19 - Input/Output su File

Programmazione e analisi di dati
Modulo A: Programmazione in Java

Paolo Milazzo

Dipartimento di Informatica, Università di Pisa
http://www.di.unipi.it/milazzo
milazzo@di.unipi.it

Corso di Laurea Magistrale in Informatica Umanistica
A.A. 2019/2020

Persistenza dei dati

Nei programmi visti fino ad ora abbiamo usato

- Output su schermo (tramite `System.out.println()`)
- Input da tastiera (tramite la classe `Scanner`)

Con queste modalità di Input/Output, al termine dell'esecuzione di un programma tutti i dati inseriti tramite tastiera vengono persi

Spesso è utile fornire funzionalità di persistenza dei dati

- i dati devono poter sopravvivere alla terminazione del programma ed essere disponibili in una successiva nuova esecuzione
- bisogna salvare i dati in un file

Stream (1)

In Java, l'Input/Output su file segue il modello degli stream (o flussi)

Uno stream di input prevede una sorgente di dati (es. un file) che possono essere utilizzati da un programma

Uno stream di output prevede una destinazione per i dati (es. un file) che sono generati da un programma

Stream (2)

Volendo fare un'analogia:

- Uno stream di input è come l'acquedotto pubblico, che porta l'acqua da una sorgente al rubinetto di casa
- Uno stream di output è come il sistema fognario, che porta l'acqua dal rubinetto di casa al depuratore

Quando si apre un rubinetto, l'acqua continua a uscire fintanto che:

- non chiudiamo il rubinetto
- non si esaurisce l'acqua della sorgente

Analogamente, in uno stream di input si continua a ricevere dati fintanto che:

- non si decide di smettere di leggerli
- la sorgente dei dati si esaurisce (es. il file è finito)

Stream (3)

In Java, le classi di libreria per la gestione degli stream si dividono in due gruppi:

- per la gestione di stream di caratteri
- per la gestione di stream di bytes

Nel caso dei file:

- le classi del primo gruppo consentono di leggere/scrivere file di testo
- le classi del secondo gruppo consentono di leggere/scrivere file binari (immagini, video, dati, ecc...)

I file di testo possono essere modificati con un qualunque programma di scrittura, i file binari no!

Stream basici (1)

Le classi di base per la gestione degli stream consentono di leggere/scrivere

- singoli caratteri
- singoli bytes

Tali classi sono le seguenti:

- Stream di caratteri: `FileReader` e `FileWriter`
- Stream di bytes: `FileInputStream` e `FileOutputStream`

Queste classi (come le classi che vedremo nel seguito) fanno parte del package `java.io`

- devono essere importate nei programmi!

Stream basici (2)

Vediamo il funzionamento degli stream di caratteri

- gli stream di bytes funzionano in maniera analoga

La classe `FileReader` consente di leggere un file un carattere alla volta

Per fare ciò dobbiamo:

1. passare al costruttore di `FileReader` il nome del file (come stringa)
2. chiamare ripetutatmente il metodo `read()` che legge un carattere
3. chiudere lo stream invocando il metodo `close()`

Stream basici (2)

Più in dettaglio:

Chiamata del costruttore:

```
FileReader reader = new FileReader("prova.txt");
```

- predispone il file per la lettura (interagendo con il sistema operativo del computer)
- lancia un'eccezione se il file non esiste

Lettura dei singoli caratteri:

```
int next = reader.read();
```

- restituisce -1 se il file è terminato
- altrimenti l'intero può essere castato a `char` e usato nel programma

Chiusura dello stream:

```
reader.close();
```

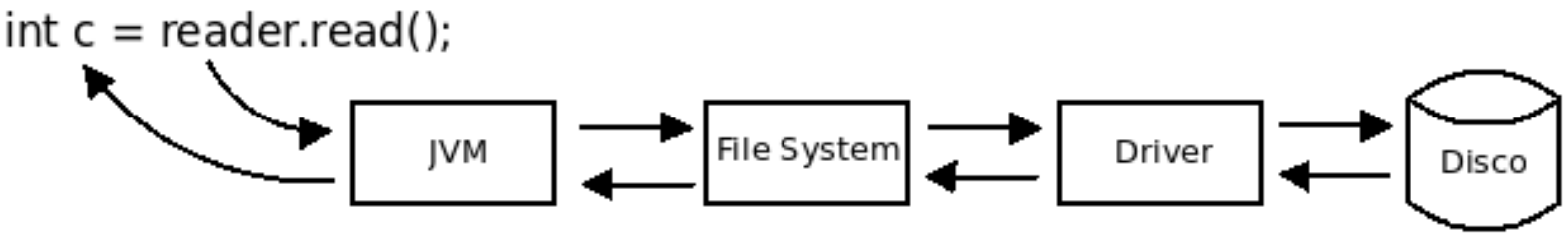
- libera la risorsa: comunica al sistema operativo del computer che il file può essere usato da altri programmi

Stream basici (3)

Le operazioni sugli stream coinvolgono pesantemente il sistema operativo del computer, e non solo!

Ad esempio: quando si esegue una `read()`

- La Java Virtual Machine fa la richiesta del carattere al file system (componente del sistema operativo che gestisce i file)
- Il file system a sua volta fa una opportuna richiesta al driver del disposivo di memorizzazione relativo (disco fisso, chiavetta USB,...)
- Il driver invia opportuni segnali al dispositivo di memorizzazione stesso
- Il dispositivo di memorizzazione recupera il dato e lo rimanda indetro ai richiedenti



Stream basici (4)

Molti componenti esterni alla Java Virtual Machine sono coinvolti

- è relativamente probabile che qualcosa vada storto...
- per tutelarsi, Java utilizza le eccezioni di tipo `IOException`
- tali eccezioni:
  - possono essere lanciate da tutti i metodi delle classi di gestione degli stream
  - sono controllate: ossia, devono essere gestite da chi chiama i metodi

Esempio: Lettura di un file

```
import java.io.FileReader;
import java.io.IOException;

public class LeggiFile {

  public static void main(String[] args) {
    try {
      int next;
      // crea lo stream di input
      FileReader reader = new FileReader("prova.txt");
        do {
          // legge i caratteri
          next = reader.read();
          if (next!=-1) {
            // casta l'intero a char e lo visualizza
            char c = (char) next;
            System.out.print(c);
          }
        } while (next!=-1); // finche il file non finisce
        // chiude lo stream
        reader.close();
    }
    catch (IOException e) {
      // in caso di errori in una delle operazioni...
      System.out.println("ERRORE di I/O");
      System.out.println(e);
    }
  }
}
```

Stream basici (5)

La scrittura su file tramite uno stream di output è analoga alla lettura:

- Si utilizza la classe `FileWriter`
- Al posto di `read()` si usa il metodo `write()`

Esempio: Copia di un file (1)

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopiaFile {
  public static void main(String[] args) {

    // questo non serve... lo uso solo per calcolare
    // il tempo di esecuzione del programma
    long inizio = System.currentTimeMillis();

    try {
      int next;

      // crea entrambi gli stream (input e output)
      FileReader reader = new FileReader("prova.txt");
      FileWriter writer = new FileWriter("prova2.txt");
      do {
        // legge un carattere dallo stream di input...
        next = reader.read();
        if (next!=-1) {
          char c = (char) next;
          // e lo scrive nello stream di output...
          writer.write(c);
        }
      } while (next!=-1); // finche' il file non finisce

(segue)
```

Esempio: Copia di un file (2)

```
(segue CopiaFile)

      // chiude entrambi gli stream
      reader.close();
      writer.close(); // provare a commentare questo!!
    }
    catch (IOException e) { // in caso di errori...
      System.out.println("ERRORE di I/O");
      System.out.println(e);
    }

    // quanto tempo ha impiegato?
    // (anche questo in realta' non serve...)
    long fine = System.currentTimeMillis();
    System.out.println("Impiegati "+(fine-inizio)+" millisecondi");

  }
}
```

Input/Output bufferizzato (1)

Leggere o scrivere un carattere per volta richiede tempo...

- per ogni carattere bisogna ripetere tutti i passaggi con il sistema operativo descritti prima
- il programma `CopiaFile` richiede diversi millisecondi (che non è poco) anche per copiare file di piccola dimensione...

La soluzione a questo problema è l'utilizzo di un buffer

- Un buffer è una memoria tampone usata per immagazzinare dati da usare in seguito
- Ritornando all'analogia dell'acquedotto, un buffer è simile a:
  - Una riserva d'acqua (nei posti a rischio di siccità) [input]
  - Un boiler (mantiene un po' di acqua calda pronta per l'uso) [input]
  - Una fossa biologica (raccoglie i liquami prodotti) [output]

Input/Output bufferizzato (2)

Uno stream bufferizzato lo si ottiene utilizzando uno stream basico all'interno di una opportuna classe

Di nuovo, le classi per la creazione di stream bufferizzati si differenziano a seconda del tipo di stream:

- Stream di caratteri: `BufferedReader` e `BufferedWriter`
- Stream di bytes: `BufferedInputStream` e `BufferedOutputStream`

Vediamo ancora il caso dello stream di caratteri (il caso dei bytes è analogo)

Input/Output bufferizzato (3)

Per usare uno stream di input bufferizzato (l'output è analogo) dobbiamo:

1. passare al costruttore di `BufferedReader` uno stream basico (di tipo `FileReader`)
2. chiamare ripetutatmente il metodo `read()` che legge un carattere
3. chiudere lo stream invocando il metodo `close()`

Rispetto allo stream basico cambia solo la chiamata al costruttore

- Il `BufferedReader` viene collegato a un `FileReader`...
- ... come una riserva idrica viene collegata a un rubinetto dell'acqua

Il `FileReader` spesso viene creato al momento

```
BufferedReader reader =
      new BufferedReader(new FileReader("prova.txt"));
```

Input/Output bufferizzato (4)

Continueremo a leggere un carattere per volta, ma di nascosto il BufferedReader si porterà avanti con il lavoro:

- quando si esegue una `read()` vengono richiesti al sistema operativo un numero di caratteri pari alla dimensione del buffer
- successive chiamate a `read()` verranno subito risolte dal buffer senza interpellare il sistema operativo

Esempio: Copia di un file (1)

```
// NOTA: cosi' importa tutte le classi del package java.io
import java.io.*;

public class CopiaFileBufferizzato {
  public static void main(String[] args) {

    // questo non serve... lo uso solo per calcolare
    // il tempo di esecuzione del programma
    long inizio = System.currentTimeMillis();

    try {
      int next;
      // crea entrambi gli stream bufferizzati
      BufferedReader reader =
          new BufferedReader(new FileReader("prova.txt"));
      BufferedWriter writer =
          new BufferedWriter(new FileWriter("prova2.txt"));
        do {
          // legge un carattere dallo stream di input...
          next = reader.read();
          if (next!=-1) {
            char c = (char) next;
            // e lo scrive nello stream di output...
            writer.write(c);
          }
        } while (next!=-1); // finche' il file non finisce

(segue)
```

Esempio: Copia di un file (2)

```
(segue CopiaFileBufferizzato)

        // chiude entrambi gli stream
        reader.close();
        writer.close();
    }
    catch (IOException e) { // in caso di errori...
      System.out.println("ERRORE di I/O");
      System.out.println(e);
    }

    // quanto tempo ha impiegato?
    // (anche questo in realta' non serve...)
    long fine = System.currentTimeMillis();
    System.out.println("Impiegati "+(fine-inizio)+" millisecondi");

  }
}
```

Input/Output formattato (1)

Leggere singoli caratteri non è il massimo...

- esistono classi che si occupano di dare una forma ai dati letti/scritti tramite uno stream

Nel caso degli stream di caratteri tali classi di formattazione sono

- `Scanner` per formattare i dati di uno stream di input
- `PrintWriter` per formattare i dati di uno stream di output

Sono in realtà sono classi analoghe a quelle già usate per input/output tramite tastiera/video!

- `Scanner` fornisce `nextInt()`, `nextLine()`, ecc..
- `PrintWriter` fornisce `println()` e `print()`

Input/Output formattato (2)

Come usare `Scanner` con i file:

1. passare al costruttore di `Scanner` uno stream di caratteri di input (meglio se bufferizzato, `BufferedReader`)
2. usare i metodi `nextInt()`, `nextLine()`, ecc... per leggere dal file facendo attezione
   - bisogna essere sicuri del tipo di valore che si legge con questi metodi
   - o, in alternativa, bisogna essere pronti a gestire l'eccezione lanciata in caso di input di tipo sbagliato
3. chiudere lo `Scanner`

Come usare `PrintWriter` con i file:

1. passare al costruttore di `PrintWriter` uno stream di caratteri di output (meglio se bufferizzato, `BufferedWriter`)
2. usare i metodi `println()` e `print()` per scrivere nel file
3. chiudere il `PrintWriter`

Esempio: registro delle eta' (1)

```
// NOTA: cosi' importo tutte le classi del package java.io
import java.io.*;
import java.util.Scanner;
import java.util.Vector;

public class RegistroEtaMinimale {

  public static void main(String[] args) {

    // memorizza nomi ed eta in due vettori "paralleli": al nome
    // in posizione i corrisponde l'eta' in posizione i
    Vector<String> nomi = new Vector<String>();
    Vector<Integer> eta = new Vector<Integer>();

    try {
      // crea lo stream (formattato) di input
      Scanner file_input =
          new Scanner(new BufferedReader(
              new FileReader("registro.txt")));

      // legge tutto il file, aggiungendo i valori letti ai vettori
      while (file_input.hasNext()) {
        // nota: nome deve essere una singola parola (usa next)
        nomi.add(file_input.next());
        eta.add(file_input.nextInt());
      }

(segue)
```

Esempio: registro delle eta' (2)

```
(segue RegistroEtaMinimale)

      // chiude lo stream
      file_input.close();
    }
    catch (IOException e) { // in caso di errori...
      System.out.println("ERRORE di I/O");
      System.out.println(e);
    }

    // stampa il contenuto del registro (i vettori)
    System.out.println("REGISTRO DELLE ETA'");
    for (int i=0; i<nomi.size(); i++) {
      System.out.println(nomi.get(i) + " " + eta.get(i));
    }

    System.out.println();

(segue)
```

Esempio: registro delle eta' (3)

```
(segue RegistroEtaMinimale)

    // consente di incrementare (di uno) l'eta' di alcune
    // persone nel registro
    boolean finito = false;
    do {
      // legge un nome
      Scanner input = new Scanner(System.in);
      System.out.println("Inserisci un nome per incrementare l'eta");
      System.out.println("oppure invio per terminare:");
      String n = input.nextLine();

      // se l'utente ha premuto invio si ferma
      if (n.equals("")) finito=true;
      else {
        // altrimenti incrementa l'eta' corrisp. al nome (se presente)
        int j=nomi.indexOf(n);
        if (j==-1)
          System.out.println("Nome "+n+" non presente nel registro");
        else {
          int e = eta.get(j);
          eta.set(j,e+1);
          System.out.println("Ora " +n+ " ha " + (e+1) + " anni");
        }
      }
    } while (!finito);

(segue)
```

Esempio: registro delle eta' (4)

```
(segue RegistroEtaMinimale)

    // quando l'utente ha terminato, salva tutto nel file
    try {
      // crea lo stream (formattato) di output
      PrintWriter file_output =
          new PrintWriter(new BufferedWriter(
              new FileWriter("registro.txt")));

      // scrive i valori usando println
      for (int i=0; i<nomi.size(); i++) {
        file_output.println(nomi.get(i) + " " + eta.get(i));
      }

      // chiude lo stream di output
      file_output.close();
    }
    catch (IOException e) { // in caso di errori...
      System.out.println("ERRORE di I/O");
      System.out.println(e);
    }
  }
}
```

Serializzazione degli oggetti (1)

Sebbene le classi `Scanner` e `PrintWriter` aiutino a salvare dati di vario tipo su file, il loro uso non è sempre facile

Quando abbiamo un programma complesso che usa molte strutture dati, salvare lo stato del programma in un file può diventare complicato

In questi casi, l'implementazione di meccanismi di persistenza dei dati può essere semplificata tramite

- la serializzazione degli oggetti

La serializzazione degli oggetti è una funzionalità di Java che consente di rappresentare oggetti arbitrariamente complessi come sequenze di byte ben definite

- tali sequenze di byte potranno essere salvate su file tramite uno stream (di byte, questa volta)

Serializzazione degli oggetti (2)

Per leggere/scrivere un oggetto in un file bisogna usare le classi:

- `ObjectInputStream` e `ObjectOutputStream`

Nel caso di `ObjectInputStream` (l'output è analogo) si procede come segue:

1. passare al costruttore di `ObjectInputStream` uno stream di bytes di input (meglio se bufferizzato, `BufferedInputStream`)
2. usare il metodo `readObject()` per leggere dal file facendo attenzione
   - se il file non contiene un oggetto viene lanciata l'eccezione `ClassNotFoundException`
3. chiudere lo stream

Esempio: salvare un vettore (1)

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

public class SerializzaVettore {

  public static void main(String[] args) {

    // crea e inizializza un vettore
    Vector<String> v1 = new Vector<String>();
    v1.add("AAA");
    v1.add("BBB");
    v1.add("CCC");

    // lo salva in un file
    try {
      // notare le classi degli stream di byte
      ObjectOutputStream out =
              new ObjectOutputStream(new BufferedOutputStream(
                      new FileOutputStream("serializza.dat")));
      out.writeObject(v1);
      out.close();
    } catch(IOException e) {
      System.out.println("ERRORE di I/O");
      System.out.println(e);
    }

(segue)
```

Esempio: salvare un vettore (2)

```
(segue SerializzaVettore)

    // lo rilegge dal file (assegnandolo a un altra variabile)
    Vector<String> v2 = null;
    try {
    // notare le classi degli stream di byte
      ObjectInputStream in =
              new ObjectInputStream(new BufferedInputStream(
                      new FileInputStream("serializza.dat")));
      v2 = (Vector<String>) in.readObject();
      in.close();
    } catch (ClassNotFoundException e) {
      // se il file non contiene un oggetto....
      System.out.println("PROBLEMA (manca oggetto nel file)");
      System.out.println(e);
    } catch(IOException e) {
      System.out.println("ERRORE di I/O");
      System.out.println(e);
    }

    // stampa quanto letto dal file
    System.out.println(v2);
  }
}
```

Serializzazione degli oggetti (3)

Affinché un oggetto possa essere serializzato e salvato su file, la sua classe deve implementare l'interfaccia `Serializable`

Tale interfaccia non prevede metodi

- questo meccanismo serve solo per far dichiarare al programmatore che autorizza il salvataggio dei suoi oggetti
- e' un meccanismo di sicurezza: la serializzazione mette a nudo (nel file) il contenuto delle variabili private di un oggetto!
- un programmatore malevolo potrebbe accedere alle variabili private di un oggetto salvandolo su file e andando a leggere il file un byte alla volta....

Serializzazione degli oggetti (4)

Tutte le principali classi della Libreria di Java sono serializzabili (implementano `Serializable`)

- ovviamente, devono essere serializzabili anche le classi degli oggetti in esse contenute
- ad esempio:
  - La classe `String` implementa `Serializable`, quindi un vettore di stringhe è ok
  - Un vettore di elementi di un tipo `Rettangolo` definito da noi è serializzabile solo se `Rettangolo` lo è

Serializzazione degli oggetti (5)

Alle classi che implementano `Serializable` viene richiesto (anche se non è obbligatorio) di includere una costante statica `serialVersionUID` (di tipo `long`)

- servirebbe per distinguere tra diverse versioni della stessa classe (es. modifiche successive)

Rendiamo la classe `Rettangolo` serializzabile:

```
public class Rettangolo implements Serializable {
   static final long serialVersionUID = 1;

   public int base;
   public int altezza;

   public Rettangolo(int base, int altezza) {
     this.base = base;
     this.altezza = altezza;
   }
}
```

Serializzazione degli oggetti (6)

In questo modo possiamo salvare su file un vettore di rettangoli:

```
....

  Vector<Rettangolo> v = new Vector<Rettangolo>();
  v.add(new Rettangolo(10,10));
  v.add(new Rettangolo(15,50));
  v.add(new Rettangolo(20,60));

  try {
    ObjectOutputStream out =
            new ObjectOutputStream(new BufferedOutputStream(
                    new FileOutputStream("rettangoli.dat")));
    out.writeObject(v);
    out.close();
  } catch(IOException e) {
    System.out.println("ERRORE di I/O");
    System.out.println(e);
  }

....
```

Serializzazione degli oggetti (7)

Un esempio più completo di programma che utilizza le serializzazione degli oggetti per implementare un meccanismo di persistenza dei dati è il programma `RegistroEtaCompleto` (vedere la pagina web del corso)

Tale programma prevede:

- una classe `NomeEta` che associa un nome a una età
- una classe `RegistroEtaCompleto` che contiene un vettore di oggetti `NomeEta` e diverse funzionalità
- una classe `GestioneRegistroEta` che contiene un menu per accedere alle varie funzionalità di `RegistroEtaCompleto`