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.

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.

InputStream…

Il compito della classe astratta *InputStream* è di descrivere i metodi per classi che producono input da diverse sorgenti in forma di stream (per esempio, un oggetto String, un flusso di byte, …)

Le diverse tipologie di sorgenti hanno una sottoclasse di *InputStream* associata.

InputStream…

Classe	Funzione
ByteArrayInputStream	Permette ad un buffer in memoria di essere usato come un InputStream
StringBufferInputStream	Permette ad un oggetto String di essere trattato come un InputStream
FileInputStream	Per la lettura di informazioni da un file su disco
PipedInputStream	Implementa il concetto di 'piping' (usato nel multithreading)
SequenceInputStream	Converte due o più oggetti InputStream in un singolo InputStream.
FilterInputStream	Classe astratta che fornisce utili funzionalità alle altre classi di InputStream

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

Tipi di FilterInputStream:

DataInputStream	Si occupa della lettura dei tipi primitivi dallo st (int, float…)
BufferedInputStream	Usata per evitare la lettura fisica dei dati qualvolta che ce n'è bisogno. Si ottiene ciò mec un buffer
LineNumberInputStream	Tiene traccia dei numeri di linea dello strea input (ad es. getLineNumber())
PushbackInputStream	Permette di tornare indietro, al carattere precedentemente, attraverso un buffer (usato pe scanner)

OutputStream

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

Classi	Funzioni
ByteArrayOutputStream	Crea un buffer in memoria. Tutti i dati inviati sullo stream sono inseriti nel buffer
FileOutputStream	Per l'invio di informazioni verso un file
PipedOutputStream	Implementa il concetto di 'piping' (usato nel multithreading)
FilterOutputStream	Classe astratta che fornisce utilità per le classi di OutputStream

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

DataOutputStream	formatta ogni tipo primitivo e gli oggetti String in uno stream che possa essere letto da ogni oggetto DataInputStream di qualsiasi macchina;
PrintStream	stampa tutti i dati di tipo primitivo e gli oggetti di tipo String in un formato leggibile. Due metodi molto importanti di tale classe sono print() e println(). Questa classe si occupa anche di catturare tutte le eccezioni di I/O;
BufferedOutputStream	specifica che lo stream può essere usato come buffer (si usa per i file o per i dispositivi I/O)

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.

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*.

…Standard I/O…

```
// come leggere dallo standard input
import java.io.*;
public class Echo {
public static void main(String[] args) throws IOException {
      BufferedReader in = new BufferedReader( new InputStreamReader(System.in));
      String s;
      while((s = in.readLine()).length() != 0)
      System.out.println(s);
      // Una linea vuota termina il programma
}
}
```

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.

Ridirezionare gli standard di I/O

```
import java.io.*;
class Redirecting {
public static void main(String[] args) throws IOException {
        BufferedInputStream in =new BufferedInputStream(new FileInputStream(
                "Redirecting.java"));
        PrintStream out = new PrintStream( new BufferedOutputStream( new
        FileOutputStream("test.out")));
        System.setIn(in);
        System.setOut(out);
        System.setErr(out);
        BufferedReader br =new BufferedReader(new InputStreamReader(System.in));
        String s;
        while((s = br.readLine()) != null)
                System.out.println(s);
        out.close(); // Ricordatene!
        }
}
```

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