Lezione 18

Le classi per l'Input/Output nel pacchetto java.io

Il package *java.io* incapsula le funzionalità di Input/Output (I/O) di Java. Come tutti i programmatori presto imparano, le applicazioni informatiche servono a ben poco se non acquisiscono dati dall'esterno, attraverso un canale di input, e se non emettono nuove informazioni, attraverso un canale di output. Dalle origini dell'informatica ad oggi, molte cose sono cambiate. Ora esistono potenti e colorate interfacce grafiche, con finestre, menù, pulsanti, etichette, icone che sembrano dei Pokémon, supporto alla multimedialità e molto altro ancora. Tuttavia, lo scopo primo di un programma, attraverso il tempo, è rimasto perlopiù lo stesso: acquisire dei dati dall'esterno, elaborarli ed emettere un risultato certo.

18.1 - La classe File

Una delle classi più importanti di *java.io* è *File*. Come il suo nome lascia facilmente intendere, lo scopo di *File* è incapsulare il concetto di file. Un oggetto *File* viene utilizzato per recuperare e manipolare le informazioni associate ad un file su disco (o su altra periferica di archiviazione). Tra queste informazioni ci sono la data dell'ultima modifica, il percorso, i permessi per la lettura e la scrittura e così via. Inoltre, *File* offre delle importanti utilità per la manipolazione e l'esplorazione del file system.

Costruttori pubblici	
`File(String pathname)`	Costruisce un oggetto *File* che corrisponde al percorso espresso.
`File(String parent, String child)`	Costruisce un oggetto *File* a partire da una stringa che identifica il percorso di base ed una seconda stringa che specifica il nome dell'elemento.
`File(File parent, String child)`	Costruisce un oggetto *File* a partire da un *File* che identifica il percorso di base ed una stringa che specifica il nome dell'elemento.
Costanti statiche pubbliche	
`String pathSeparator`	Il carattere che, nel sistema in uso, viene impiegato per separare i percorsi, riportato come stringa. Ad esempio, ; per Windows e : per gli UNIX.
`char pathSeparatorChar`	Il carattere che, nel sistema in uso, viene impiegato per separare i percorsi. Ad esempio, ; per Windows e : per gli UNIX.
`String separator`	Il carattere che, nel sistema in uso, viene impiegato per separare gli elementi di un percorso, riportato come stringa. Ad esempio, \ per Windows e / per gli UNIX.
`char separatorChar`	Il carattere che, nel sistema in uso, viene impiegato per separare gli elementi di un percorso. Ad esempio, \ per Windows e / per gli UNIX.

Metodi statici pubblici	
`File createTempFile(String prefix, String suffix)`	Crea un file temporaneo con il prefisso ed il suffisso specificati, restituendolo. Il file sarà posizionato nella cartella temporanea predefinita del sistema.
`File createTempFile(String prefix, String suffix, File directory)`	Crea un file temporaneo con il prefisso ed il suffisso specificati, restituendolo. Il file sarà posizionato nella cartella identificata dal terzo argomento.
`File[] listRoots()`	Restituisce l'elenco delle radici delle periferiche di archiviazione riconosciute dal sistema.
Metodi pubblici	
`boolean canRead()`	Restituisce *true* se il file può essere letto.
`boolean canWrite()`	Restituisce *true* se il file può essere scritto.
`int compareTo(File f)`	Paragona il maniera lessicografica il percorso del file corrente con quello dell'argomento fornito. Restituisce 0 se i due percorsi sono equivalenti, un valore negativo se il primo precede il secondo, un valore positivo se il secondo precede il primo.
`boolean createNewFile()`	Crea il file, ma solo se non esiste già. Se la creazione ha luogo, restituisce *true*.
`boolean delete()`	Cancella il file. Se l'operazione riesce, restituisce *true*.
`void deleteOnExit()`	Cancella il file alla chiusura del programma.
`boolean equals(Object o)`	Confronta il file con l'oggetto *o*. Restituisce *true* solo se *o* identifica il medesimo percorso sul disco.
`boolean exists()`	Restituisce *true* se il file esiste.
`String getAbsolutePath()`	Restituisce il percorso assoluto al file.
`String getCanonicalPath()`	Restituisce il percorso canonico al file.
`String getName()`	Restituisce il nome del file.
`String getParent()`	Restituisce il percorso della cartella genitrice del file.
`File getParentFile()`	Restituisce un oggetto *File* che rappresenta la cartella genitrice del file.
`String getPath()`	Restituisce il percorso del file.
`int hashCode()`	Calcola un codice hash per l'oggetto.
`boolean isAbsolute()`	Restituisce *true* se il file è espresso mediante un percorso assoluto.
`boolean isDirectory()`	Restituisce *true* se l'oggetto rappresenta una directory.

`boolean isFile()`	Restituisce *true* se l'oggetto rappresenta un file, nel senso letterale del termine.
`boolean isHidden()`	Restituisce *true* se il file è nascosto.
`long lastModified()`	Restituisce la data dell'ultima modifica del file, espressa con la tipica notazione in millisecondi.
`long length()`	Restituisce la grandezza del file in byte.
`File[] listFiles()`	Se l'oggetto rappresenta una directory, restituisce un array che contiene dei riferimenti a tutti i suoi file e a tutte le sue sotto-directory. In caso contrario, restituisce *null*.
`boolean mkdir()`	Se l'oggetto rappresenta una directory inesistente, la crea. Se l'operazione ha successo, restituisce *true*.
`boolean mkdirs()`	Crea tutte le directory inesistenti contenute nel percorso dell'oggetto. Se l'operazione ha successo, restituisce *true*.
`boolean renameTo(File dest)`	Cambia il nome del file. Se l'operazione ha successo, restituisce *true*.
`boolean setLastModified(long t)`	Cambia la data dell'ultima modifica del file. Se l'operazione ha successo, restituisce *true*.
`boolean setReadOnly()`	Imposta il file come di sola lettura. Se l'operazione ha successo, restituisce *true*.

Tabella 18.1
I principali membri della classe *File*.

Prima precisazione: la classe *File* rappresenta sia i file, nel senso letterale della parola, sia le directory.

Il costruttore più semplice è:

```
File(String pathname)
```

Questo costruttore genera un oggetto *File* che rappresenta il percorso espresso. Si possono esprimere percorsi sia relativi, sia assoluti.

```
File f = new File("prova.txt");
```

Nel caso appena mostrato, è stato usato un percorso relativo. Automaticamente, Java ritiene che il file *prova.txt* sia nell'attuale cartella di lavoro associata al programma. Quale sia la cartella di lavoro associata al programma, dipende dal sistema operativo. Ad ogni modo, il percorso assoluto della cartella di lavoro è tra le proprietà dell'ambiente. Può essere letto con una chiamata a:

```
System.getProperty("user.dir")
```

In Windows, ad esempio, questa cartella corrisponde al percorso dal quale il programma viene lanciato. Quindi, ipotizzando che tale percorso sia

```
C:\provejava\
```

l'oggetto *File* sopra creato corrisponderà a

```
C:\provejava\prova.txt
```

I percorsi relativi possono poi condurre verso altre cartelle, secondo le normali regole di composizione. Ad esempio:

```
sottocartella/prova.txt
../prova.txt
```

Non tutti i sistemi operativi impiegano lo stesso carattere per separare gli elementi di un percorso. Sotto Windows, ad esempio, si usa il contro-slash ("\"), mentre in Linux ha valore lo slash classico ("/"). Questo, ovviamente, costituisce un potenziale problema alla portabilità delle applicazioni. Per questo motivo, Java adotta le seguenti misure:

- Le costanti *File.separator* e *File.separatorChar* riportano il separatore valido nel sistema in uso.
- Se si usa lo slash classico per esprimere un percorso, Java è capace di adattarlo alle esigenze del sistema. Quindi basta usare lo slash alla UNIX anche sotto Windows e non si avranno problemi.

Un oggetto *File* può essere creato anche con un percorso assoluto. Qualcosa come:

```
File f1 = new File("C:\\miacartella\\miofile.ext"); // Windows
File f2 = new File("/home/miahome/miofile.ext"); // Linux
```

Naturalmente, i percorsi assoluti vanno usati con criterio, altrimenti la portabilità del programma va a farsi benedire (per non dire di peggio). Molti programmatori Java alle prime armi sbagliano proprio in questo. Java può realizzare programmi del tutto portabili, grazie alla sua struttura multipiattaforma. Tuttavia, se il programmatore si lega ai percorsi tipici di un solo sistema operativo, lo sforzo di Java diventa completamente inutile. Ricordo, inoltre, che due differenti versioni dello stesso sistema operativo possono usare strutture di cartelle diverse. Quindi, bisogna essere parsimoniosi nello studio dei percorsi da impiegare. In caso contrario si rischia di creare dei programmi che possono funzionare solo nel computer su cui è stato sviluppato. La portabilità di Java deve essere supportata da corretti e consapevoli criteri di progettazione. Inoltre, qualcosa può sempre sfuggire inavvertitamente, quindi è bene sperimentare i propri programmi con sistemi operativi diversi. L'ideale sarebbe svolgere delle prove almeno con Windows, Linux e Mac OS X.

Gli altri costruttori di *File* permettono la fabbricazione del percorso a partire da due elementi.

```
File f = new File("C:\\provejava\\", "prova.txt");
```

Questo file corrisponde al percorso:

```
C:\provejava\prova.txt
```

Ad esempio, se si desidera creare o recuperare un file nella directory home dell'utente, va bene la seguente formulazione:

```
File f = new File(System.getProperty("user.home"), "prova.txt");
```

Nel mio Windows XP, questo corrisponde a:

```
C:\Documents and Settings\Carlo\prova.txt
```

Sotto Linux, invece, ottengo il file:

```
/home/carlo/prova.txt
```

Dopo tutte queste premesse, non resta che esaminare un esempio:

```
import java.io.*;

public class FileTest1 {

  public static void main(String[] args) {
    File f = new File("prova.txt");
    System.out.println(f.getAbsolutePath());
    System.out.println(
      "Il file esiste? " + f.exists()
    );
    System.out.println(
      "Il file può essere letto? " + f.canRead()
    );
    System.out.println(
      "Il file può essere scritto? " + f.canWrite()
    );
    System.out.println(
      "Il file è nascosto? " + f.isHidden()
    );
    System.out.println(
      "E' una directory? " + f.isDirectory()
    );
    System.out.println(
      "E' proprio un file? " + f.isFile()
    );
    System.out.println(
      "Grandezza del file: " + f.length() + " byte"
    );
    System.out.println(
      "Ultima modifica: " + new java.util.Date(f.lastModified())
    );
  }

}
```

E' necessario creare un file chiamato *prova.txt*, nella cartella di lavoro dell'applicazione, che può essere riempito arbitrariamente. Quindi è possibile eseguire il programma. Si ottiene qualcosa del tipo:

```
/home/carlo/lezionidijava/prova.txt
Il file esiste? true
Il file può essere letto? true
```

```
Il file può essere scritto? true
Il file è nascosto? false
E' una directory? false
E' proprio un file? true
Grandezza del file: 15 byte
Ultima modifica: Tue Jul 21 20:04:29 CET 2005
```

Adesso si cancelli il file creato ed si esegua di nuovo il programma, per vedere come reagisce la classe *File* davanti ad un file inesistente. Si otterrà un output del tipo:

```
/home/carlo/librojava/prova.txt
Il file esiste? false
Il file può essere letto? false
Il file può essere scritto? false
Il file è nascosto? false
E' una directory? false
E' proprio un file? false
Grandezza del file: 0 byte
Ultima modifica: Thu Jan 01 01:00:00 CET 1970
```

Un secondo interessante esperimento è dato dal seguente codice:

```
import java.io.*;
import java.util.Date;

public class FileTest2 {

  public static void main(String[] args) {
    File homeDir = new File(System.getProperty("user.home"));
    System.out.println("Sono in " + homeDir.getAbsolutePath());
    File[] files = homeDir.listFiles();
    for (int i = 0; i < files.length; i++) {
      if (files[i].isDirectory()) {
        System.out.print("[D] ");
      } else {
        System.out.print("[F] ");
      }
      System.out.print(files[i].getName() + " ");
      System.out.print(
        "[" + files[i].length() + " byte] "
      );
      System.out.print(
        "[" + new Date(files[i].lastModified()) + "]"
      );
      System.out.println();
    }
  }

}
```

Lascio al lettore il compito di sperimentare il codice e di studiare il suo funzionamento.

18.2 - Gli stream

Tutto il sistema I/O di Java si basa sul concetto di *stream*. "Stream", in inglese, significa "flusso", "to stream" è "scorrere". Finora, mi sono riferito agli stream etichettandoli come "canali di comunicazione". La definizione è corretta, e può ancora considerarla valida.
Ogni stream è unidirezionale. Quindi, è immediatamente possibile dividere gli stream in due categorie: gli stream in entrata (*input stream*) e gli stream in uscita (*output stream*). Gli

input stream servono per inviare dati, gli output stream per riceverli. Insomma, dagli input stream si legge, mentre negli output stream si scrive. Se voglio leggere il contenuto di un file, devo stabilire un input stream con lo stesso. Viceversa, se devo scrivere dentro il file, ho bisogno di un output stream.
Ad ogni modo, il concetto di stream e quello di file non sono accoppiati. Uno stream è un canale di comunicazione generico, che non racchiude in sé informazioni su quale sia la sorgente (in caso di input) o la destinazione (in caso di output) del flusso. Questa potente astrazione consente di lavorare sempre alla stessa maniera, indipendentemente da chi è attaccato all'altra parte del canale. Un esempio di stream non diretti ad un file viene fornito dai canali *System.err*, *System.in* e *System.out* che, per default, sono associati alla riga di comando. Il bello degli stream è proprio questo: si usano sempre le stesse metodologie per scambiare i dati, indipendentemente da quello che avviene fisicamente. In Java, scambiare dati con la riga di comando, con un file o tramite una rete è tecnicamente la stessa cosa.

18.3 - Le classi astratte InputStream e OutputStream

In *java.io* sono definite due importanti classi astratte: *InputStream* e *OutputStream*. *InputStream* fornisce l'interfaccia di base per gli stream in entrata, mentre *OutputStream* cura gli stream in uscita.

Metodi pubblici	
`int available()`	Restituisce il numero di byte che ancora devono essere letti.
`void close()`	Chiude lo stream.
`void mark(int readlimit)`	Memorizza l'attuale posizione dello stream, in modo che poi sia possibile tornarci chiamando *reset()*. Dopo *readlimit* byte letti, comunque, la posizione viene dimenticata.
`boolean markSupported()`	Restituisce *true* se lo stream corrente supporta la funzionalità offerta da *mark()* e *reset()*.
`int read()`	Legge un byte dallo stream, e lo restituisce come valore *int*. Restituisce -1 se lo stream ha raggiunto il termine.
`int read(byte[] b)`	Come *read(b, 0, b.length)*.
`int read(byte[] b, int off, int len)`	Legge *len* byte dallo stream, salvandoli nell'array *b*, a partire dalla posizione *off*. Restituisce il numero di byte effettivamente letti, o -1 se la fine dello stream è stata raggiunta.
`void reset()`	Torna al punto raggiunto nello stream l'ultima volta che è stato chiamato *mark()*.
`long skip(long n)`	Avanza di *n* byte nello stream, senza leggerli. Restituisce il numero dei byte effettivamente saltati.

Tabella 18.2
I membri definiti dalla classe astratta *InputStream*.

Un *InputStream* è un flusso che va letto consecutivamente, dall'inizio alla fine. Le funzioni

mark() e ***reset()***, effettivamente funzionanti solo con alcuni tipi di stream, permettono di fare qualche passo indietro nella lettura. Tuttavia, questa è l'eccezione, non la regola. Un *InputStream* si legge, normalmente, dal primo all'ultimo byte, senza involuzioni. Il metodo ***read()*** legge un byte dallo stream di input, e lo restituisce sotto forma di *int*. Quando *read()* restituisce -1, non ci sono più dati da leggere. La chiamata a *read()* blocca il thread corrente fino all'effettiva ricezione del dato. La cosa, di norma, non è rilevante nella lettura di un file, ma va considerata quando si ricevono dati dalla rete o dalla riga di comando. Finché non c'è un dato da leggere (o finché lo stream non viene chiuso), il thread resta fermo. La prassi di utilizzo di un *InputStream* si articola sempre in tre passi:

1. Apri lo stream.
2. Leggi i dati che occorrono.
3. Chiudi lo stream.

Metodi pubblici	
`void close()`	Chiude lo stream.
`void flush()`	Scrive il buffer nello stream.
`void write(int b)`	Scrive un byte nel buffer o nello stream. Vengono scritti gli 8 bit più significativi del valore *int* trasmesso.
`void write(byte[] b)`	Scrive tutti i byte di *b* nel buffer o nello stream.
`void write(byte[] b, int off, int len)`	Scrive *len* byte nel buffer o nello stream, prelevandoli dall'array *b* a partire dalla posizione *off*.

Tabella 18.3
I membri definiti dalla classe astratta *OutputStream*.

Dentro un *OutputStream* i dati vanno scritti consecutivamente, byte dopo byte, servendosi dei metodi *write()*. Alcuni *OutputStream* possono essere *bufferizzati*, ed in questo caso i dati non raggiungono direttamente la loro destinazione finale. Un *buffer* è un segmento interno, nel quale i dati vengono "parcheggiati" momentaneamente. Quando il buffer è pieno, o quando si richiama il metodo *flush()*, avviene la reale consegna. I buffer permettono un minimo di ripensamento su quello che si è già scritto. Anche in questo caso, comunque, la funzionalità non è nella norma. I buffer, più che per altro, sono stati concepiti perché, in certe situazioni, garantiscono prestazioni migliori. La prassi di utilizzo di un *OutputStream* è:

1. Apri lo stream.
2. Scrivi i dati che devi scrivere.
3. Chiudi lo stream.

Vorrei sottolineare l'importanza della chiusura degli stream non più utili, sia in lettura sia in scrittura, per evitare lo spreco di importanti risorse.

18.4 - L'eccezione IOException

L'utilizzo degli stream genera codice rischioso, nel senso che diverse cose potrebbero andare storte. Ad esempio, si potrebbe tentare la lettura di un file inesistente, o la scrittura di un file protetto. In rete, le minacce si moltiplicano, perché il collo di bottiglia o l'improvvisa perdita della linea sono sempre dietro l'angolo. Circa tutti costruttori e tutti i

metodi delle classi di stream presenti in Java possono lanciare un particolare tipo di eccezione: *java.io.IOException*. Se si riceve una *IOException* significa che qualcosa è andato male durante un'operazione di I/O. In pseudo-codice, quindi, le operazioni di interazione con gli stream si eseguono sempre alla seguente maniera:

```
// Dichiara un riferimento ad uno stream.
try {
  // Inizializza lo stream.
  // Usa lo stream per i tuoi scopi.
} catch (IOException e) {
  // Gestisci il problema riscontrato.
} finally {
  // Chiudi lo stream, se è ancora aperto.
}
```

18.5 - Le classi FileInputStream e FileOutputStream

FileInputStream e *FileOutputStream* sono delle concrete derivazioni di *InputStream* e *OutputStream*. Costituiscono lo strumento più basilare per stabilire degli stream nei confronti di un file.

FileInputStream ha tre costruttori. Per ora se ne possono comprendere soltanto due.

```
FileInputStream(File f)
```

Questo costruttore apre uno stream utile per la lettura del file *f*.

```
FileInputStream(String pathname)
```

Questo costruttore permette di non passare attraverso un oggetto *File* per aprire lo stream. A tutti gli effetti, però, agisce appellandosi al primo costruttore visto. E' come se fosse:

```
FileInputStream(new File(String pathname))
```

Ambo i costruttori possono lanciare una *IOException*, nel caso non sia possibile stabilire lo stream richiesto (la causa più tipica è l'inesistenza del file, oppure la mancanza dei permessi di sicurezza necessari).
Per il resto, *FileInputStream* non fa altro che dare concreta implementazione ai metodi definiti da *InputStream*.
Passando alla pratica, ecco come leggere un file:

```
import java.io.*;

public class FileInputStreamTest {

  public static void main(String[] args) {
    // Il file da leggere.
    File f = new File("prova.txt");
    // Riferimento allo stream.
    FileInputStream in = null;
    try {
      // Stabilisco lo stream.
      in = new FileInputStream(f);
      // Leggo finché ci sono dati.
      int i;
      while ((i = in.read()) != -1) {
```

```
        char c = (char) i;
        System.out.print(c);
      }
    } catch (IOException e) {
      // Problema!
      System.out.println("Problema durante la lettura...");
    } finally {
      // Chiusura dello stream.
      if (in != null) try {
        in.close();
      } catch (IOException e) {}
    }
  }

}
```

Se non si desidera ricevere un errore al primo ufficiale tentativo di lettura di un file, si inserisca nella cartella di lavoro dell'applicazione un documento di testo chiamato *prova.txt*, che riempito arbitrariamente. L'applicazione riprodurrà in output il suo contenuto.

FileOutputStream ha cinque costruttori. Quattro sono immediatamente comprensibili.

```
FileOutputStream(File f)
```

Questo costruttore stabilisce uno stream di scrittura verso il file *f*. Se il file specificato non esiste, lo crea, altrimenti lo sovrascrive.

```
FileOutputStream(File f, boolean append)
```

Se *append* è *false*, è esattamente come il caso precedente. Se *append* è *true*, e se il file già esiste, quello che sarà convogliato nello stream non sovrascriverà i contenuti già esistenti. Semplicemente, saranno aggiunti in coda.

```
FileOutputStream(String pathname)
FileOutputStream(String pathname, boolean append)
```

Questi due costruttori sono esattamente come i precedenti, solo che usano una stringa al posto di un oggetto *File*. E' esattamente la stessa cosa che accade con *FileInputStream*.
Naturalmente, i costruttori di *FileOutputStream* possono propagare una *IOException*, quando il canale di scrittura non può essere stabilito.
Ecco l'esempio pratico:

```
import java.io.*;

public class FileOutputStreamTest {

  public static void main(String[] args) {
    // Il file da leggere.
    File f = new File("prova.txt");
    // Riferimento allo stream.
    FileOutputStream out = null;
    try {
      // Stabilisco lo stream.
      out = new FileOutputStream(f);
      // Scrivo "Ciao".
      out.write("Ciao".getBytes());
```

```
    } catch (IOException e) {
      // Problema!
      System.out.println("Problema durante la scrittura...");
    } finally {
      // Chiusura dello stream.
      if (out != null) try {
        out.close();
      } catch (IOException e) {}
    }
  }

}
```

Questa applicazione genera un file chiamato *prova.txt*, e al suo interno scrive "Ciao".

Mettiamo insieme *FileInputStream* e *FileOutputStream*:

```
import java.io.*;

public class FileCopier {

  public static void main(String[] args) {
    // Controlla gli argomenti forniti.
    if (args.length != 2) {
      System.out.println();
      System.out.println(
        "Utilizzo scorretto..."
      );
      System.out.println();
      System.out.println(
        "Devi rispettare la sintassi:"
      );
      System.out.println();
      System.out.println(
        "java FileCopier sorgente destinazione"
      );
      System.out.println();
      return;
    }
    // Crea i file associati.
    File s = new File(args[0]);
    File d = new File(args[1]);
    // Esegue un po' di controlli iniziali.
    // Non sono indispensabili, la gestione delle
    // eccezioni basterebbe, ma così si possono
    // dare subito delle informazioni all'utente.
    if (!s.exists()) {
      System.out.println();
      System.out.println(
        "Il file sorgente non esiste..."
      );
      System.out.println();
      return;
    }
    if (s.isDirectory()) {
      System.out.println();
      System.out.println(
        "Copio solo i file, non le directory..."
      );
      System.out.println();
```

```
  return;
}
if (!s.canRead()) {
  System.out.println();
  System.out.println(
    "Non posso leggere il file sorgente..."
  );
  System.out.println();
  return;
}
if (d.exists() && d.isDirectory()) {
  System.out.println();
  System.out.println(
    "La destinazione non può essere una directory..."
  );
  System.out.println();
  return;
}
if (d.exists() && !d.canWrite()) {
  System.out.println();
  System.out.println(
    "Non posso sovrascrivere il file di destinazione..."
  );
  System.out.println();
  return;
}
if (s.equals(d)) {
  System.out.println();
  System.out.println(
    "Sorgente e destinazione coincidono..."
  );
  System.out.println();
  return;
}
// Inizia la procedura di copia.
FileInputStream in = null;
FileOutputStream out = null;
try {
  // Apre gli stream.
  in = new FileInputStream(s);
  out = new FileOutputStream(d);
  // Copia un KB alla volta, per questo usa
  // un array di 1024 byte.
  byte[] mioBuffer = new byte[1024];
  // Legge e copia tutti i byte letti.
  int l;
  while ((l = in.read(mioBuffer)) != -1) {
    out.write(mioBuffer, 0, l);
  }
  // Messaggio di successo.
  System.out.println();
  System.out.println(
    "Operazione eseguita!"
  );
  System.out.println();
} catch (IOException e) {
  // Problemi...!
  System.out.println();
  System.out.println(
    "Copia fallita..."
  );
```

```
      System.out.println();
      System.out.println(
        "Messaggio d'errore:"
      );
      System.out.println();
      System.out.println(e.getMessage());
      System.out.println();
    } finally {
      // Chiusura degli stream.
      if (out != null) try {
        out.close();
      } catch (IOException e) {}
      if (in != null) try {
        in.close();
      } catch (IOException e) {}
    }
  }

}
```

Questo esempio è davvero interessante, e consiglio di esaminarlo a fondo. Scopo dell'applicazione è copiare un file. Il software si usa da riga di comando, al seguente modo:

```
java FileCopier sorgente destinazione
```

Ad esempio:

```
java FileCopier prova.txt copia_di_prova.txt
```

18.6 - La classe PrintStream

PrintStream deriva da *OutputStream*, specializzandola con numerosi nuovi metodi. Dentro *java.io* ci sono parecchi stream derivati da *InputStream* e da *OutputStream*. Non saranno tutti esaminati, sia perché alcuni di essi sono assai peculiari, sia perché, dopo aver appreso le nozioni di base, il loro utilizzo è davvero intuitivo. Nonostante questo, si dedicherà qualche minuto a *PrintStream*, se non altro perché i canali *System.err* e *System.out* sono proprio di questo tipo.
La prima peculiarità di *PrintStream* è che i suoi costruttori ed i suoi metodi aggiuntivi non propagano delle *IOException*. Quindi, è possibile usare un *PrintStream* senza doversi ingarbugliare con le strutture *try ... catch*. Questa scelta, ovviamente, ha dei pro e dei contro. I pro già li ho elencati, i contro sono costituiti dal fatto che non si può sapere se le operazioni effettuate hanno avuto successo oppure no.
Il principale costruttore di *PrintStream* è:

```
PrintStream(OutputStream out)
```

PrintStream è ideale quando si gestiscono informazioni testuali. Per il trattamento di informazioni binarie, è meglio ricorrere ad altro. I metodi aggiuntivi di *PrintStream* sono tutti del tipo ***print()*** e ***println()***. Il primo invia semplicemente in output il dato passato; il secondo introduce un ritorno a nuova riga dopo di esso. Ne esistono tantissime varianti, in pratica uno per ogni tipo semplice di Java, più altri per gli array di caratteri, per le stringhe e per tutti gli altri oggetti. In particolare, le varianti di *print()* e *println()* che gestiscono gli *Object* fanno ricorso al metodo *toString()*, per inviare in output una loro rappresentazione testuale.

PrintStream cerca di arginare il problema indotto dalla mancata gestione delle eccezioni attraverso il metodo *checkError()*, che restituisce *true* nel caso in cui l'ultima operazione tentata sia fallita.
Il seguente esempio dimostra come dirottare *System.out* dalla riga di comando ad un file di testo:

```
import java.io.*;

public class PrintStreamTest {

  public static void main(String[] args) {
    File log = new File("log.txt");
    try {
      System.setOut(
        new PrintStream(
          new FileOutputStream(log)
        )
      );
    } catch (IOException e) {}
    System.out.println("Ciao!");
    System.out.println("Eccomi qui!");
  }

}
```

Faccio notare che l'eccezione *IOException* è propagata dalla chiamata a *FileOutputStream*, e non dal costruttore di *PrintStream*.

18.7 - Serializzazione

Nel dialetto di Java, un oggetto è *serializzabile* quando può essere convogliato attraverso uno stream, ad esempio per essere conservato su disco o per essere trasmesso attraverso una rete. Quando lo stream sarà recuperato ed interpretato in senso inverso (operazione detta di *deserializzazione*), sarà possibile ricostruire l'oggetto, ripristinando l'esatto stato interno che aveva al momento della sua serializzazione. Perno della serializzazione sono i due stream ***ObjectOutputStream*** e ***ObjectInputStream***, rispettivamente per la serializzazione e per la deserializzazione degli oggetti. Ambo le classi hanno costruttori che accettano in ingresso, rispettivamente, un *OutputStream* ed un *InputStream* già stabiliti. In pratica, lavorano da filtro verso dei canali già instaurati. *ObjectOutputStream* definisce il metodo ***writeObject()***, che permette di serializzare un oggetto all'interno dello stream. In maniera speculare, *ObjectInputStream* definisce ***readObject()***, che recupera l'intero oggetto precedentemente serializzato, restituendo un riferimento di tipo *Object*.
Un oggetto è serializzabile quando implementa l'interfaccia *java.io.**Serializable***. Un po' come avviene per *java.lang.Cloneable*, *Serializable* non richiede alcun metodo. Semplicemente, etichetta l'oggetto come disponibile alla serializzazione.
Il migliore approccio alla serializzazione che può essere offerto, in casi come questo, passa sempre attraverso la dimostrazione di un esempio pratico. Si realizzi la classe *Persona*, che implementa l'interfaccia *Serializable*:

```
import java.io.*;

public class Persona implements Serializable {

  private String nome;
  private String cognome;
```

```
  public Persona(String nome, String cognome) {
    this.nome = nome;
    this.cognome = cognome;
  }

  public String getNome() {
    return nome;
  }

  public String getCognome() {
    return cognome;
  }

}
```

Si tenti la serializzazione di un oggetto *Persona*, con il seguente codice:

```
import java.io.*;

public class Serializza {

  public static void main(String[] args) {
    // Creazione di un oggetto Persona.
    Persona p = new Persona("Mario", "Rossi");
    // Il file in cui riporre la serializzazione.
    File f = new File("mariorossi.ser");
    // Serializzazione dell'oggetto.
    ObjectOutputStream out = null;
    try {
      out = new ObjectOutputStream(
        new FileOutputStream(f)
      );
      out.writeObject(p);
      System.out.println("Oggetto serializzato!");
    } catch (IOException e) {
      System.out.println("Impossibile serializzare...");
    } finally {
      if (out != null) try {
        out.close();
      } catch (IOException e) {}
    }
  }

}
```

L'esecuzione dell'applicazione genera il file *mariorossi.ser*, che contiene l'immagine serializzata dell'oggetto *Persona* generato in apertura di codice, vale a dire di “Mario Rossi”. Ora il file può essere recuperato e deserializzato, come mostrato di seguito:

```
import java.io.*;

public class Deserializza {

  public static void main(String[] args) {
    // Il file da recuperare.
    File f = new File("mariorossi.ser");
    // Deserializzazione.
    ObjectInputStream in = null;
    try {
```

```
      in = new ObjectInputStream(
        new FileInputStream(f)
      );
      Persona p = (Persona)in.readObject();
      System.out.println("Deserializzazione riuscita!");
      System.out.println("Nome: " + p.getNome());
      System.out.println("Cognome: " + p.getCognome());
    } catch (IOException e) {
      System.out.println("Impossibile deserializzare...");
    } catch (ClassNotFoundException e) {
      // Bisogna gestire l'eventualità che la classe
      // Persona non venga trovata.
      System.out.println("Impossibile deserializzare...");
    } finally {
      if (in != null) try {
        in.close();
      } catch (IOException e) {}
    }
  }

}
```

L'esecuzione di questa classe, se tutto va a buon fine, produce l'output:

```
Deserializzazione riuscita!
Nome: Mario
Cognome: Rossi
```

Lo stato interno dell'oggetto è stato mantenuto, attraverso le operazioni di serializzazione e deserializzazione. Riassumendo, la serializzazione è una tecnica che fornisce agli oggetti di Java delle doti di persistenza.

18.8 - Le classi astratte Reader e Writer

Gli stream di tipo più basilare non tengono conto del tipo dei dati scambiati. Semplicemente, trasportano i dati attraverso il flusso, senza curarsi del loro effettivo contenuto. Per questo, classi come *InputStream* e *OutputStream* sono idonee alla gestione di sequenze binarie (immagini, audio, eseguibili, e così via). Per la gestione dei flussi di testo, è meglio servirsi di classi più specializzate. In questo modo, anziché ragionare per byte, si potrà ragionare per caratteri e per stringhe. Le classi astratte *Reader* e *Writer* sono il parallelo di *InputStream* e *OutputStream*, nell'ambito dei flussi di testo. Il loro funzionamento, ad ogni modo, richiama molte nozioni già note.

Metodi pubblici	
`void close()`	Chiude lo stream.
`void mark(int readlimit)`	Memorizza l'attuale posizione dello stream, in modo che poi sia possibile tornarci chiamando *reset()*. Dopo *readlimit* caratteri letti, comunque, la posizione viene dimenticata.
`boolean markSupported()`	Restituisce *true* se lo stream corrente supporta la funzionalità offerta da *mark()* e *reset()*.
`int read()`	Legge un carattere dallo stream, e lo restituisce come valore *int*. Restituisce -1 se lo stream ha raggiunto il termine.

`int read(char[] c)`	Come *read(c, 0, c.length)*.
`int read(char[] c, int off, int len)`	Legge *len* caratteri dallo stream, salvandoli nell'array *c*, a partire dalla posizione *off*. Restituisce il numero di caratteri effettivamente letti, o -1 se la fine dello stream è stata raggiunta.
`void reset()`	Torna al punto raggiunto nello stream l'ultima volta che è stato chiamato *mark()*.
`long skip(long n)`	Avanza di *n* caratteri nello stream, senza leggerli. Restituisce il numero dei caratteri effettivamente saltati.

Tabella 18.4
I membri definiti dalla classe astratta *Reader*.

Metodi pubblici	
`void close()`	Chiude lo stream.
`void flush()`	Scrive il buffer nello stream.
`void write(int c)`	Scrive un carattere nel buffer o nello stream.
`void write(char[] c)`	Scrive tutti i caratteri di *c* nel buffer o nello stream.
`void write(byte[] c, int off, int len)`	Scrive *len* caratteri nel buffer o nello stream, prelevandoli dall'array *c* a partire dalla posizione *off*.
`void write(String str)`	Scrive tutti i caratteri della stringa *str* nel buffer o nello stream.
`void write(String srt, int off, int len)`	Scrive *len* caratteri nel buffer o nello stream, prelevandoli dalla stringa *str* a partire dalla posizione *off*.

Tabella 18.5
I membri definiti dalla classe astratta *Writer*.

Come è possibile osservare dalle tabelle riassuntive, *Reader* e *Writer* sono molto simili a *InputStream* e *OutputStream*, con l'unica differenza che ragionano per caratteri anziché per byte.

18.9 - Le classi InputStreamReader e OutputStreamWriter

La classe *InputStreamReader* è un ponte tra *InputStream* e *Reader*, mentre *OutputStreamWriter* è un ponte tra *OutputStream* e *Writer*.

InputStreamReader è un *Reader*, in altre parole estende la classe *Reader* e ne concretizza i metodi. Il suo costruttore fondamentale è:

```
InputStreamReader(InputStream in)
```

Dunque, il suo compito è trasformare un *InputStream* in un *Reader*.

La situazione è analoga per *OutputStreamWriter*, che estende la classe *Writer* e ne

concretizza i metodi. Il suo costruttore fondamentale è:

```
OutputStreamWriter(OutputStream out)
```

Quindi, il suo compito è trasformare un *OutputStream* in un *Writer*.

I costruttori appena esaminati non propagano eccezioni, giacché si collegano a canali già stabiliti.

18.10 - Le classi FileReader e FileWriter

FileReader e *FileWriter* estendono, rispettivamente, *InputStreamReader* e *OutputStreamWriter*. Come è facile immaginare, sussiste un'estrema somiglianza tra *FileReader* e *FileInputStream*, e tra *FileWriter* e *FileOutputStream*.

FileReader definisce i costruttori:

```
FileReader(File f)
FileReader(String pathname)
```

FileWriter, invece, ha:

```
FileWriter(File f)
FileWriter(File f, boolean append)
FileWriter(String pathname)
FileWriter(String pathname, boolean append)
```

A questo punto, è lecito chiedersi: per leggere un file, si deve usare *FileInputStream* o *FileReader*? E per scrivere un file, è meglio *FileOutputStream* o *FileWriter*? La risposta è: dipende dal tipo di file. I surrogati di *Reader* e *Writer* sono consigliati quando si ha a che fare con del testo, mentre gli stream di tipo classico gestiscono meglio le informazioni binarie.

18.11 - Le classi BufferedReader e BufferedWriter

Importanti derivazioni di *Reader* e *Writer* sono *BufferedReader* e *BufferedWriter*. Queste due classi meritano una menzione speciale, perché gestiscono il concetto di riga.

BufferedReader ha un costruttore del tipo:

```
BufferedReader(Reader in)
```

Rispetto a *Reader*, *BufferedReader* definisce il metodo *readLine()*. Questo metodo ritorna una stringa che corrisponde ad un'intera riga letta dal canale di comunicazione. Se il flusso è giunto al termine, restituisce *null*.

BufferedWriter ha un costruttore del tipo:

```
BufferedWriter(Writer out)
```

Rispetto a *Writer*, *BufferedWriter* definisce il metodo *newLine()*. Questo metodo introduce nel canale un ritorno a nuova riga.

I costruttori di *BufferedReader* e *BufferedWriter* non propagano eccezioni, giacché si

rifanno a dei canali già instaurati.

Il seguente esempio insegna come leggere un file di testo riga per riga:

```
import java.io.*;

public class BufferedReaderTest1 {

  public static void main(String[] args) {
    // Il file da leggere.
    File f = new File("prova.txt");
    // Lettura riga per riga.
    BufferedReader in = null;
    try {
      in = new BufferedReader(new FileReader(f));
      int i = 1;
      String line;
      while ((line = in.readLine()) != null) {
        System.out.println(i + ": " + line);
        i++;
      }
    } catch (IOException e) {
      System.out.println("Impossibile leggere il file");
    } finally {
      if (in != null) try {
        in.close();
      } catch (IOException e) {}
    }
  }

}
```

Prima di avviare l'applicazione, si deve realizzare un file chiamato *prova.txt* con qualche riga arbitraria al suo interno.

Dulcis in fundo, ecco come si preleva un input dalla riga di comando:

```
import java.io.*;

public class BufferedReaderTest2 {

  public static void main(String[] args) {
    BufferedReader in = new BufferedReader(
      new InputStreamReader(
        System.in
      )
    );
    try {
      while (true) {
        System.out.print(
          "Scrivi qualcosa ('quit' per uscire): "
        );
        String line = in.readLine();
        if (line != null) {
          if (line.equals("quit")) {
            break;
          } else {
            System.out.println(
              "Hai scritto: " + line
```

```
            );
          }
        }
      }
    } catch (IOException e) {}
  }

}
```

L'istruzione cardine è:

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

In breve:

- *System.in* è un *InputStream*.
- Con *new InputStreamReader(System.in)* si trasforma il canale di input della riga di comando in un *Reader*.
- Con il passaggio a *BufferedReader* si guadagna la possibilità di usare il metodo *readLine()*, che consente di acquisire una riga per volta.

E' curioso osservare come gli stream, che semplificano notevolmente ogni lavoro di interazione con l'esterno, abbiano reso ostica un'operazione che, in altri linguaggi, è sempre tra le prime apprese. Ma, lo assicuro, questa è l'unica cosa che gli stream di Java hanno complicato. Le altre operazioni tipiche sono tutte più semplici, eleganti, robuste e stabili.