Capitolo 10

Input/Output

Lorenzo Bettini

Introduzione

In questo capitolo verrà illustrato il package java.io, che supporta il sistema fondamentale di input/output (I/O) di Java.

Nei programmi Java vengono spesso utilizzate istruzioni per stampare sullo schermo delle stringhe; utilizzare l'interfaccia a caratteri, invece che quella grafica, risulta molto comodo sia per scrivere esempi semplici, che per stampare informazioni di debug. Del resto se si scrive un'applicazione che utilizza intensamente la grafica, è comunque possibile stampare informazioni in una finestra di testo. In effetti il supporto di Java per l'I/O della console (testo) è un po' limitato, e presenta qualche complessità di utilizzo, anche nei programmi più semplici.

Comunque Java fornisce un ottimo supporto per l'I/O per quanto riguarda i file e la rete, tramite un sistema stabile e coerente. Si tratta di un ottimo esempio di libreria orientata agli oggetti che permette di sfruttare a pieno le feature della programmazione object oriented. Una volta compresi i concetti fondamentali dell'I/O di Java, è semplice sfruttare la parte restante del sistema I/O e, se si progettano le proprie classi tenendo presente la filosofia object oriented, si noterà come tali classi saranno riutilizzabili, ed indipendenti dal particolare mezzo di input/output.

Stream

I programmi in Java comunicano (cioè effettuano l'I/O) tramite gli *stream* (in italiano *flussi*). Uno stream è un'astrazione ad alto livello che produce o consuma informazioni: rappresenta una connessione a un canale di comunicazione. Uno stream quindi è collegato a un dispositivo

fisico dal sistema I/O di Java. Gli stream possono sia leggere da un canale di comunicazione che scrivere su tale canale: quindi si parla di stream di input, e stream di output.

Gli stream si comportano in modo omogeneo, indipendentemente dal dispositivo fisico con cui sono collegati (da qui il concetto di astrazione ad alto livello). Infatti le stesse classi e gli stessi metodi di I/O possono essere applicati a qualunque dispositivo. Uno stream (astratto) di input può essere utilizzato per leggere da un file su disco, da tastiera, o dalla rete; allo stesso modo uno stream di output può fare riferimento alla console (e quindi scrivere sullo standard output), a un file (e quindi scrivere e aggiornare un file), o ancora ad una connessione di rete (e quindi spedire dei dati in rete).

Un flusso quindi rappresenta un'estremità di un canale di comunicazione a un senso solo. Le classi di stream forniscono metodi per leggere da un canale o per scrivere su un canale. Quindi un *output stream* scrive dei dati su un canale di comunicazione, mentre un *input stream* legge dati da un canale di comunicazione. Non esistono delle classi di stream che forniscano funzioni sia per leggere che per scrivere su un canale. Se si desidera sia leggere che scrivere su uno stesso canale di comunicazione si dovranno aprire due stream (uno di input ed uno di output) collegati allo stesso canale.

Di solito un canale di comunicazione collega uno stream di output al corrispondente stream di input. Tutti i dati scritti sullo stream di output, potranno essere riletti (nello stesso ordine) dallo stream di input. Poiché, come si è già detto, gli stream sono indipendenti dal particolare canale di comunicazione, essi mettono a disposizione uno strumento semplice e uniforme per la comunicazione fra applicazioni. Due applicazioni che si trovano su due macchine diverse, ad esempio, potrebbero scambiarsi i dati tramite uno stream collegato alla rete, oppure un'applicazione può semplicemente comunicare con l'utente utilizzando gli stream collegati alla console. Gli stream implementano una struttura FIFO (*First In First Out*), nel senso che il primo dato che sarà scritto su uno stream di output sarà il primo che verrà letto dal corrispondente stream di input. Fondamentalmente, quindi, gli stream mettono a disposizione un accesso sequenziale alle informazioni scambiate.

Quando si parla di input/output, si parla anche del problema dell'azione bloccante di una richiesta di input (il concetto di input/output tra l'altro si ritrova anche nelle architetture dei processori). Ad esempio, se un thread cerca di leggere dei dati da uno stream di input che non contiene dati, verrà bloccato finché non saranno presenti dei dati disponibili per essere letti. In effetti, quando un thread cerca di leggere dei caratteri immessi da un utente da tastiera, rimarrà in attesa finché l'utente non inizierà a digitare qualcosa. Il problema dell'azione bloccante è valido anche per le operazioni di output: se si cerca di scrivere qualcosa in rete, si rimarrà bloccati finché l'operazione non sarà terminata. Questo può avvenire anche quando si scrive su un file su disco, ma le operazioni in rete di solito sono le più lente.

Il thread bloccato sarà risvegliato solo quando sarà stata completata l'operazione bloccante. Se si vuole evitare di essere bloccati da queste operazioni si dovrà utilizzare il multithreading; si vedranno degli esempi nel capitolo che riguarda il networking.

Le classi

Le classi degli stream sono contenute nel pacchetto `java.io`, che dovrà quindi essere incluso nei programmi che ne fanno uso.

Tutti gli stream fanno parte di una gerarchia. In realtà si hanno due sottogerarchie: una per gli stream di output ed una per quella di input.

In cima a questa gerarchia ci sono due classi astratte i cui nomi sono abbastanza ovvi: InputStream e OutputStream. Trattandosi di classi astratte, non si potranno istanziare direttamente oggetti appartenenti a queste classi. Comunque si possono dichiarare delle variabili appartenenti a queste classi (per i programmatori C++, si ricorda che le variabili dichiarate sono in effetti dei riferimenti o puntatori, e quindi dichiarando una variabile non si istanzia automaticamente un oggetto di tale classe), e a queste si potrà assegnare qualsiasi oggetto appartenente ad una classe derivata (l'analogia con il C++ prosegue: un puntatore a una classe base può puntare a un qualsiasi oggetto appartenente a una classe derivata); in questo modo si potrà utilizzare a pieno il polimorfismo, rendendo le proprie classi indipendenti dal particolare stream (e quindi anche dal particolare canale di comunicazione).

Java ovviamente mette a disposizione diverse sottoclassi che specializzano gli stream per i diversi dispositivi e canali di comunicazione, ma vediamo prima i metodi messi a disposizione da queste due classi base.

La classe OutputStream

La classe OutputStream rappresenta una porta verso un canale di comunicazione; tramite questa porta si possono scrivere dati sul canale con il quale la porta è collegata. Si ricorda che si tratta di una classe astratta, che quindi fornisce un'interfaccia coi metodi caratteristici di ogni stream di output. Saranno le sottoclassi a fornire un'implementazione effettiva di tali metodi, che ovviamente dipenderà dal particolare canale di comunicazione.

Descrizione classe

```
public abstract class OutputStream extends Object
```

Trattandosi di una classe astratta, non sono presenti costruttori utilizzabili direttamente.

Metodi

```
public abstract void write(int b) throws IOException
```

Viene accettato un singolo byte, che verrà scritto sul canale di comunicazione con il quale lo stream è collegato. Notare che, nonostante l'argomento sia di tipo intero, verrà scritto solo il byte meno significativo. Ovviamente si tratta di un metodo astratto, in quanto la scrittura dipende fortemente dal particolare dispositivo fisico del canale di comunicazione.

```
public void write(byte b[], int off, int len) throws IOException
```

```
public void write(byte b[]) throws IOException
```

Questi metodi permettono di scrivere un array di byte sul canale di comunicazione. È possibile scrivere l'intero array (secondo metodo), o solo una parte (primo metodo), specificando l'indice del primo elemento (off), e il numero di elementi (len). Il secondo metodo, nell'implementazione di default, richiama semplicemente il primo sull'intero array. A sua volta il primo metodo, nella sua implementazione di default, richiama il numero di volte necessario il metodo write(int b). Il metodo bloccherà il chiamante fino a che tutti i byte dell'array non saranno stati scritti.

```
public void flush() throws IOException
```

Questo metodo effettua il *flush* dei dati bufferizzati nello stream, cioè fa in modo che eventuali dati non ancora scritti effettivamente, vengano scritti nel canale di comunicazione. A volte infatti, per motivi di ottimizzazione e performance, i dati scritti nello stream non vengono scritti immediatamente nel canale di comunicazione, ma vengono tenuti temporaneamente in un buffer. Con questo metodo si fa in modo che i dati presenti nel buffer vengano scritti effettivamente sul canale. Quando si tratta di comunicazioni in rete, la tecnica della "bufferizzazione" è quasi d'obbligo, per ovviare alla lentezza di tali comunicazioni.

```
public void close() throws IOException
```

Con questo metodo si chiude lo stream e quindi il canale di comunicazione. Prima della chiusura tutti i dati eventualmente bufferizzati vengono sottoposti a flush; questo può comportare il dover attendere (e quindi rimanere bloccati) fino al completamente dell'operazione di scrittura.

L'eccezione IOException può essere lanciata per vari motivi che riguardano dei problemi del canale di comunicazione. Il tipo esatto dell'eccezione dipende quindi dal particolare canale. Tipicamente le operazioni sugli stream dovrebbero essere racchiuse nei classici blocchi try–catch–finally, o fare in modo che il metodo che li utilizza dichiari di lanciare una tale eccezione.

La classe InputStream

La classe InputStream è la classe complementare della classe OutputStream, che fornisce funzionalità per l'input, quindi per la lettura di dati da un canale di comunicazione. Quanto si è detto sui metodi astratti è valido anche per questa classe.

Descrizione classe

```
public abstract class InputStream extends Object
```

Metodi

Questa classe fornisce metodi per leggere byte, per determinare il numero di byte disponibili per essere letti senza rimanere bloccati, e per saltare o rileggere dei dati. Come è già stato accennato, leggere da uno stream che non contiene dati bloccherà il thread che ha effettuato

l'operazione di lettura. Se alcuni dati sono già arrivati dal canale di comunicazione, verranno messi temporaneamente in un buffer in attesa di essere effettivamente letti. Quando, a questo punto, un thread cercherà di leggere dallo stream, lo potrà fare immediatamente senza bisogno di attendere e di bloccarsi.

```
public abstract int read() throws IOException
```

Questo metodo legge un singolo byte, aspettando eventualmente che ve ne sia uno disponibile. Ancora una volta, pur trattandosi di un `int`, il valore restituito sarà comunque compreso fra `0` e `255`. Se viene raggiunta la fine dello stream, verrà restituito il valore `-1`. Il concetto di fine dello stream dipende dal particolare canale di comunicazione che si sta utilizzando (ad esempio nel caso di un file rappresenta la fine del file). Si tratta di un metodo astratto perché la lettura di dati da uno stream dipende dal particolare canale di comunicazione con cui lo stream è collegato.

```
public int read(byte b[], int off, int len) throws IOException
```

```
public int read(byte b[]) throws IOException
```

Con questi metodi è possibile leggere una serie di byte e memorizzarli nell'array specificato. È possibile specificare anche il numero di byte da leggere (`len`) e memorizzare nell'array, specificando l'indice iniziale (`off`). L'array dovrà già essere stato allocato. Si tratta ovviamente di un metodo bloccante, se non sono presenti dati da leggere. Il metodo restituisce inoltre il numero di byte letti. Infatti non è detto che venga letto esattamente il numero di byte richiesti: vengono letti i dati che possono essere letti immediatamente senza necessità di attendere, e questi possono essere in numero inferiore a quelli effettivamente richiesti. L'implementazione di default del secondo metodo è quella di richiamare il primo su tutto l'array. A sua volta l'implementazione di default del primo è di richiamare ripetutamente il metodo `read()`.

```
public abstract int available() throws IOException
```

Restituisce il numero di byte che sono disponibili nello stream per essere letti senza attendere.

```
public void close() throws IOException
```

Chiude lo stream e il canale di comunicazione con cui lo stream è collegato. I dati non ancora letti andranno persi.

```
public long skip(long n) throws IOException
```

Vengono saltati e scartati `n` byte presenti nello stream. Questo è utile se si vogliono ignorare dei byte, ed è più efficiente che leggere i byte e ignorarli. Il metodo restituisce il numero di byte effettivamente saltati; questo perché, per vari motivi, può non essere possibile saltare esattamente il numero di byte richiesto.

```
public synchronized void mark(int readlimit)

public synchronized void reset() throws IOException
```

Marca la posizione corrente all'interno dello stream. Una successiva chiamata al metodo reset riposizionerà lo stream alla precedente posizione marcata. Dopo la chiamata del metodo reset letture successive leggeranno dall'ultima posizione marcata. Con il parametro readlimit si specifica il numero massimo di byte che saranno letti, prima che la posizione marcata non sia più valida. Se sono letti più di readlimit byte, una successiva chiamata di reset potrebbe fallire.

Questi due metodi possono risultare utili nelle situazioni in cui vi sia bisogno di leggere alcuni byte prima di capire quale tipo di dati è presente nello stream. Se si deve decodificare tali dati, e si hanno vari tipi di decodificatori, quando un decodificatore si rende conto che non sono dati che lo riguardano, può "rimettere a posto" i dati già letti, rendendoli disponibili ad un altro decodificatore.

```
public boolean markSupported()
```

Permette di capire se lo stream corrente gestisce il corretto funzionamento delle operazioni di mark e reset.

Anche nel caso di InputStream l'eccezione IOException può essere lanciata in varie occasioni.

Gli stream predefiniti

Il pacchetto java.lang, incluso automaticamente da tutti i programmi Java, definisce alcuni stream predefiniti, contenuti nella classe System. Si tratta di tre variabili statiche e pubbliche (quindi utilizzabili in qualunque parte del programma, senza aver istanziato un oggetto System) denominate in, out e err. Queste si riferiscono rispettivamente allo standard input, che per default è la tastiera, al flusso standard di output, che per default è lo schermo, e al flusso standard di errori che, anche in questo caso, per default è lo schermo. Tali stream possono essere reindirizzati quando si lancia il programma da linea di comando utilizzando > e < (per questo si rimanda al sistema operativo che si utilizza).

Esempi

Si prenderanno ora in considerazione due semplici esempi che utilizzano tali stream predefiniti:

```
import java.io.*;

public class OutSample {
 public static void main (String args[]) throws IOException {
    for (int i = 0; i < args.length; ++ i) {
       synchronized(System.out) {
```

```
            for (int j = 0; j < args[i].length (); ++j)
               System.out.write ((byte) args[i].charAt (j));
            System.out.write ('\n');   // scrive un invio
            System.out.flush ();    // scarica il buffer
         }
      }
   }
}
```

Questo semplice programma scrive sullo schermo i vari argomenti passati sulla linea di comando. Viene utilizzato il metodo write per scrivere un byte alla volta, ed il metodo flush per essere sicuri che ogni stringa passata venga stampata subito. Si può notare che il metodo main dichiara di poter lanciare un'eccezione IOException; in effetti i metodi write e flush possono lanciare tali eccezioni.

Un po' meno chiaro può risultare l'utilizzo di un blocco sincronizzato. In questo caso non sarebbe necessario in quanto non si usano più thread. Nel caso di un programma con più thread è bene sincronizzare l'accesso alla variabile System.out in modo che, quando un thread ha iniziato a scrivere su tale stream, non venga interrotto prima che abbia finito; nello stream altrimenti sarebbero presenti informazioni rovinate e mischiate.

Un'alternativa potrebbe essere quella di scrivere una stringa alla volta, invece dei suoi singoli byte. Per far questo si deve convertire la stringa in un array di byte, e poi richiamare il metodo write appropriato. Vale a dire che al posto del ciclo for più interno si sarebbe potuto scrivere

```
byte buffer[] = new byte[args[i].length()];
msg.getBytes (0, args[i].length (), buffer, 0);
System.out.write (buffer);
```

In effetti la variabile out appartiene alla classe PrintStream, che specializza un OutputStream per scrivere dati in formato testo (e quindi adatto per scrivere dati sullo schermo). Questa classe mette a disposizione due metodi molto utilizzati per stampare facilmente stringhe e altri dati come testo: si tratta dei metodi print e println (quest'ultimo si distingue dal precedente perché aggiunge un newline dopo la stampa). In effetti il programma precedente può essere riscritto in modo molto più semplice:

```
public class OutSamplePrint {
 public static void main (String args[]) throws IOException {
    for (int i = 0; i < args.length; ++ i)
       System.out.println(i + ": " + args[i]);
 }
}
```

Come si vede non c'è bisogno di tradurre la stringa in un array di byte, in quanto i metodi suddetti gestiscono direttamente le stringhe, e sono anche in grado di tradurre dati di altro tipo (ad esempio i è un intero) in stringa (infatti questo programma stampa le stringhe immesse da riga di comando insieme alla numerazione). Non c'è nemmeno bisogno di sincronizzarsi su System.out in quanto questi metodi sono già dichiarati come sincronizzati.

Ecco adesso un semplice programma che legge i caratteri immessi da tastiera e li ristampa sullo schermo. In questo caso si utilizzerà anche la variabile System.in.

```
import java.io.*;

public class InSample {
 static public void main (String args[]) throws IOException {
    int c;
    while ((c = System.in.read ()) >= 0)
      System.out.print((char)c);
 }
}
```

Come si vede, dei caratteri da tastiera vengono letti e poi stampati sullo schermo (la conversione esplicita a char è necessaria, altrimenti verrebbe stampato un numero). Da notare che il metodo read memorizza in un buffer i caratteri digitati e li restituisce solo quando l'utente preme Invio. Chiaramente questo metodo non è molto indicato per un input interattivo da console.

Ancora una volta, può essere più efficiente utilizzare dei buffer per ottimizzare le prestazioni. Per far questo basta cambiare il corpo del main con il seguente:

```
    byte buffer[] = new byte[8];
    int numberRead;
    while ((numberRead = System.in.read (buffer)) > -1)
      System.out.write (buffer, 0, numberRead);
```

Questo semplice programma può essere utilizzato anche per visualizzare il contenuto di un file di testo: basterà semplicemente ridirezionare lo standard input (la tastiera) su un file. Ad esempio con il seguente comando

```
java InSample < InSample.java
```

si visualizzerà sullo schermo il contenuto del sorgente del programma stesso.

Si vedranno adesso alcune classi che specializzano gli stream di input e output. Come si è visto le classi base offrono solo metodi per scrivere singoli byte e al massimo array di byte. Spesso invece si ha la necessità di leggere e/o scrivere stringhe o numeri, quindi si avrebbe bisogno di stream che forniscano metodi per effettuare direttamente queste operazioni, senza dover manualmente effettuare conversioni.

Stream filtro

Si vedrà ora il concetto di stream filtro (*filter stream*), cioè uno stream che fornisce metodi ad alto livello per inviare o ricevere i dati primitivi di Java su un qualsiasi stream di comunicazione.

Uno stream filtro agisce appunto come un filtro per uno stream già esistente, aggiungendo

funzionalità ad alto livello. Tra l'altro questo permette di tralasciare tutti i dettagli su come i dati vengono memorizzati in uno stream (ad esempio se un intero viene memorizzato partendo dal byte più alto o da quello più basso).

È necessario quindi fornire uno stream già esistente ad uno stream filtro. Ad uno stream filtro di input passeremo uno stream di input qualsiasi (in pratica un oggetto di classe InputStream), così come ad uno stream filtro di output passeremo uno stream di output qualsiasi (un oggetto di classe OutputStream). Anche gli stream filtro sono sottoclassi delle classi base InputStream e OutputStream, quindi è possibile costruire una serie di stream filtro in cascata, a seconda delle varie esigenze. Ci sarà modo di vedere alcuni esempi successivamente.

Le classi FilterOutputStream e FilterInputStream

Queste sono le classi base per ogni stream filtro, e non sono altro che template (modelli) per tutti gli altri stream filtro. L'unica funzionalità aggiuntiva che mettono a disposizione è il fatto di poter passare ai loro costruttori un qualsiasi stream con il quale collegarsi (quindi rispettivamente un OutputStream e un InputStream). Gli unici metodi che mettono a disposizione sono gli stessi che sono presenti nella classe base. La semplice azione di default sarà quella di richiamare il metodo corrispondente dello stream con il quale sono collegati. La loro utilità si riduce quindi a fornire un'interfaccia uniforme per tutti gli altri stream filtro, e ovviamente a fornire una classe base comune.

Le classi DataOutputStream e DataInputStream

Queste classi sono fra le più utilizzate in quanto mettono a disposizione proprio le funzionalità che cercavamo negli stream filtro: forniscono metodi rispettivamente per scrivere e leggere tutti i tipi primitivi del linguaggio (stringhe, interi, ecc.).

Ovviamente questi due stream, come spesso accade negli stream filtro, devono lavorare in coppia affinché le comunicazioni di informazioni abbiano successo: se da una parte si utilizza un DataOutputStream per spedire una stringa, dall'altra parte ci dovrà essere in ascolto un DataInputStream, che sia in grado di decodificare la stringa ricevuta dal canale di comunicazione. Infatti i metodi di questi stream filtro si occupano, rispettivamente, di codificare e decodificare i vari tipi di dato. Non sarà necessario preoccuparsi dell'ordine dei byte di un intero o della codifica di una stringa, ovviamente purché tali stream siano utilizzati in coppia.

Nonostante non ci si debba preoccupare della codifica dei dati spediti, può comunque essere interessante sapere che questi stream utilizzano il *network byte order* per la memorizzazione dei dati: il byte più significativo viene scritto per primo (e dall'altra parte letto per primo). In questo modo le applicazioni scritte in Java, potranno comunicare dati con questi stream, con qualsiasi altro programma scritto in un altro linguaggio che usi la convenzione del network byte order.

Descrizione classe DataOutputStream

```
public class DataOutputStream
extends FilterOutputStream implements DataOutput
```

L'unica cosa da notare è l'interfaccia DataOutput. Questa interfaccia, insieme alla simmetrica DataInput, descrive gli stream che scrivono e leggono (rispettivamente) dati in un formato indipendente dalla macchina.

Costruttore

```
public DataOutputStream(OutputStream out)
```

Come già accennato quando si è parlato in generale degli stream filtro, viene passato al costruttore lo stream sul quale si agisce da filtro. Vale la pena di ricordare che si passa un OutputStream, quindi, trattandosi della classe base di tutti gli stream di output, si può passare un qualsiasi stream di output.

Metodi

I metodi seguenti fanno parte della suddetta interfaccia DataOutput. A questi vanno aggiunti i metodi derivati dalla classe base, che non verranno descritti (il loro nome è di per sé molto esplicativo).

```
public final void writeBoolean(boolean v) throws IOException

public final void writeByte(int v) throws IOException

public final void writeShort(int v) throws IOException

public final void writeChar(int v) throws IOException

public final void writeInt(int v) throws IOException

public final void writeLong(long v) throws IOException

public final void writeFloat(float v) throws IOException

public final void writeDouble(double v) throws IOException
```

Come si può notare, esiste un metodo per ogni tipo di dato primitivo di Java. Il loro significato dovrebbe essere abbastanza immediato. I prossimi metodi invece meritano una spiegazione un po' più dettagliata.

```
public final void writeBytes(String s) throws IOException
```

Questo metodo scrive una stringa sullo stream collegato come una sequenza di byte. Viene scritto solo il byte più basso di ogni carattere, quindi può essere utilizzato per trasmettere dei dati in formato ASCII a un dispositivo come un terminale carattere, o un client scritto in C. La lunghezza della stringa non viene scritta nello stream.

```
public final void writeChars(String s) throws IOException
```

La stringa passata viene scritta come sequenza di caratteri. Ogni carattere viene scritto come una coppia di byte. Non viene scritta la lunghezza della stringa, né il terminatore.

```
public final void writeUTF(String str) throws IOException
```

La stringa viene scritta nel formato Unicode UTF-8 in modo indipendente dalla macchina. La stringa viene scritta con una codifica in modo tale che ogni carattere viene scritto come un solo byte, due byte, o tre byte. I caratteri ASCII saranno scritti come singoli byte, mentre i caratteri più rari vengono scritti con tre byte. Inoltre i primi due byte scritti rappresentano il numero di byte effettivamente scritti. Quindi la lunghezza della stringa viene scritta nello stream.

Tutti questi metodi possono lanciare l'eccezione IOException; questo perché viene usato il metodo write dello stream con il quale lo stream filtro è collegato, che può lanciare un'eccezione di questo tipo.

Descrizione classe DataInputStream

```
public class DataInputStream extends FilterInputStream implements DataInput
```

Valgono le stesse considerazioni fatte riguardo alla classe DataOutputStream.

Costruttore

```
public DataInputStream(InputStream in)
```

Anche in questo caso si passa un InputStream al costruttore.

Metodi

Sono presenti i metodi simmetrici rispetto a quelli di DataOutputStream.

```
public final boolean readBoolean() throws IOException

public final byte readByte() throws IOException

public final int readUnsignedByte() throws IOException
```

```
public final short readShort() throws IOException

public final int readUnsignedShort() throws IOException

public final char readChar() throws IOException

public final int readInt() throws IOException

public final long readLong() throws IOException

public final float readFloat() throws IOException

public final double readDouble() throws IOException

public final String readUTF() throws IOException
```

Metodi che meritano particolare attenzione sono i seguenti:

```
public final void readFully(byte b[], int off, int len) throws IOException

public final void readFully(byte b[]) throws IOException
```

Questi metodi leggono un array di byte o un sottoinsieme, ma bloccano il thread corrente finché tutto l'array (o la parte di array richiesta) non viene letto. Viene lanciata un'eccezione EOFException se viene raggiunto prima l'EOF. A tal proposito si può notare che non può essere restituito il numero -1 per segnalare l'EOF, in quanto se si sta leggendo un intero, -1 è un carattere intero accettabile. Per questo motivo si ricorre all'eccezione suddetta.

```
public final static String readUTF(DataInput in) throws IOException
```

Si tratta di un metodo statico che permette di leggere una stringa con codifica UTF, dall'oggetto in, quindi un oggetto (in particolare uno stream) che implementi l'interfaccia DataInput.

Anche in questo caso può essere lanciata un'eccezione IOException. In particolare la suddetta eccezione EOFException deriva da IOException. Un'altra eccezione (sempre derivata da IOException) che può essere lanciata è UTFDataFormatException, nel caso in cui i dati ricevuti dal metodo readUTF non siano nel formato UTF.

Classi BufferedOutputStream e BufferedInputStream

Talvolta nelle comunicazioni è molto più efficiente bufferizzare i dati spediti. Questo è senz'altro vero per le comunicazioni in rete, ma può essere vero anche quando si deve scrivere o leggere da un file (anche se a questo pensaautomaticamente il sistema operativo sottostante).

Richiamando i metodi di scrittura della classe BufferedOutputStream, i dati verranno memorizzati temporaneamente in un buffer interno (quindi in memoria), finché non viene chiamato

il metodo flush, che provvederà a scrivere effettivamente i dati nello stream con cui il filtro è collegato, oppure finché il buffer non diventa pieno.

Quindi è molto più efficiente scrivere dei dati su un canale di comunicazione utilizzando un DataOutputStream collegato a un BufferedOutputStream. Ad esempio se si utilizza un DataOutputStream collegato direttamente a un canale di comunicazione di rete, e si scrive un intero con il metodo writeInt, è molto probabile che il primo byte dell'intero scritto sarà spedito subito in rete in un pacchetto. Un altro pacchetto — o forse più pacchetti — sarà utilizzato per i rimanenti byte. Sarebbe molto più efficiente scrivere tutti i byte dell'intero in un solo pacchetto e spedire quel singolo pacchetto. Se si costruisce un DataOutputStream su un BufferedOutputStream si otterranno migliori prestazioni. Questo è, tra l'altro, un esempio di due stream filtro collegati in cascata.

Se da una parte della comunicazione c'è un BufferedOutputStream che scrive dei dati, non è detto che dall'altra parte ci debba essere un BufferedInputStream in ascolto: in effetti questi stream filtro non codificano l'output ma semplicemente effettuano una bufferizzazione.

Comunque converrebbe utilizzare anche in lettura uno stream bufferizzato, cioè un BufferedInputStream. Utilizzare un buffer in lettura significa leggere i dati dal buffer interno, e solo quando nuovi dati, non presenti nel buffer, dovranno essere letti, si accederà al canale di comunicazione.

Descrizione classe BufferedOutputStream

```
public class BufferedOutputStream
extends FilterOutputStream
```

Costruttori

```
public BufferedOutputStream(OutputStream out)

public BufferedOutputStream(OutputStream out, int size)
```

Nel primo caso viene creato uno stream bufferizzato collegato allo stream di output out; la dimensione del buffer sarà quella di default, cioè 512 byte. Nel secondo caso è possibile specificare la dimensione del buffer. I dati scritti in questo stream saranno scritti sullo stream collegato out solo quando il buffer è pieno, o verrà richiamato il metodo flush.

Metodi

Come si è visto la classe deriva direttamente da FilterOutputStream e l'unico metodo che aggiunge a quelli della classe base (cioè quelli di OutputStream) è il metodo flush.

```
public synchronized void flush() throws IOException
```

Questo metodo fa sì che i dati contenuti nel buffer siano effettivamente scritti sullo stream collegato.

Descrizione classe BufferedInputStream

public class BufferedInputStream extends FilterInputStream

Costruttori

public BufferedInputStream(InputStream in)
public BufferedInputStream(InputStream in, int size)

Viene creato uno stream di input bufferizzato collegato allo stream si input in; è possibile anche in questo caso specificare la dimensione del buffer, o accettare la dimensione di default (512 byte).

Metodi

Non vengono aggiunti metodi, e quindi si hanno a disposizione solo quelli di FilterInputStream (cioè solo quelli di InputStream); l'unica differenza è che tali metodi faranno uso del buffer interno.

Stream per l'accesso alla memoria

Java supporta l'input e l'output di array di byte tramite l'uso delle classi ByteArrayOutputStream e ByteArrayInputStream. Questi stream non sono collegati con un canale di comunicazione vero e proprio: queste classi infatti utilizzano dei buffer di memoria come sorgente e come destinazione dei flussi di input e output. In questo caso, tali stream non devono essere utilizzati necessariamente insieme.

Vi sono poi le classi PipedInputStream e PipedOutputStream, che permettono la comunicazione, tramite appunto memoria, di due thread di un'applicazione. Un thread leggerà da un lato della *pipe* e riceverà tutto quello che sarà scritto dall'altro lato da altri thread. Questi stream saranno creati sempre in coppia; un lato della *pipe* viene creato senza essere connesso, mentre l'altro sarà creato connettendolo con il primo. Quindi basta collegare uno stream con l'altro, e non entrambi.

Descrizione classe ByteArrayInputStream

public class ByteArrayInputStream extends InputStream

Questa classe crea uno stream di input da un buffer di memoria, in particolare da un array di byte.

Costruttori

public ByteArrayInputStream(byte buf[])
public ByteArrayInputStream(byte buf[], int offset, int length)

Con questi costruttori si può specificare l'array (o una parte dell'array nel secondo caso) con il quale lo stream sarà collegato.

Metodi

Tale classe non mette a disposizione nuovi metodi, semplicemente ridefinisce i metodi della classe base InputStream. In particolare chiamando il metodo read, in una delle sue forme, verranno letti i byte dell'array collegato, fino a che non sarà raggiunta la fine dell'array, e in tal caso sarà restituito EOF. Inoltre la semantica del metodo reset è leggermente differente: resettare un ByteArrayInputStream vuol dire ripartire sempre dall'inizio dell'array, in quanto il metodo mark marca sempre la posizione iniziale.

Descrizione classe ByteArrayOutputStream

```
public class ByteArrayOutputStream extends OutputStream
```

Questa classe crea uno stream di output su un array di byte, ed è un po' più potente della sua classe complementare: permette all'array di byte con il quale è collegata di crescere dinamicamente, man mano che vengono aggiunti nuovi dati. Il buffer di memoria può essere estratto e utilizzato.

Costruttori

```
public ByteArrayOutputStream()
public ByteArrayOutputStream(int size)
```

È possibile specificare la dimensione iniziale del buffer o accettare quella di default (32 byte).

Metodi

Anche in questo caso il metodo reset() acquista un significato particolare: svuota il buffer, e successive scritture memorizzeranno i dati a partire dall'inizio. Vi sono poi alcuni metodi aggiunti:

```
public int size()
```

Viene restituito il numero di byte che sono stati scritti nel buffer (da non confondersi con la dimensione del buffer, che può essere anche maggiore).

```
public synchronized byte[] toByteArray()
```

Viene restituito un array di byte rappresentante una copia dei dati scritti nel buffer. Il buffer interno non sarà resettato da questo metodo, quindi successive scritture nello stream continueranno a estendere il buffer.

```
public String toString()
```

Viene restituita una stringa rappresentante una copia del buffer dello stream. Anche in questo caso il buffer dello stream non viene resettato. Ogni carattere della stringa corrisponderà al relativo byte del buffer.

```
public synchronized void writeTo(OutputStream out) throws IOException
```

I contenuti del buffer dello stream vengono scritti nello stream di output out. Anche in questo caso il buffer dello stream non viene resettato. Se si verificano degli errori durante la scrittura nello stream di output out verrà sollevata un'eccezione IOException.

Ecco adesso un piccolo esempio che fa uso dei suddetti stream. Le stringhe che vengono passate sulla riga di comando vengono tutte inserite in ByteArrayOutputStream. Il buffer dello stream viene estratto e su tale array di byte viene costruito un ByteArrayInputStream. Da questo stream verranno poi estratti e stampati sullo schermo tutti i byte.

```
Import java.io.* ;

public class ByteArrayIOSample {
 public static void main(String args[]) throws IOException {
    ByteArrayOutputStream oStream = new ByteArrayOutputStream();



for (int i = 0; i < args.length; i++)
       for (int j = 0; j < args[i].length(); j++)
          oStream.write(args[i].charAt(j));

 // per la concatenazione a stringa viene
 // chiamato toString()
    System.out.println("oStream: " + oStream);

    System.out.println("size: " + oStream.size());

    ByteArrayInputStream iStream = new ByteArrayInputStream(oStream.toByteArray());

    System.out.println("Byte disponibili: " + iStream.available());
    int c ;
    while((c = iStream.read()) != -1)
       System.out.write(c);
 }
}
```

Descrizione classe PipedOutputStream

```
public class PipedOutputStream extends OutputStream
```

Costruttori

```
public PipedOutputStream() throws IOException

public PipedOutputStream(PipedInputStream snk) throws IOException
```

Si può creare un PipedOutputStream e poi connetterlo con un PipedInputStream, oppure lo si può passare direttamente al costruttore, se già esiste.

Metodi

Sono disponibili i metodi standard della classe OutputStream ed in più è presente il metodo per connettere lo stream con un PipedInputStream:

```
public void connect(PipedInputStream src) throws IOException
```

Se si scrive su un PipedOutputStream, e il thread che è in ascolto sul corrispondente PipedInputStream termina, si otterrà un'IOException.

Questi stream sono implementati con un buffer di memoria, e se il buffer diventa pieno, una successiva chiamata al metodo write bloccherà il thread che scrive sullo stream, finché il thread in ascolto sullo stream di input corrispondente non legge qualche byte. Se questo thread termina, l'eccezione suddetta evita che l'altro processo rimanga bloccato indefinitamente.

Descrizione classe PipedInputStream

Per questa classe, che è la relativa classe di lettura della precedente, valgono le stesse annotazioni fatte per la classe PipedOutputStream.

```
public class PipedInputStream extends InputStream
```

Costruttori

```
public PipedInputStream() throws IOException

public PipedInputStream(PipedOutputStream src) throws IOException
```

Metodi

```
public void connect(PipedOutputStream src) throws IOException
```

Anche in questo caso si deve evitare che un thread bloccato a leggere da un PipedInputStream, rimanga bloccato indefinitamente; se viene chiamato il metodo read su uno stream vuoto, verrà sollevata un'eccezione IOException. Segue un semplice esempio che illustra l'utilizzo di questi due stream per la comunicazione fra due thread (il thread principale e un thread parallelo):

```
import java.io.* ;

public class PipedIOSample extends Thread {
 protected DataInputStream iStream ;

 public PipedIOSample(InputStream i) {
       this.iStream = new DataInputStream(i);
 }

 public void run() {
    try {
       String str;
       while (true) {
          str = iStream.readUTF();
          System.out.println("Letta: " + str);
       }
    }


catch (IOException e) {
       e.printStackTrace();
    }
 }

 public static void main(String args[]) throws IOException {
    PipedOutputStream o = new PipedOutputStream();
    PipedInputStream iStream = new PipedInputStream(o);
    DataOutputStream oStream = new DataOutputStream(o);

    (new PipedIOSample(iStream)).start();

    for (int i = 0; i < args.length; i++) {
       System.out.println("Scrivo: " + args[i]);
       oStream.writeUTF(args[i]);
    }

    oStream.close();
 }
}
```

Come si può notare viene creato un PipedOutputStream senza specificare nessuno stream da collegare; poi viene creato un PipedInputStream collegato al precedente stream. A questo punto i due stream sono connessi e tutto quello che viene scritto sullo stream di output potrà essere letto da quello di input. L'idea è quella di scrivere le stringhe da passare sulla riga di comando sullo stream di output; tali stringhe saranno lette da un altro thread sullo stream (sempre di tipo piped) di input. In particolare invece di utilizzare più volte il me-

todo write per scrivere un singolo byte alla volta, utilizziamo un DataOutputStream collegato al PipedOutputStream, e scriviamo una stringa alla volta con il metodo readUTF. Allo stesso modo il thread che legge le stringhe lo farà tramite un DataInputStream collegato allo stream passato al costruttore. Vale la pena di notare che al costruttore viene passato un InputStream generico. Il thread che legge le stringhe lo fa in un ciclo infinito, che terminerà non appena verrà chiuso lo stream di output (ultima istruzione del main), a causa dell'eccezione IOException.

I file

Trattando l'input/output non si può certo tralasciare l'argomento file. Java fornisce l'accesso ai file tramite gli stream. In questo modo, per la genericità degli stream, un'applicazione progettata per leggere e/o scrivere utilizzando le classi InputStream e OutputStream, può utilizzare i file in modo trasparente.

Java inoltre mette a disposizione altre classi per facilitare l'accesso ai file e alle directory

Descrizione classe File

```
public class File extends Object implements Serializable
```

La classe File fornisce l'accesso a file e directory in modo indipendente dal sistema operativo. Tale classe mette a disposizione una serie di metodi per ottenere informazioni su un certo file e per modificarne gli attributi; tramite questi metodi, ad esempio, è possibile sapere se un certo file è presente in una certa directory, se è a sola lettura, e via dicendo.

Si è parlato di indipendenza dal sistema operativo: effettivamente ogni sistema operativo utilizza convenzioni diverse per separare le varie directory in un *path*. Quando si specifica un file e/o un path, si suppone che vengano utilizzate le convenzioni del sistema operativo sottostante. I vari metodi che sono messi a disposizione dalla classe permettono di ottenere le informazioni relative a tali convenzioni. Inoltre è possibile cancellare file, rinominarli, e ottenere la lista dei file contenuti in una certa directory.

Costruttori

```
public File(String path)
public File(String path, String name)
public File(File dir, String name)
```

È possibile creare un oggetto File specificando un path e anche un nome di file. Il path deve essere specificato utilizzando le convenzioni del sistema operativo sottostante. Se viene specificato anche il nome del file, oltre al percorso, verrà creato un path concatenando il percorso specificato ed il file con il separatore utilizzato dal sistema operativo. Con la terza versione è possibile specificare la directory del file tramite un altro oggetto File.

Metodi

Come già detto, tale classe è utile per avere un meccanismo in grado di utilizzare file e directory in modo indipendente dalle convenzioni del sistema operativo e per eseguire le classiche operazioni sui file e sulle directory. Tali metodi non lanciano un'IOException, in caso di fallimento, ma restituiscono un valore booleano.

```
public String getName()
```

```
public String getPath()
```

Restituiscono rispettivamente il nome e il percorso dell'oggetto File.

```
public String getAbsolutePath()
```

```
public String getCanonicalPath() throws IOException
```

Restituiscono rispettivamente il percorso assoluto dell'oggetto File, e il percorso canonico. Quest'ultimo è un percorso completo in cui eventuali riferimenti relativi e simbolici sono già stati valutati e risolti. Quest'ultimo concetto ovviamente dipende fortemente dal sistema operativo.

```
public String getParent()
```

Restituisce il nome della *parent directory* dell'oggetto File. Per un file si tratta del nome della directory.

```
public boolean exists()
```

```
public boolean canWrite()
```

```
public boolean canRead()
```

Questi metodi permettono di capire se un file con il nome specificato esiste, se è scrivibile e se è leggibile.

```
public boolean isFile()
```

```
public boolean isDirectory()
```

```
public boolean isAbsolute()
```

Permettono di capire se l'oggetto File rappresenta un file, una directory o un percorso assoluto.

```
public long lastModified()
```

```
public long length()
```

Permettono di conoscere la data dell'ultima modifica del file, e la sua lunghezza in byte.

```
public boolean renameTo(File dest)

public boolean delete()
```

Permettono di rinominare e di cancellare un file.

```
public boolean mkdir()
public boolean mkdirs()
```

Permette di creare una directory che corrisponde all'oggetto File. La seconda versione crea se necessario tutte le directory del percorso dell'oggetto File.

```
public String[] list()
public String[] list(FilenameFilter filter)
```

Restituiscono l'elenco di tutti i file della directory corrispondente all'oggetto File. Nella seconda versione è possibile specificare un filtro.

Descrizione classe RandomAccess

```
public class RandomAccessFile extends Object implements DataOutput, DataInput
```

Anche in Java è possibile accedere ai file in modo random, cioè non in modo sequenziale. Tramite questa classe infatti è possibile accedere a una particolare posizione in un file, ed è inoltre possibile accedere al file contemporaneamente in lettura e scrittura (cosa che non è possibile con l'accesso sequenziale messo a disposizione dagli stream sui file, che saranno illustrati successivamente). È comunque possibile specificare in che modo accedere a un file (solo lettura, o lettura e scrittura).

La classe RandomAccessFile, implementando le interfacce DataInput e DataOutput, rende possibile scrivere in un file tutti gli oggetti e i tipi di dati primitivi. La classe inoltre fornisce i metodi per gestire la posizione corrente all'interno del file.

Se si scrive su un file esistente ad una particolare posizione si sovrascriveranno i dati a quella posizione.

Costruttori

```
public RandomAccessFile(String file, String mode) throws IOException

public RandomAccessFile(File file, String mode) throws IOException
```

Si può specificare il file da aprire sia tramite una stringa, sia tramite un oggetto File. Si deve inoltre specificare il modo di apertura del file nella stringa mode. Con la stringa "r" si apre il file in sola lettura, e con "rw" sia in lettura che in scrittura.

Metodi

```
public int read() throws IOException
```

Legge un byte. Blocca il processo chiamante se non è disponibile dell'input.

```
public int read(byte b[], int off, int len) throws IOException
```

```
public int read(byte b[]) throws IOException
```

Riempie un array o una parte dell'array specificato con i dati letti dal file. Viene restituito il numero di byte effettivamente letti.

```
public final void readFully(byte b[]) throws IOException
```

```
public final void readFully(byte b[], int off, int len) throws IOException
```

Questi metodi cercano di riempire un array (o una sua parte) con i dati letti dal file. Se viene raggiunta la fine del file prima di aver terminato, viene lanciata un'eccezione EOFException.

```
public final FileDescriptor getFD() throws IOException
```

Viene restituito un descrittore di file utilizzato dal sistema operativo per gestire il file. Si tratta di un descrittore a basso livello rappresentato dalla classe FileDescriptor. Difficilmente ci sarà la necessità di gestire direttamente tale informazione.

```
public int skipBytes(int n) throws IOException
```

Questo metodo salta n byte, bloccandosi finché non sono stati saltati. Se prima di questo si incontra la fine del file, viene sollevata un'eccezione EOFException.

```
public void write(int b) throws IOException
public void write(byte b[]) throws IOException
public void write(byte b[], int off, int len) throws IOException
```

Questi metodi permettono di scrivere rispettivamente in un file un singolo byte (nonostante l'argomento sia di tipo intero, solo il byte meno significativo viene effettivamente scritto nel file), un intero array, o una parte.

```
public native long getFilePointer() throws IOException
```

Restituisce la posizione corrente all'interno del file, cioè la posizione in cui si sta leggendo o scrivendo.

```
public void seek(long pos) throws IOException
```

Sposta il puntatore all'interno del file alla posizione assoluta specificata in pos.

```
public long length() throws IOException
```

Restituisce la lunghezza del file.

```
public void close() throws IOException
```

Chiude il file (scrivendo sul disco eventuali dati bufferizzati).

Nella classe sono poi presenti diversi metodi per leggere e scrivere particolari tipi di dati (ad esempio readBoolean, writeBoolean, readInt, writeInt, ecc.), come quelli già visti nelle classi DataInputStream e DataOutputStream, del resto, come abbiamo visto, RandomAccessFile implementa le interfacce DataInput e DataOutput. Per una lista completa si faccia riferimento alla guida in linea.

Le classi FileOutputStream e FileInputStream

Tramite queste classi è possibile accedere, rispettivamente in scrittura ed in lettura, sequenzialmente ai file, con il meccanismo degli stream.

Descrizione classe FileOutputStream

```
public class FileOutputStream extends OutputStream
```

Costruttori

```
public FileOutputStream(String name) throws IOException
public FileOutputStream(String name, boolean append) throws IOException
```

Si può aprire un file in scrittura specificandone il nome tramite una stringa. Se esiste già un file con lo stesso nome, verrà sovrascritto. È possibile (con il secondo costruttore) specificare se il file deve essere aperto in *append mode*.

```
public FileOutputStream(File file) throws IOException
```

Si può specificare il file da aprire tramite un oggetto File già esistente. Anche in questo caso, se il file esiste già, viene sovrascritto.

```
public FileOutputStream(FileDescriptor fdObj)
```

Si può infine specificare il file con cui collegare lo stream tramite un FileDescriptor. In questo modo si apre uno stream su un file già aperto, ad esempio uno aperto per accesso random. Ovviamente utilizzando questo costruttore non si crea (e quindi non si sovrascrive) un file, che anzi, come già detto, deve essere già aperto.

Metodi

```
public final FileDescriptor getFD() throws IOException
```

In questo modo è possibile ottenere il FileDescriptor relativo al file collegato allo stream.

```
public native void close() throws IOException
```

Si dovrebbe chiamare sempre questo metodo quando non si deve più scrivere sul file. Questo metodo sarà comunque chiamato automaticamente quando lo stream sarà sottoposto al *garbage collecting*.

Descrizione classe FileInputStream

```
public class FileInputStream extends InputStream
```

Costruttori

```
public FileInputStream(String name) throws FileNotFoundException
public FileInputStream(File file) throws FileNotFoundException
public FileInputStream(FileDescriptor fdObj)
```

Uno stream può essere aperto specificando il file da aprire negli stessi modi visti nella classe FileOutputStream. Nel terzo caso il file è già aperto, ma nei primi due no: in tal caso il file deve esistere, altrimenti verrà lanciata un'eccezione FileNotFoundException.

Metodi

```
public final FileDescriptor getFD() throws IOException
```

In questo modo è possibile ottenere il FileDescriptor relativo al file collegato allo stream.

Ecco ora un piccolo esempio di utilizzo di questi due stream per effettuare la copia di due file. Il nome dei due file (sorgente e destinazione) dovrà essere specificato sulla linea di comando.

```
import java.io.*;

public class CopyFile {
 static public void main (String args[]) throws IOException {
```

```
    if(args.length != 2){
      String Msg;
      Msg = "Sintassi: CopyFile <sorgente> <destinazione>"
      throw(new IOException(Msg));
    }

    FileInputStream in = new FileInputStream(args[0]);
    FileOutputStream out = new FileOutputStream(args[1]);

    byte buffer[] = new byte[256];
    int n;
    while((n = in.read (buffer)) > -1)
      out.write(buffer, 0, n);

    out.close();
    in.close();
  }
}
```

Classi Reader e Writer

Dalla versione 1.1 del JDK, sono stati introdotti gli stream che gestiscono i caratteri (*character stream*). Tutti gli stream esaminati fino ad adesso gestiscono solo byte; i character stream sono come i byte stream, ma gestiscono caratteri Unicode a 16 bit, invece che byte (8 bit). Le classi base della gerarchia di questi stream sono Reader e Writer; tali classi supportano le stesse operazioni che erano presenti in InputStream e OutputStream, tranne che per il fatto che, laddove i byte stream operano su byte e su array di byte, i character stream operano su caratteri, array di caratteri, o stringhe.

Il vantaggio degli stream di caratteri è che rendono i programmi indipendenti dalla particolare codifica dei caratteri del sistema su cui vengono eseguite le applicazioni (a tal proposito si veda anche il capitolo sull'internazionalizzazione).

Java infatti per memorizzare le stringhe utilizza l'Unicode; l'Unicode è una codifica con la quale è possibile rappresentare la maggior parte dei caratteri delle varie lingue. I character stream quindi rendono trasparente la complessità di utilizzare le varie codifiche, mettendo a disposizione delle classi che automaticamente provvedono a eseguire la conversione fra gli stream di byte e gli stream di caratteri. La classe InputStreamReader, ad esempio, implementa un input stream di caratteri che legge i byte da un input stream di byte e li converte in caratteri. Allo stesso modo un OutputStreamWriter implementa un output stream di caratteri che converte i caratteri in byte e li scrive in un output stream di byte. Per creare un InputStreamReader basterà quindi eseguire la seguente operazione:

```
InputStreamReader in = new InputStreamReader(System.in);
```

Inoltre gli stream di caratteri sono più efficienti dei corrispettivi stream di byte, in quanto,

mentre questi ultimi eseguono spesso operazioni di lettura e scrittura un byte alla volta, i primi tendono a utilizzare di più la bufferizzazione.

A tal proposito esistono anche le classi BufferedReader e BufferedWriter, che corrispondo a BufferedInputStream e BufferedOutputStream; si può quindi scrivere

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
```

Le classi PrintStream e PrintWriter

Della classe PrintStream si è già parlato all'inizio del capitolo quando si sono introdotti gli stream predefiniti in, out ed err. Tale classe, nel JDK 1.1, è stata modificata in modo da utilizzare la codifica dei caratteri della piattaforma sottostante. Quindi in realtà ogni PrintStream incorpora un OutputStreamWriter e utilizza tale stream per gestire in modo adeguato i caratteri da stampare.

Invece di rendere *deprecated* l'intera classe PrintStream, sono stati resi deprecati i suoi costruttori. In questo modo tutti i programmi esistenti che, per stampare informazioni di debug o errori sullo schermo, utilizzano il metodo System.out.println o System.err.println potranno essere compilati senza ottenere warning. Si otterrà invece un warning se si costruisce esplicitamente un PrintStream.

In questi casi si dovrebbe invece costruire un PrintWriter, a cui si può passare un OutputStream, e che provvederà automaticamente a utilizzare un OutputStreamWriter intermedio per codificare in modo corretto i caratteri da stampare. I metodi per stampare sono quelli di PrintStream e cioè print e println, in grado di gestire i vari tipi primitivi di Java.

Altre classi e metodi deprecati

Quando si è trattato DataInputStream si è volutamente evitato il metodo readLine, per leggere una linea di testo dallo stream di input collegato, perché tale metodo è deprecated; questo è dovuto al fatto che non avviene la giusta conversione da byte a carattere. In tal caso si dovrebbe usare invece un BufferedReader, e il relativo metodo readLine.

Quindi, dato uno stream di input in, invece di creare un DataInputStream, se si vuole utilizzare il metodo readLine, si dovrà creare un BufferedReader:

```
BufferedReader d= new BufferedReader(new InputStreamReader(in));
```

Comunque si potrà continuare a utilizzare la classe DataInputStream per tutte le altre operazioni di lettura.

Anche la classe LineNumberInputStream, utilizzata per tenere traccia delle linee all'interno di uno stream tramite il metodo getLineNumber(), è deprecata; al suo posto si dovrà utilizzare un LineNumberReader.