	**string** GetDirectoryName(**string** percorso)
Ritorna la parte che disegna la directory del percorso specificato. Viene ritornata una stringa nulla se `percorso` è la directory radice oppure non contiene nomi di directory.	
GetExtension()	**string** GetExtension(**string** percorso)
Ritorna l'estensione del percorso specificato. Ritorna una stringa nulla se `percorso` non contiene un'estensione. La stringa ritornata è comprensiva del carattere '.' (punto). Ad esempio: `GetExtension("testo.doc");` ritorna la stringa `".doc"`.	
GetFileName()	**string** GetFileName(**string** percorso)
Ritorna il nome file completo di estensione relativamente al percorso specificato. Ritorna una stringa nulla se l'ultimo carattere del percorso equivale a `DirectorySeparatorChar`. (E cioè una barra retroversa.)	
GetFileNameWithoutExtension()	

string GetFileNameWithoutExtension(**string** percorso)	
Ritorna il nome file senza estensione relativamente al percorso specificato. Ritorna una stringa nulla se l'ultimo carattere del percorso equivale a `DirectorySeparatorChar`. (E cioè una barra retroversa.)	
GetFullPath()	**string** GetFullPath(**string** percorso)
Ritorna il percorso completo relativamente al percorso specificato. Questo metodo fa riferimento alla directory corrente per generare il percorso assoluto. Non è richiesto che il percorso specificato esista realmente, d'altra parte se esiste e se il codice chiamante non ha l'autorizzazione ad accedervi viene sollevata un'eccezione. `GetFullPath()`, dunque, diversamente dal altri metodi accede effettivamente al *file system*.	
GetPathRoot()	**string** GetPathRoot(**string** percorso)
Ritorna la directory radice comprensiva del nome dell'unità relativamente al percorso specificato. Ritorna una stringa nulla se il percorso specificato non è assoluto. Questo metodo non accede al *file system* e dunque non verifica l'effettiva esistenza del percorso.	

Tabella 16-11 Metodi principali della classe `Path` (continua)

METODO / PROTOTIPO / DESCRIZIONE	
GetTempFileName()	**string** GetTempFileName()
Crea un file temporaneo e ne ritorna il nome (sicuramente univoco). Questo metodo accede al *file system*.	
GetTempPath()	**string** GetTempPath()
Ritorna il percorso della directory temporanea del sistema. Questo metodo accede al *file system* e dunque solleva un'eccezione se il codice chiamante non ha i permessi per accedere a questa informazione.	
HasExtension()	**bool** HasExtension(**string** percorso)
Ritorna `true` se il percorso specificato include una estensione, `false` altrimenti.	
IsPathRooted ()	**bool** IsPathRooted (**string** percorso)
Ritorna `true` se il percorso specificato è assoluto, `false` altrimenti.	

5 Classi FileInfo e DirectoryInfo

Abbiamo detto che le classi File e Directory sono classi; ne esistono anche altre due versioni chiamate FileInfo e Directory Info che consentono di svolgere quasi le medesime operazioni ma su variabili d'istanza.

In linea generale se si deve effettuare una sola operazione in un file è bene utilizzare le classi statiche File e Directory, mentre se si devono compiere più di un'operazione si dovrebbero usare le loro versioni "Info".

Questo perché, nel primo caso non si ha spreco di tempo e memoria per distanziare un oggetto che si utilizzerà solo una volta.

Nel secondo caso, invece, avremo un aumento di prestazioni in quanto il controllo di sicurezza sulle credenziali di accesso al file viene svolto una sola volta nel caso di variabili di tipo FileInfo o Directory Info, inoltre, nel momento della costruzione di oggetti di questo tipo, viene effettuato un solo accesso al file per recuperare tutte le informazioni come nome, attributi, etc.

Infatti sono presenti in queste classi proprietà che restituiscono queste informazioni (al contrario delle loro versioni statiche che hanno bisogno di invocare dei metodi, come nell'esempio che segue:

```
string percorso = @"..\..\prova.txt";
FileAttributes attributiFile = File.GetAttributes(percorso);
DateTime ultimoAccesso = File.GetLastAccessTime(percorso);

// stesso esempio ma con FileInfo
FileInfo provaFileInfo = new FileInfo(percorso);
FileAttributes attributiInfo = provaFileInfo.Attributes;
DateTime ultimoAccessoInfo = provaFileInfo.LastAccessTime;
```

6 Uso delle classi "File", "Directory" e "Path"

Segue un programma di esempio che mostra l'uso di alcuni dei metodi esposti dalle classi `File`, `Directory` e `Path`. Il programma, se pur in modo molto semplificato, è una Applicazione Console che emula il funzionamento del comando DEL, il quale consente di cancellare file e directory i cui nomi sono passati come argomenti sulla linea di comando.

> Per eseguire il comando DEL è necessario aprire una finestra console. Analogamente, per passare agli argomenti al programma Elimina è necessario eseguirlo da una finestra console oppure creare un collegamento e accedere alle proprietà del programma per specificare gli argomenti sulla linea di comando.

6.1 Requisiti del programma "Elimina"

Il programma accetta un argomento obbligatorio che rappresenta il percorso del file o della directory da eliminare. Mediante l'uso di caratteri jolly è possibile eliminare gruppi di file o directory.

Il programma accetta inoltre le seguenti opzioni:

- '**-v**': visualizza un messaggio per ogni operazione eseguita (con o senza successo);
- '**-q**': non chiede conferma dell'eliminazione di un file o una directory. Per default il programma chiede la conferma di ogni eliminazione.
- '**-d**': il programma elimina anche le directory che verificano il *pattern* specificato . Per default le directory non vengono processate.

Ecco un esempio d'uso del programma:

```
ELIMINA *.doc –v –q
```

Così eseguito, il programma elimina tutti i file con estensione ".DOC" presenti nella directory corrente, visualizzando le operazioni compiute e senza chiedere conferma delle stesse.

Nel seguente caso, invece:

```
ELIMINA c:\documenti\*.* –v -d
```

il programma elimina tutti i file e le directory contenute nella cartella "C:\Documenti", chiedendo conferma per ogni eliminazione.

> Un simile programma deve essere usato con molta attenzione, poiché è in grado di eliminare con estrema facilità interi alberi di directory, senza collocare nel "Cestino" gli oggetti eliminati.

6.2 Struttura generale del programma

Il programma, attraverso il parametro `args` del metodo `Main()`, esamina innanzitutto la riga di comando, verificando e interpretando gli argomenti specificati (non è stata implementata una forma di verifica molto stringente) e impostando le variabili booleane corrispondenti alle opzioni specificate.

Successivamente esamina il primo argomento – il percorso – verificando se contiene o meno dei caratteri jolly. In caso negativo, esso fa riferimento a un singolo oggetto del *file system* e dunque il programma si limita a invocare il metodo di eliminazione della classe appropriata – `File` o `Directory` – oppure segnala che il percorso è inesistente o non valido.

Se vi sono dei caratteri jolli, dal percorso vengono estratti la parte relativa alla directory e quella relativa al nome del file, contenente i caratteri in questione; questa funge da stringa modello per la ricerca di tutti gli oggetti – file o directory – il cui nome verifica il *pattern* specificato.

Vengono prima estratti tutti i nomi di file, per i quali viene invocato il metodo di eliminazione della classe `File`. Successivamente, se è stata specificata l'opzione '**`-d`**', vengono estratti i nomi di directory, per i quali viene invocato il metodo `Delete()` della classe `Directory`.

Segue l'esame di alcuni frammenti di codice nei quali viene fatto uso dei metodi presentati nei precedenti paragrafi.

6.3 Individuazione degli oggetti – file o directory – da eliminare

Questa parte contiene la logica principale e la sua esecuzione presuppone che l'utente abbia invocato il programma specificando gli argomenti nel modo corretto. In grigio sono evidenziate le istruzioni che contengono chiamate ai metodi delle classi `File`, `Directory` e `Path`:

```
if (percorso.IndexOfAny(new char[] {'*', '?'}) != -1)
{
    string nomeDir = Path.GetDirectoryName(percorso);
    string pattern = Path.GetFileName(percorso);
    if (nomeDir == "")
        nomeDir = Directory.GetCurrentDirectory();
    else
        if (!Directory.Exists(nomeDir) || !SeDirectory(nomeDir))
        {
            Console.WriteLine("\aErrore: " + percorsoNonValido);
            return;
        }

    string[] nomiFile = Directory.GetFiles(nomeDir, pattern);
    foreach(string nome in nomiFile)
        EliminaFile(nome);

    if (eliminaDirectory)
    {
        string[] nomiDirectory = Directory.GetDirectories(nomeDir, pattern);
```

```
        foreach(string nome in nomiDirectory)
            EliminaDirectory(nome);
    }
}
else
{
    if (File.Exists(percorso))
        EliminaFile(percorso);
    else
        if (Directory.Exists(percorso))
            EliminaDirectory(percorso);
        else
            Console.WriteLine("\aErrore: " + percorsoNonValido);
}
```

Il codice è suddiviso in due parti, in base al fatto che il percorso contenga o meno dei caratteri jolli, verifica che si ottiene con testando la condizione:

```
percorso.IndexOfAny(new char[] {'*', '?'}) != -1
```

Esame di un percorso che contiene caratteri jolly

Vengono innanzitutto estratte la parte relativa alla directory e quella relativa al nome del file, che contiene (si suppone) il *pattern* di ricerca dei file e/ delle directory da eliminare:

```
string nomeDir = Path.GetDirectoryName(percorso);
string pattern = Path.GetFileName(percorso);
```

Se l'utente ha specificato il solo *pattern*, la variabile `nomeDir` assume il valore stringa nulla; in questo caso viene assunta come directory quella corrente, altrimenti viene verificato se la directory esiste realmente. In caso negativo viene emesso un errore e il programma termina.

```
if (nomeDir == "")
    nomeDir = Directory.GetCurrentDirectory();
else
    if (!Directory.Exists(nomeDir))
    {
        Console.WriteLine("\aErrore: " + percorsoNonValido);
        return 1;
    }
```

Dopodiché vengono estratti i nomi dei file che verificano il *pattern* specificato; per ognuno di essi viene invocato il metodo di eliminazione:

```
string[] nomiFile = Directory.GetFiles(nomeDir, pattern);
foreach(string nome in nomiFile)
    EliminaFile(nome);
```

Se è stata specificata l'opzione '**-d**' viene eseguito un codice del tutto analogo per le directory:

```
if (eliminaDirectory)
{
    string[] nomiDirectory = Directory.GetDirectories(nomeDir, pattern);
```

```
    foreach(string nome in nomiDirectory)
        EliminaDirectory(nome);
}
```

6.4 Eliminazione dei file e delle directory

L'eliminazione di file e directory avviene mediante due metodi distinti, `EliminaFile()` ed `EliminaDirectory()`, dalla struttura del tutto analoga; essi invocano i metodi corrispondenti delle classi `File` e `Directory` all'interno di un costrutto `try...catch`.

```
static void EliminaFile(string percorso)
{
    if (!ConfermaEliminazione(percorso))
        return;
    try
    {
        File.Delete(percorso);
        VisualizzaMessaggio("Eliminato file: [{0}]", percorso);
        numFileEliminati++;
    }
    catch (IOException)
    {
        VisualizzaMessaggio(erroreEliminazione, percorso);
        return;
    }
}

static void EliminaDirectory(string percorso)
{
    if (!ConfermaEliminazione(percorso))
        return;
    try
    {
        Directory.Delete(percorso, true);
        VisualizzaMessaggio("Eliminata directory: [{0}]", percorso);
        numDirEliminate++;
    }
    catch (IOException)
    {
        VisualizzaMessaggio(erroreEliminazione, percorso);
        return;
    }
}
```

Il valore `true` passato come secondo argomento al metodo `Delete()` della classe `Directory` fa sì che la directory venga eliminata con tutti i file e le sotto directory che eventualmente contiene; in caso contrario il metodo rimuoverebbe soltanto le directory vuote.