JAVA – Input/Output

Metodi Avanzati di Programmazione

Laurea Triennale in Informatica

Università degli Studi di Bari Aldo Moro

Docente: Pierpaolo Basile

I/O Stream

- Un flusso di I/O rappresenta una sorgente di input o una destinazione di output
  - Un flusso può rappresentare molti tipi diversi: file su disco, dispositivi, altri programmi e array in memoria
- Gli stream supportano molti tipi diversi di dati, inclusi byte semplici, tipi di dati primitivi, caratteri, oggetti
  - Alcuni flussi semplicemente trasmettono dati; altri manipolano e trasformano i dati
- Indipendentemente dal modo in cui funzionano internamente, tutti i flussi presentano lo stesso modello: **una sequenza di dati**

Byte stream

- I byte stream sono utilizzati per leggere e scrivere byte (8 bit) su un dispositivo di I/O
- Ereditano dalle classi InputStream e OutputStream
- Ci sono diverse classi che eredito dai byte stream per leggere/scrivere su file
  - FileInputStream
  - FileOutputStream

Esempio (copia byte)…

```
public class CopyBytes {
    public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("sorgente.txt");
            out = new FileOutputStream("destinazione.txt");
            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
```

Esempio (copia byte)…

```
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Esempio (copia byte)…note

- **Chiudere** gli stream è importante per non perdere le informazioni scritte sullo stream e per liberare risorse
- La scrittura dei singoli byte andrebbe evitata
  - è un'operazione di basso livello
  - esistono degli stream **idonei** per scrivere/leggere caratteri

Character stream

- Da utilizzare in caso di I/O di caratteri
- Gestiscono automaticamente la codifica corretta per i caratteri
  - ISO, UTF, …
- Character stream su file
  - FileReader, FileWriter

Character stream (copia file)

```
FileReader inputStream = null;
FileWriter outputStream = null;

try {
    inputStream = new FileReader("sorgente.txt");
    outputStream = newFileWriter("destinazione.txt");

    int c;
    while ((c = inputStream.read()) != -1) {
        outputStream.write(c);
    }
} finally {…
```

Line-oriented I/O

- In genere i file contenenti testo contengono molto più dei singoli caratteri
- E’ richiesto l’accesso a sequenze di caratteri (stringhe)
  - Lettura di intere linee di testo
- BufferedReader, PrintWriter

Line-oriented I/O (copia file)

```
 BufferedReader inputStream = null;
 PrintWriter outputStream = null;
 try {
     inputStream = new BufferedReader(new
FileReader("sorgente.txt"));
     outputStream = new PrintWriter(new
FileWriter("destinazione.txt"));

     String l;
     while ((l = inputStream.readLine()) != null) {
           outputStream.println(l);
      }
 } finally {…
```

Buffered I/O…

- La maggior parte delle classi che abbiamo visto sono unbeffered
  - le operazione vengono eseguite direttamente sul dispositivo di I/O
- Le classi buffered (BufferedReader e BufferedWriter) utilizzo un buffer
  - un buffer è un'area di memoria predisposta all'I/O che velocizza le operazioni di read/write

…Buffered

- **BufferedInputStream** e **BufferedOutputStream** creano le versioni buffered di un byte stream
- **BufferedReader** e **BufferedWriter** creano le versioni buffered di un character stream

```
inputStream = new BufferedReader(new FileReader(“pippo.txt"));
outputStream = new BufferedWriter(new FileWriter(“pluto.txt"));
```

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*.

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.

Esiste un'altra interfaccia simile che si chiama FileFilter in cui il metodo accept prende in input direttamente un oggetto di tipo File

…La classe *File*.

Si può usare la classe *File* anche per:

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

https://docs.oracle.com/javase/8/docs/api/java/io/File.html

Guardare la classe: `di.uniba.map.b.lab.io.EsempioFile`

I/O con l'utente

- Spesso un programma comporta delle operazioni di I/O con l'utente
- Vi è la necessità di formattare l'output in modo da renderlo comprensibile (`format`)
- Vi è la necessità di processare l'input dell'utente in modo da ottenere le singole informazioni necessarie

Scanning

- La classe Scanner permette di processare l'input
  - suddividere l'input in token
  - tradurre ogni token rispetto ad un tipo predefinito
- I token vengono suddivisi utilizzando i white space

Scanning di un file

```
  Scanner s = null;
  try {
    s = new Scanner(new BufferedReader(new
FileReader("input.txt")));
    while (s.hasNext()) {
        System.out.println(s.next());
    }
  } finally {
```

Può essere utilizzato un delimitatore diverso per separare i token
`s.useDelimiter(<espr. regolare>);`

Scanning di un file 2

```
  Scanner s = null;
  double sum = 0;
  try {
      s = new Scanner(new BufferedReader(new
FileReader("numbers.txt")));
      while (s.hasNext()) {
          if (s.hasNextDouble()) {
              sum += s.nextDouble();
          } else {
              s.next();
          }
      }
  } finally {
      s.close();
  }
  System.out.println(sum);
```

Siamo alla ricerca di un tipo specifico
Lo scanner supporta tutti i tipi semplici

I/O da linea di comando

- La classe System mette a disposizione tre stream collegati alla linea di comando
  - System.in: InputStream che legge l'input
    - System.in è un byte stream per utilizzarlo come character stream → `InputStreamReader cin = new InputStreamReader(System.in);`
  - System.out: PrintStream che stampa l'output
  - System.err: PrintStream che stampa i messaggi di errori

Scanning da linea di comando

```
 public static void main(String[] args) {
        Scanner scanner = new Scanner(new InputStreamReader(System.in));
        String s = "";
        while (scanner.hasNext()) {
            s = scanner.next();
            if (!s.equalsIgnoreCase("exit")) {
                System.out.println("Hai scritto: " + s);
            } else {
                System.out.println("Goodbye!");
                break;
            }
        }
        scanner.close();
}
```

Data Streams

- I flussi di dati (Data Streams) supportano l'I/O binario dei valori di dati primitivi (booleano, char, byte, short, int, long, float e double) nonché i valori String.
- Tutti i flussi di dati implementano l'interfaccia DataInput o l'interfaccia DataOutput
  - implementazioni più utilizzate di queste interfacce, DataInputStream e DataOutputStream
- L'esempio DataStreams mostra i flussi di dati scrivendo un set di record di dati e quindi rileggendoli. Ogni record è costituito da tre valori relativi ad un articolo:
  - Prezzo (double), quantità (int), descrizione (String)

Data Streams (esempio)

- Definizione delle costanti utili all'esempio

```
static final String dataFile = "invoicedata";

static final double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99
};
static final int[] units = { 12, 8, 13, 29, 50 };
static final String[] descs = {
    "Java T-shirt",
    "Java Mug",
    "Duke Juggling Dolls",
    "Java Pin",
    "Java Key Chain"
};
```

Data Streams (esempio)

- Creazione di un output data stream che utilizza un output stream bufferizzato su file

```
out = new DataOutputStream(new BufferedOutputStream(
                new FileOutputStream(dataFile)));
```

Data Streams (esempio)

- Scrittura dei dati
  - writeUTF ci garantisce che le String vengano salvate codificate con lo standard UTF

```
for (int i = 0; i < prices.length; i ++) {
    out.writeDouble(prices[i]);
    out.writeInt(units[i]);
    out.writeUTF(descs[i]);
}
```

Data Streams (lettura)

```
in = new DataInputStream(new
            BufferedInputStream(new FileInputStream(dataFile)));
double price;
int unit;
String desc;
double total = 0.0;
try {
    while (true) {
        price = in.readDouble();
        unit = in.readInt();
        desc = in.readUTF();
        System.out.format("You ordered %d" + " units of %s at $%.2f%n",
            unit, desc, price);
        total += unit * price;
    }
} catch (EOFException e) {
}
```

Date Streams (note)

- DataStreams rileva una condizione di fine file catturando una EOFException, anziché testare un valore restituito non valido
  - tutte le implementazioni dei metodi DataInput utilizzano IOException (end of stream)
- Ogni scrittura specializzata in DataStreams corrisponde esattamente alla lettura specializzata corrispondente
  - Spetta al programmatore assicurarsi che i tipi di output e i tipi di input siano abbinati correttamente
  - Il flusso di input è costituito da semplici dati binari, senza nulla che indichi il tipo dei singoli valori o dove iniziano nel flusso

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: DataStream
- 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.

```
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+": "+nroNave;
    }
    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*

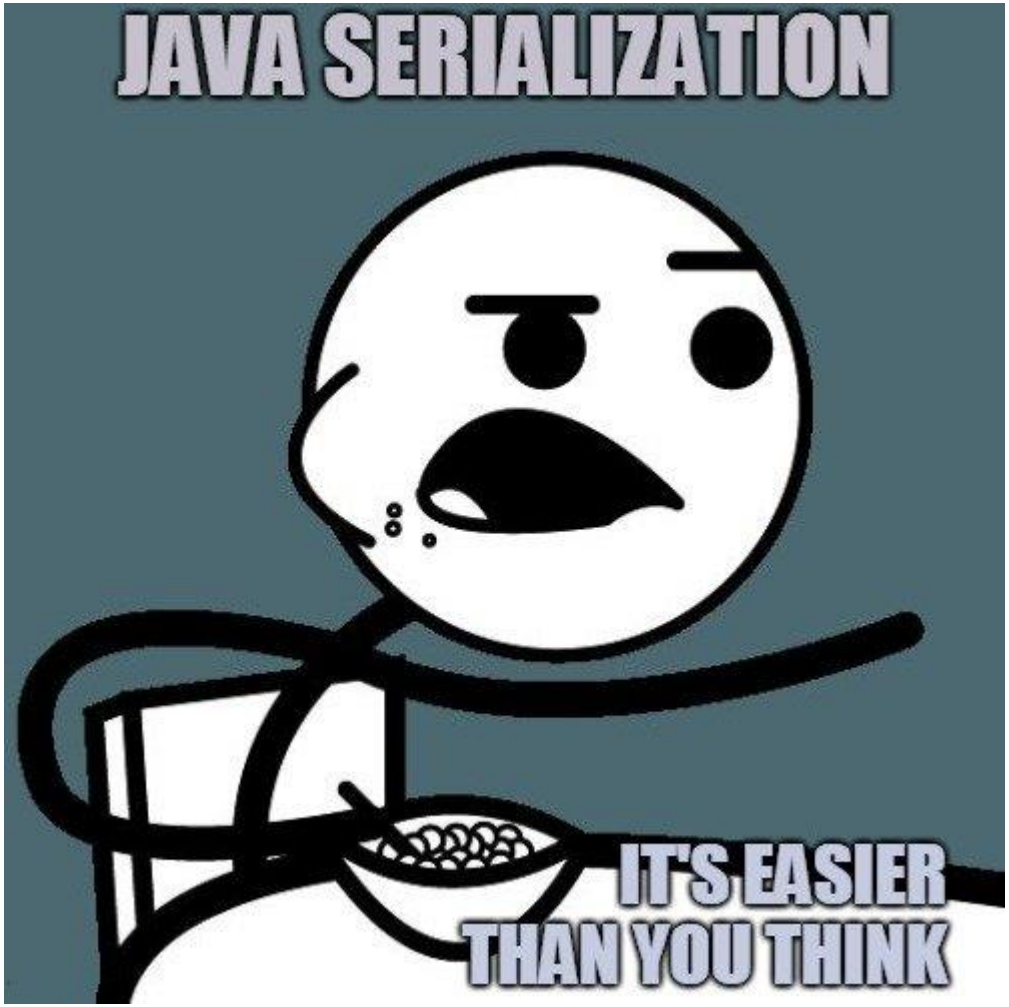
- A volte, quando si serializza un oggetto, si può desiderare di escludere delle informazioni, ad esempio, una password
  - Questo accade quando le informazioni vengono trasmesse via rete
  - Il pericolo è che, pur dichiarandola con visibilità privata, la password possa essere letta e usata da soggetti 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


JAVA SERIALIZATION
IT'S EASIER
THAN YOU THINK