Input/Output

Input/Output

Gli archivi di dati sono rappresentati da un insieme di informazioni registrate su un supporto di memorizzazione secondaria, quale può essere un disco o un nastro. Le informazioni presenti nella memoria secondaria sono indicate con il termine generico di file.

Si hanno due principali vantaggi nel memorizzare i dati all'interno di file. Il primo è la persistenza dei dati, in quanto non vengono persi quando si chiude l'applicazione oppure si spegne il computer

Il secondo vantaggio è rappresentato dalla grande quantità di dati che può essere memorizzata in un file: le memorie di massa hanno dimensioni che superano quelle della memoria principale e sono utili per gestire grandi quantità di informazioni

Le operazioni fondamentali riguardanti il trattamento di un file sono:

- apertura del file: il sistema operativo riserva una porzione di memoria per il file e ne restituisce un puntatore
- lettura del file: il programma legge il contenuto del file
- scrittura del file: il programma scrive sul file
- chiusura del file: il sistema operativo libera la memoria precedentemente riservata

> La lettura si indica anche con input e la scrittura con output. Le operazioni che trasferiscono informazioni dalla memoria centrale al file e viceversa si chiamano, in generale, operazioni di I/O (Input/Output) sul file.

Java gestisce tutte le operazioni di I/O, e quindi anche i file, con il concetto di stream, cioè di flusso di dati. Gli stream costituiscono in pratica sequenze ordinate di dati e si dividono in:

- stream di input: permettono di leggere i dati da un file
- stream di output: permettono di scrivere i dati su un file

La sorgente di uno stream di input può essere un file, la tastiera o un altro dispositivo di input. La destinazione di uno stream di output può essere un file, lo schermo o un altro dispositivo di output. Gli stream sono utilizzati anche per la comunicazione in internet.

Gli stream vengono creati in Java istanziando le classi opportune contenute nel package `java.io`. È possibile dividere queste classi in due categorie principali:

- Classi basate su byte: sono classi che leggono e scrivono byte. Sono utilizzate per leggere e scrivere dati binari, come ad esempio immagini, audio e video. Le classi principali di questo tipo sono `InputStream` e `OutputStream`.
- Classi basate su caratteri: sono classi che leggono e scrivono caratteri. Sono utilizzate per leggere e scrivere dati testuali. Le classi principali di questo tipo sono `Reader` e `Writer`.

Tipi di file

I file possono essere di due tipi:

- file strutturati: sono file che contengono dati strutturati, per interpretare il contenuto di un file strutturato se ne deve conoscere la struttura. Ad esempio se un file contiene un numero intero, una stringa e un numero reale, per interpretare il contenuto del file è necessario sapere che il primo dato è un intero, il secondo è una stringa e il terzo è un numero reale.
- file di testo: sono file che contengono dati testuali

File strutturati

L'apertura di un file strutturato per le operazioni di output, cioè per la scrittura, viene eseguita con le dichiarazioni dei seguenti oggetti:

```
FileOutputStream fos = new FileOutputStream("file.dat");
ObjectOutputStream oos = new ObjectOutputStream(fos);
```

La prima riga crea uno stream `fos` di output verso il file `file.dat`. La seconda riga crea uno stream `oos` di output verso lo stream `fos`. Lo stream `oos` è in grado di scrivere oggetti sul file `file.dat`.

Usando il primo comando, se il file `file.dat` non esiste, viene creato. Se il file esiste già, viene sovrascritto. Per aprire un file per le operazioni di input, cioè per la lettura, si utilizzano le seguenti dichiarazioni:

```
FileInputStream fis = new FileInputStream("file.dat");
ObjectInputStream ois = new ObjectInputStream(fis);
```

Se invece vogliamo accodare i dati al file, senza sovrascriverlo, dobbiamo utilizzare le seguenti dichiarazioni:

```
FileOutputStream fos = new FileOutputStream("file.dat", true);
ObjectOutputStream oos = new ObjectOutputStream(fos);
```

Il secondo parametro del costruttore di `FileOutputStream` indica se il file deve essere sovrascritto (`false`) o se i dati devono essere accodati al file (`true`).

Una volta finito di utilizzare gli stream, è necessario chiuderli. Per chiudere uno stream si utilizza il metodo `close()`. Ad esempio, per chiudere lo stream `oos` si utilizza il comando:

```
oos.close();
```

Eccezioni

Le operazioni che gestiscono l'I/O possono generare diverse eccezioni, per esempio quando il metodo close riscontra una situazione anomala, oppure quando l'apertura di uno stream di input specifica un nome di file non esistente. Per questi motivi si devono racchiudere tutte le istruzioni che gestiscono i file in un blocco `try...catch` che cattura le eccezioni

```
FileInputStream fis = null;
try {
  fis = new FileInputStream("file.dat");
  ObjectInputStream ois = new ObjectInputStream(fis);
  // altre istruzioni
  ois.close();
} catch (IOException e) {
  System.out.println("Errore di I/O" + e.getMessage());
}
```

Dopo l'apertura dello stream di output si possono utilizzare diversi metodi per la scrittura. Ogni metodo memorizza sul file un particolare tipo di dato e assume la forma writeDato. Al posto di Dato si sostituisce il tipo di dato che si vuole memorizzare.

Ad esempio, per memorizzare un intero si utilizza il metodo `writeInt`:

```
oos.writeInt(10);
```

Prima di chiudere il file è importante eseguire il metodo `flush`, che serve per scrivere su disco tutti i dati che sono attualmente contenuti nel buffer dello stream.

Scrittura di un file strutturato

Il metodo di scrittura più importante è `writeObject`. Con questo metodo è possibile salvare su di un file anche gli oggetti: è Java che si preoccupa di salvare la struttura dell'oggetto, memorizzando tutti i suoi attributi non statici. Un oggetto salvato su disco diventa persistente, cioè sopravvive alla singola esecuzione del programma e può essere richiamato in una successiva esecuzione. Una classe le cui istanze si vogliono rendere persistenti deve implementare l'interfaccia `Serializable`.

```
public class Persona implements Serializable {
  private String nome;
  private String cognome;
  private int eta;
  // costruttore e metodi
}
```

```java
import java.io.*;

public class ProvaFile {
  public static void main(String[] args) {
    Persona p1, p2, p3;
    p1 = new Persona("Dante", "Dua", 30);
    p2 = new Persona("Antonio", "Ragazzo", 24);
    p3 = new Persona("Arturo", "Cinque" 57);
    try {
      FileOutputStream fos = new FileOutputStream("persone.dat");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(p1);
      oos.writeObject(p2);
      oos.writeObject(p3);
      oos.flush();
      oos.close();
    } catch (IOException e) {
      System.out.println("Errore di I/O" + e.getMessage());
    }
  }
}
```

Lettura di un file strutturato

Per leggere le informazioni contenute in un file strutturato si usano i metodi della classe `ObjectInputStream`. Questi metodi assumono la forma readDato, dove al posto di Dato si sostituisce il tipo di dato che si vuole leggere. Per evitare di ottenere inconsistenza, i dati devono essere letti nello stesso ordine in cui sono stati salvati.

Tra i metodi di lettura c'è il metodo readObject, che consente di recuperare un oggetto precedentemente salvato. Questo metodo restituisce un oggetto di classe Object che attraverso il casting può essere riportato alla sua classe originaria. La lettura di un oggetto comporta la creazione di una nuova istanza della classe, in cui a ogni attributo viene assegnato il valore letto dal file.

Se non si conosce quanti sono i dati contenuti nel file, si può usare un ciclo infinito per continuare a leggere finchè viene generata l'eccezione `EOFException`. Questa eccezione segnala che si è raggiunta la fine del file e non ci sono più dati da leggere; si può quindi interrompere il ciclo.

```java
import java.io.*;

public class ProvaFile {
  public static void main(String[] args) {
    try {
      FileInputStream fis = new FileInputStream("persone.dat");
      ObjectInputStream ois = new ObjectInputStream(fis);
      while (true) {
        Persona p = (Persona) ois.readObject();
        System.out.println(p);
      }
    } catch (EOFException e) {
      System.out.println("Fine file");
    } catch (IOException e) {
      System.out.println("Errore di I/O" + e.getMessage());
    } catch (ClassNotFoundException e) {
      System.out.println("Classe non trovata" + e.getMessage());
    }
  }
}
```

File di testo

I file di testo sono file che contengono dati testuali. Per leggere e scrivere file di testo si utilizzano le classi `FileReader` e `FileWriter`. Queste classi sono derivate dalle classi `Reader` e `Writer` e quindi ereditano tutti i metodi di queste classi.

```
FileReader fr = new FileReader("file.txt");
BufferedReader br = new BufferedReader(fr);
```

```
FileWriter fw = new FileWriter("file.txt");
PrintWriter pw = new PrintWriter(fw);
```

Lettura di un file di testo

Per leggere un file di testo si utilizza il metodo `readLine` della classe `BufferedReader`. Questo metodo legge una riga di testo dal file e la restituisce sotto forma di stringa. Se il file è finito, il metodo restituisce il valore `null`.

```
String s = br.readLine();
```

Per leggere tutto il file si può utilizzare un ciclo infinito che termina quando il metodo `readLine` restituisce `null`.

```
String s = br.readLine();
while (s ≠ null) {
  System.out.println(s);
  s = br.readLine();
}
```

Per leggere un file di testo carattere per carattere si utilizza il metodo `read` della classe `FileReader`. Questo metodo legge un carattere dal file e lo restituisce sotto forma di intero. Se il file è finito, il metodo restituisce il valore -1.

```
int c = br.read();
```