Il sistema Input/Output di Java

Il sistema Input/Output di Java

La libreria standard di Java offre una vasta gamma di classi per la gestione dell'I/O. Queste permettono di gestire tanto un collegamento di rete, quanto un buffer di memoria o un file su disco.

Sebbene questi dispositivi siano fisicamente diversi, sono gestiti dalla stessa astrazione: il *flusso* o *stream*.

Uno *stream* è una entità logica che produce o consuma informazioni.

Tutti gli *stream* si comportano 'logicamente' allo stesso modo anche se i dispositivi fisici a cui sono collegati sono differenti.

La classe *File*...

La classe *File* rappresenta sia il nome di un particolare file che i nomi di un insieme di file presenti in una cartella (*directory*).

Se denota un insieme di file, si può conoscere tale insieme attraverso il metodo *list()*, che restituisce un array di *String*, gli elementi di tale insieme.

In tal modo abbiamo una lista completa di tutti i file presenti nella directory.

È possibile anche selezionare solo un tipo particolare di oggetti della cartella (per esempio tutti i file .java presenti nella cartella) ricorrendo a un *filtro* (*directory filter*).

…La classe *File*.

Esempio

dirlist.doc

L'interfaccia *FilenameFilter* è piuttosto semplice:

```
public interface FilenameFilter {
boolean accept(File dir, String name);
}
```

Una classe che la implementa deve fornire un metodo **accept( )** che il metodo **list( )** della classe **File** potrà richiamare (call back) per determinare quali nomi di file devono essere inclusi nella lista.

…La classe *File*.

Gli argomenti del metodo **accept( )** sono due:

- Un oggetto **File** che rappresenta la directory in cui si deve trovare un file,
- Un oggetto **String** contenente il nome del file.

Nell'esempio precedente, con il metodo *accept* ci si assicurava che si stava lavorando solo con il nome del file, senza informazione sul percorso.

…La classe *File*.

Si può usare la classe *File* anche per:

- creare nuove cartelle o interi percorsi;
- accedere alle caratteristiche dei file (dimensione, proprietà, ultima modifica);
- verificare se un oggetto *File* è una directory o un file;
- eliminare un file.

Esempio

crea directory.doc

Input e Output

Le librerie di classi Java per I/O sono divise in librerie di input e librerie di output.

Ogni classe per l'input è derivata dalle classi *InputStream* o *Reader* che hanno metodi di base chiamati *read()* per leggere singoli byte oppure array di byte.

Analogamente le classi per l'output sono derivate dalle classi *OutputStream* o *Writer* che hanno i metodi di base chiamati *write()* per scrivere singoli byte o array di byte.

Tipi di InputStream…

Il compito della classe astratta *InputStream* è di descrivere i metodi per classi che producono input da diverse sorgenti quali:

- un array di byte;
- un oggetto *String*;
- un file;
- un '*pipe*';
- una sequenza di altri stream convogliati in un unico stream;
- altre sorgenti come ad esempio le connessioni Internet.

Ognuna di queste sorgenti ha una sottoclasse di *InputStream* associata.

…Tipi di InputStream

Tabella

classi di input.doc

Tipi di OutputStream

Il compito di *OutputStream* è di rappresentare classi che producono output negli stessi tipi di sorgenti di *InputStream* (ad eccezione del tipo String).

Tabella

classi di output.doc

La lettura dei dati dallo stream di input mediante l'uso di *FilterInputStream*

Tipi di FilterInputStream:

tipi di Filterinputstream.doc

Di queste, la classe più utilizzata è *DataInputStream* mentre le altre classi forniscono delle funzionalità più avanzate.

Java(TM) 2 Platform, Standard Edition, v1.4.2 API Specification: Class DataInputStream

La scrittura dei dati sullo stream di output mediante la classe *FilterOutputStream*

Le classi che si occupano si scrivere sullo stream di output sono le seguenti (tipi di *FilterOutputStream*) :

tipi di FilterOutputstream.doc

Input e Output…

Mentre le classi *InputStream* e *OutputStream* forniscono funzionalità orientate ai byte, Java 1.1 prevede anche due nuove classi *Reader* e *Writer* che hanno funzionalità analoghe alle precedenti ma orientate ai caratteri (Unicode compatibili).

Quasi tutte classi di stream I/O di Java hanno le corrispondenti classi *Reader* e *Writer*.

In generale conviene provare prima con le librerie orientate ai caratteri e usare le librerie orientate ai byte nel caso sorgano dei problemi.

…Input e Output…

Nella tabella seguente sono mostrate le corrispondenze tra le due librerie.

readwriter1.doc

…Input e Output...

Le seguenti classi restano invariate sia in Java 1.0 che in Java 1.1:

- *DataOutputStream;*
- *File;*
- *RandomAccessFile;*
- *SequenceInputFile.*

La classe *RandomAccesFile* è usata per i file contenenti record di dimensione nota in modo tale che si possa accedere ai record usando il metodo *seek().*

…Input e Output.

Essenzialmente, *RandomAccessFile* lavora come un *DataInputStream* e *DataOutputStream* insieme. Il suo costruttore richiede come argomento anche se si sta accedendo al file in lettura ("*r*") o in lettura-scrittura ("*rw*") in modo simile alla funzione *fopen()* del C.

Questo esempio mostra i tipici usi degli Stream

Esempio

IOStreamDemo.doc

Standard I/O…

Tutti gli input di un programma arrivano dal dispositivo standard di input, tutti gli output vanno sul dispositivo standard di output e tutti i messaggi di errore sono inviati sul dispositivo standard di errore.

La classe *System* di Java include tre variabili di classe per il dispositivo standard di input (*System.in*), di output (*System.out*) e di errore (*System.err*).

System.out e *System.err* sono "pre-wrappati" come oggetti della classe *PrintStream*.

System.in è, invece, un *InputStream* grezzo, senza "pre-wrapping".

…Standard I/O…

Per leggere dallo standard input mediante una *readLine()* è necessario wrappare *System.in* nella classe *BufferedReader*, per fare ciò è necessario convertire System.in in un *Reader* usando *InputStreamReader*.

Esempio

come leggere dallo standard input.doc

Si nota che *System.in* dovrebbe essere bufferizzato (come la maggior parte degli stream).

Ridirezionare gli standard di I/O

La classe *System* di Java permette di ridirezionare gli stream di I/O standard usando semplici invocazioni a metodi di classe: *setIn(InputStream)*, *setOut(PrintStream)*, *setErr(PrintStream)*.Il ridirezionamento dello stream di output è molto utile quando si deve visualizzare una grande mole di dati che non rientra in uno schermo.

Esempio

ridirezionamento dello standard IO.doc

Il programma aggiunge lo standard di input in un file e ridireziona lo standard di output e quello degli errori in un altro file.

Compressione dei dati…

La libreria I/O di Java contiene delle classi che supportano la lettura e scrittura di flussi in formato compresso.

Queste classi *non* sono derivate da *Reader* e *Writer* ma sono parte delle gerarchie di *InputStream* e *OutputStream*, poiché la libreria di compressione lavora con i byte.

- *DeflaterOutputStream***:** classe base per le classi di compressione;
- *ZipOutputStream*: un *DeflaterOutputStream* che comprime i dati nel formato *zip*;

…Compressione dei dati.

- *GZIPOutputStream*: un *DeflaterOutputStream* che comprime i dati nel formato *GZIP*;
- *InflaterInputStream*: classe base per le classi di decompressione;
- *ZipInputStream*: un *InflaterInputStream* che decomprime i dati memorizzati nel formato *zip*;
- *GZIPInputStream*: un *InflaterInputStream* che decomprime i dati memorizzati nel formato *GZIP*.

Esempio

compressione in formato GZIP.doc

La libreria che supporta il formato zip è più ampia di quella del formato GZIP.

I file JAR…

Il formato *Zip* è anche usato nel formato di file JAR (*Java Archives*) che rappresenta un modo per raggruppare un insieme di file (audio, immagini, .class) in un unico file compresso indipendente dalla piattaforma.

Inoltre i file JAR sono molto utili per Internet.

Infatti quando un applet è composto da diversi file *.class*, il Web browser deve effettuare diverse richieste al server. Raggruppando tutti i file *.class* in unico file JAR è necessaria solo una richiesta pertanto il trasferimento è più veloce e più sicuro (i file JAR possono essere firmati digitalmente).

…I file JAR…

Un file JAR è costituito da un singolo file contenente una collezione di file zippati e un file "*manifesto*" che li descrive.

Il manifesto si compone di una sezione principale che contiene informazioni sulla sicurezza e configurazione dello stesso file JAR, e di tante sezioni che contengono gli attributi dei package e dei file contenuti nel file JAR.

L'utente può creare il proprio file manifesto, oppure lasciare al programma *jar* il compito di crearne uno sulla base delle opzioni specificate.

…I file JAR.

Per creare un file JAR l'ambiente JDK mette a disposizione l'utility *jar* che viene invocata da linea di comando con la seguente sintassi:

```
jar [options] destination [manifest] inputfile(s)
```

Per le opzioni basta consultare lo help del comando *jar*.

c	Creates a new or empty archive.
t	Lists the table of contents.
x	Extracts all files.
x file	Extracts the named file.
f	Says: "I'm going to give you the name of the file." If you don't use this, **jar** assumes that its input will come from standard input, or, if it is creating a file, its output will go to standard output.
m	Says that the first argument will be the name of the user-created manifest file.
v	Generates verbose output describing what **jar** is doing.
0	Only store the files; doesn't compress the files (use to create a JAR file that you can put in your classpath).
M	Don't automatically create a manifest file.

…I file JAR.

Esempi:

```
jar cf myJarFile.jar *.class
```

per comprimere, e

```
jar tf myJarFile.jar
```

per visualizzare i file in `myJarFile.jar`.

Serializzazione degli oggetti

Al termine della esecuzione di un programma, i dati utilizzati vengono distrutti.

Per poterli preservare fra due esecuzioni consecutive è possibile ricorrere all'uso dell'I/O su file.

Nel caso di semplici strutture o di valori di un tipo primitivo, questo approccio è facilmente implementabile.

I problemi si presentano quando si desidera memorizzare strutture complesse (e.g., collezioni di oggetti): in questo caso occorrerebbe memorizzare tutte le parti di un oggetto separatamente, secondo una ben precisa rappresentazione, per poi ricostruire l'informazione dell'oggetto all'occorrenza. Questo processo può risultare impegnativo e noioso.

Serializzazione degli oggetti

La ***persistenza*** di un oggetto indica la capacità di un oggetto di poter "vivere" separatamente dal programma che lo ha generato.

Java contiene un meccanismo per creare oggetti persistenti, detto ***serializzazione degli oggetti***: un oggetto viene serializzato trasformandolo in una sequenza di byte che lo rappresentano. In seguito questa rappresentazione può essere usata per ricostruire l'oggetto originale. Una volta serializzato, l'oggetto può essere memorizzato in un file o inviato a un altro computer perché lo utilizzi.

Serializzazione degli oggetti

In Java la serializzazione viene realizzata tramite

- una interfaccia e
- due classi.

Ogni oggetto che si vuole serializzare deve implementare l'interfaccia ***Serializable***, la quale non contiene metodi e serve soltanto al compilatore per comprendere che un oggetto di quella determinata classe può essere serializzato.

Per serializzare un oggetto si invoca poi il metodo ***writeObject*** della classe ***ObjectOutputStream***; per deserializzarlo si usa il metodo ***readObject*** della classe ***ObjectInputStream***.

Serializzazione degli oggetti

ObjectInputStream e *ObjectOutputStream* sono stream di manipolazione e devono essere utilizzati congiuntamente a un *OutputStream* e un *InputStream*. Quindi gli stream di dati effettivamente usati dall'oggetto serializzato possono rappresentare file, comunicazioni su rete, stringhe, ecc.

Esempio:

```
FileOutputStream outFile = new
FileOutputStream("info.dat");
ObjectOutputStream outStream = new
ObjectOutputStream(outFile);
outStream.writeObject(myCar)
```

dove `myCar` è un oggetto di una classe `Car` definita dal programmatore e che implementa l'interfaccia `Serializable`.

Serializzazione degli oggetti

Per poter leggere l'oggetto serializzato e ricaricarlo in memoria centrale si procederà come segue:

```
FileInputStream inFile = new
  FileInputStream("info.dat");
ObjectInputStream inStream = new
  ObjectInputStream(inFile);
Car myCar = (Car) inStream.readObject();
```

La serializzazione di un oggetto si occupa di serializzare tutti gli eventuali riferimenti ad esso collegati. Dunque, se la classe `Car` contenesse dei riferimenti (variabili di classe o di istanza) a oggetti di classe `Engine`, questa verrebbe serializzata automaticamente e diverrebbe parte della serializzazione di `Car`. La classe `Engine` dovrà, pertanto, implementare anch'essa l'interfaccia serializable.

Serializzazione degli oggetti

Attenzione:

Gli attributi di classe, cioè definiti come **static**, NON vengono serializzati. Per poterli salvare occorre provvedere in modo personalizzato.

Esempio:

Attributo statico nroNavi in Nave.

```
class Nave implements Serializable{
    private static int nroNavi=1;
    private int nroNave;
    private String nomeNave;
    Nave(String nomeNave){
       nroNave=nroNavi++;
       this.nomeNave=nmeNave;
    }
    public String toString(){
       return nomeNave+":"+i;
    }
    public void salva() throws FileNotFoundException, IOException {
        ObjectOutputStream out = new ObjectOutputStream(new
                        FileOutputStream("info.dat"));
        out.writeObject(this);
        out.writeObject(nroNavi);
        out.close();
    }
    public static Nave carica() throws FileNotFoundException,
       IOException {
        ObjectInputStream in = new ObjectInputStream(new
                        FileInputStream("info.dat"));
        Nave n=(Nave)in.readObject();
        Nave.nroNavi=in.readObject();
        in.close();
        return n;
    }
```

Serializzazione degli oggetti

Molte classi della libreria standard Java implementano l'interfaccia Serializable in modo da essere serializzate quando necessario.

Esempi di tali classi sono: String, HashMap, …

Ovviamente, nel caso di HashMap anche gli oggetti memorizzati nella struttura dati devono implementare l'interfaccia Serializable.

Il modificatore *transient*

A volte, quando si serializza un oggetto, si può desiderare di escludere delle informazioni, ad esempio, una password.

Questo accade quando le informazioni vengono trasmesse via rete.

Il pericolo è che, pur dichiarandola con visibilità privata, la password possa essere letta e usata da soggetti non autorizzati quando viene serializzata.

Un'altra ragione potrebbe essere quella di voler escludere l'informazione dalla serializzazione semplicemente perché tale informazione può essere semplicemente riprodotta quando l'oggetto viene deserializzato. In questo modo lo stream di byte che contiene l'oggetto serializzato non ha informazioni inutili che ne aumenterebbero la dimensione.

Il modificatore *transient*

Per modificare la dichiarazione di una variabile può essere usata la parola chiave ***transient***: questa indica al compilatore di non rappresentarla come parte dello stream di byte della versione serializzata dell'oggetto.

Ad esempio, si supponga che un oggetto contenga la seguente dichiarazione:

```
private transient String password
```

Questa variabile, quando l'oggetto che la contiene viene serializzato, non viene inclusa nella rappresentazione.

Riferimenti bibliografici

La parte sulla serializzazione degli oggetti è presa da:

J. Lewis, W. Loftus.

Java: Fondamenti di progettazione software (prima edizione italiana).

Addison-Wesley, 2001

È possibile anche visitare il sito Web degli autori all'indirizzo:

http://jss.villanova.edu