Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Gestione dell'I/O in Java
Parte 1: I/O binario

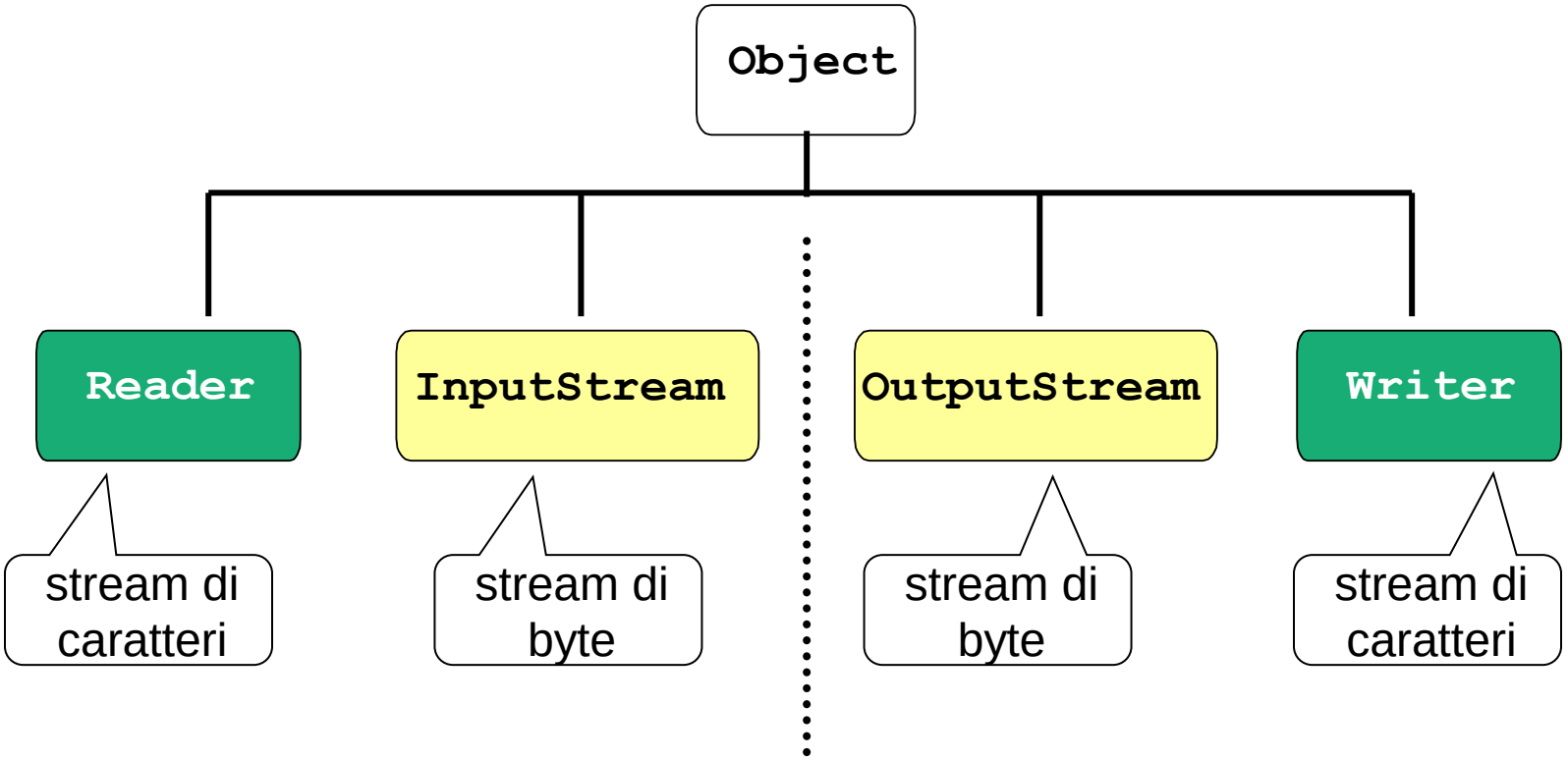
Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

RECAP: CLASSI BASE ASTRATTE



RECAP: ARCHITETTURA

Il package **`java.io`** definisce i *concetti-base* e *l'architettura* per gestire l'I/O:

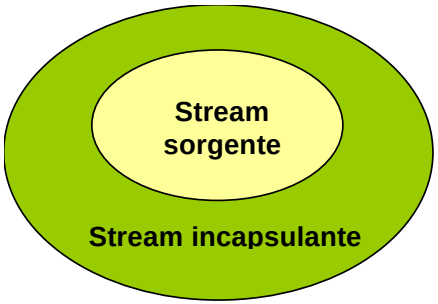
- da qualsiasi *sorgente*
- verso qualsiasi *destinazione*

in modo *configurabile.*

Incapsulare in un oggetto i dettagli di una data *sorgente* o di un *dispositivo di output*

Comporre uno stream personalizzato sovrapponendo a uno *stream "nucleo" altri strati* che lo adattino alle necessità del caso specifico

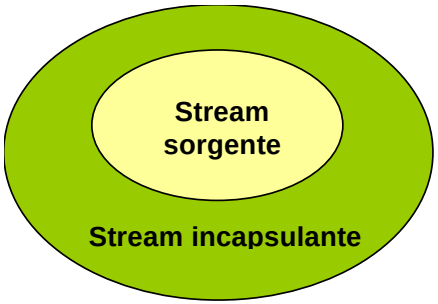
PATTERN Adapter + Chain of responsibility



APPROCCIO "A CIPOLLA"

Per questo, sia gli stream di byte, sia quelli di caratteri si dividono in *due grandi categorie*:

- stream destinati a incapsulare sorgenti fisiche di dati o dispositivi fisici di uscita
  - i loro costruttori hanno come argomento *il dispositivo che interessa*: file, connessioni di rete, array di byte,…
- stream di adattamento pensati per *adattare i precedenti* alle necessità, aggiungendo ulteriori funzionalità
  - i loro costruttori hanno come argomento *uno stream già esistente* (il nucleo da "avvolgere")



Stream di byte

STREAM DI BYTE: INPUT

La classe base **`InputStream`** definisce il concetto generale di *"canale di input"* operante *a byte*

- il costruttore apre lo stream
- il metodo **`read`** legge uno o più *byte*
- il metodo **`close`** chiude lo stream

Poiché **`InputStream`** è una *classe astratta*, il metodo **`read`** *dovrà essere effettivamente definito dalle classi derivate*, in modo adeguato alla specifica sorgente dati.

STREAM DI BYTE: OUTPUT

La classe base **`OutputStream`** definisce il concetto generale di *"canale di output"* operante *a byte*

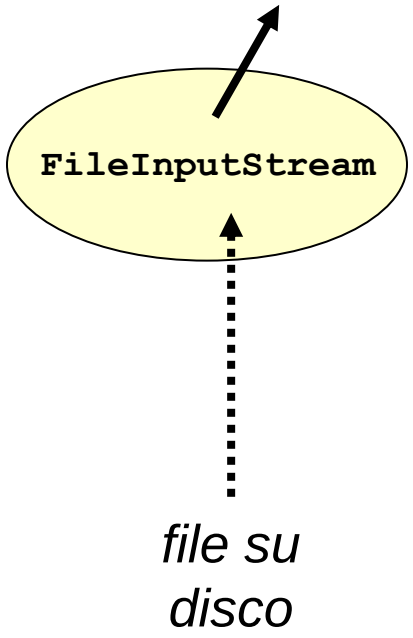
- il costruttore apre lo stream
- il metodo **`write`** scrive uno o più *byte*
- il metodo **`flush`** svuota il buffer di uscita
- il metodo **`close`** chiude lo stream

Poiché anche **`OutputStream`** è una *classe astratta*, il metodo **`write`** *dovrà essere effettivamente definito dalle classi derivate*, in modo adeguato alla specifica destinazione dati.

STREAM DI BYTE: INPUT DA FILE

`FileInputStream` è la classe concreta che rappresenta il concetto di *sorgente di byte agganciata a un file*

- il costruttore apre il file in lettura
  - argomento: o il nome del file da aprire, o un oggetto **`File`** preventivamente costruito
  - lancia eccezione a controllo obbligatorio
- il metodo **`read`** legge un byte
  - restituisce il byte letto come intero fra 0 e 255
  - se lo stream è finito, restituisce -1
  - se non ci sono byte, ma lo stream non è finito, *rimane in attesa* dell'arrivo di un byte
  - lancia eccezione a controllo obbligatorio



ESEMPIO (1/3)

```
import java.io.*;

public class LetturaDaFileBinario {
 public static void main(String args[]){
   FileInputStream is = null;
   try {
     is = new FileInputStream(args[0]);
   }
   catch(FileNotFoundException e){
     System.out.println("File non trovato");
     System.exit(1);
   }
  // QUI va la lettura
 }
}
```

Per terminare in modo pulito restituendo un codice d'errore

ESEMPIO (2/3)

La fase di lettura:

```
try {
 int x = is.read();
 int n = 0;
 while (x>=0) {
    System.out.print(" " + x); n++;
    x = is.read();
 }
System.out.println("\nTotale byte: " + n);
} catch(IOException ex){
  System.out.println("Errore di input");
  System.exit(2);
}
```

quando lo stream termina, **read** restituisce -1

Per terminare in modo pulito restituendo un codice d'errore

ESEMPIO (3/3)

Uso:

```
C:\temp>java LetturaDaFileBinario question.gif
```

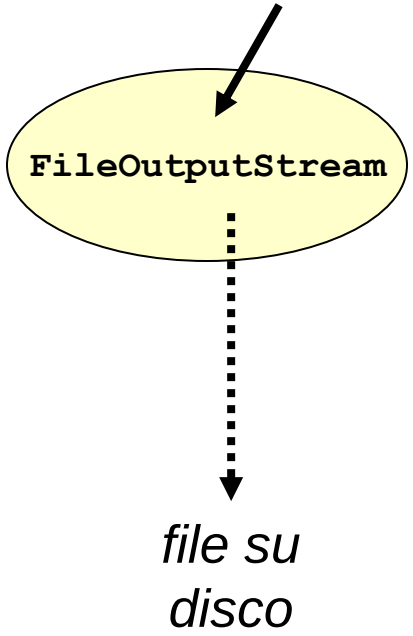
Risultato:

```
71 73 70 56 57 97 32 0 32 0 161 0 0 0 0 0 255 255 255 0 128 0
191 191 191 33 249 4 1 0 0 3 0 44 0 0 0 0 32 0 32 0 0 2 143 156
143 6 203 155 15 15 19 180 82 22 227 178 156 187 44 117 193 72
118 0 56 0 28 201 150 214 169 173 237 236 65 170 60 179 102 114
91 121 190 11 225 126 174 151 112 56 162 208 130 61 95 75 249
100 118 4 203 101 173 57 117 234 178 155 172 40 58 237 122 43
214 48 214 91 54 157 167 105 245 152 12 230 174 145 129 183 64
140 142 115 83 239 118 141 111 23 120 161 16 152 100 7 3 152
229 87 129 152 200 166 247 119 68 103 24 196 243 232 215 104
249 181 21 25 67 87 9 130 7 165 134 194 35 202 248 81 106 211
128 129 82 0 0 59
Totale byte: 190
```

STREAM DI BYTE: OUTPUT DA FILE

`FileOutputStream` è la classe concreta che rappresenta il concetto di *dispositivo di uscita agganciato a un file*

- il costruttore apre il file in scrittura
  - primo argomento: il nome del file da aprire, o un oggetto **`File`** preventivamente costruito
  - secondo argomento (opzionale): un boolean per chiedere l'apertura in modalità append
  - lancia eccezione a controllo obbligatorio
- il metodo **`write`** scrive uno o più byte
  - scrive l'intero passatogli (fra 0 e 255)
  - non restituisce nulla
  - lancia eccezione a controllo obbligatorio



OUTPUT SU FILE: ESEMPIO (1/3)

```
import java.io.*;
public class ScritturaSuFileBinario {
 public static void main(String args[]){
   FileOutputStream os = null;

   try {
     os = new FileOutputStream(args[0]);
   }
   catch(FileNotFoundException e){
     System.out.println("Imposs. aprire file");
     System.exit(1);
   }
  // QUI va la scrittura
 }
}
```

Per aprirlo in modalità append:
`FileOutputStream(args[0],true)`

OUTPUT SU FILE: ESEMPIO (2/3)

Fase di scrittura (di alcuni byte "importantissimi")

```
try {
 for (int x=0; x<10; x+=3) {
    System.out.println("Scrittura di " + x);
    os.write(x);
 }
} catch(IOException ex){
   System.out.println("Errore di output");
   System.exit(2);
}
```

OUTPUT SU FILE: ESEMPIO (3/3)

Uso:

```
C:\temp>java ScritturaSuFileBinario prova.dat
```

Risultato:

```
Scrittura di 0
Scrittura di 3
Scrittura di 6
Scrittura di 9
```

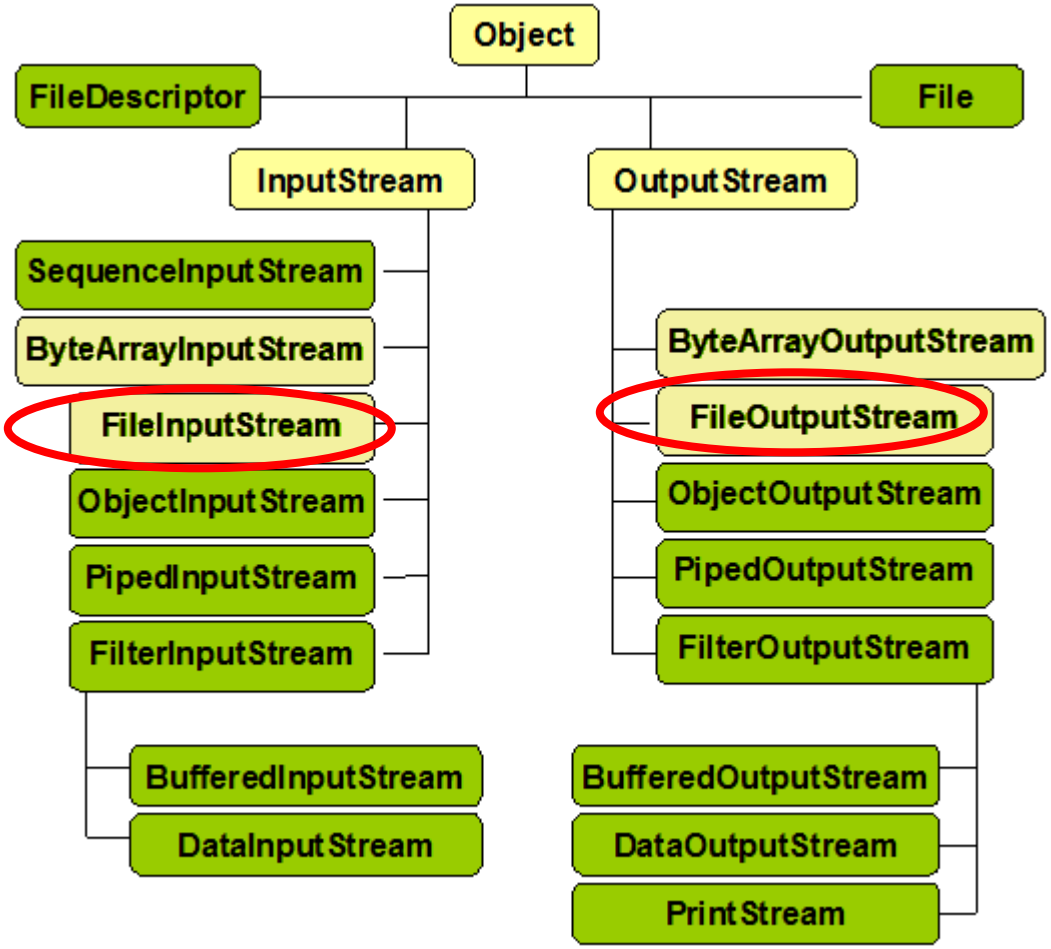
Controllo:

```
C:\temp>dir prova.dat
16/01/17  prova.dat   4 byte
```

Esperimenti

- Provare a rileggere il file con il programma precedente
- Aggiungere altri byte riaprendo il file in modalità append

STREAM DI BYTE: TASSONOMIA COMPLETA

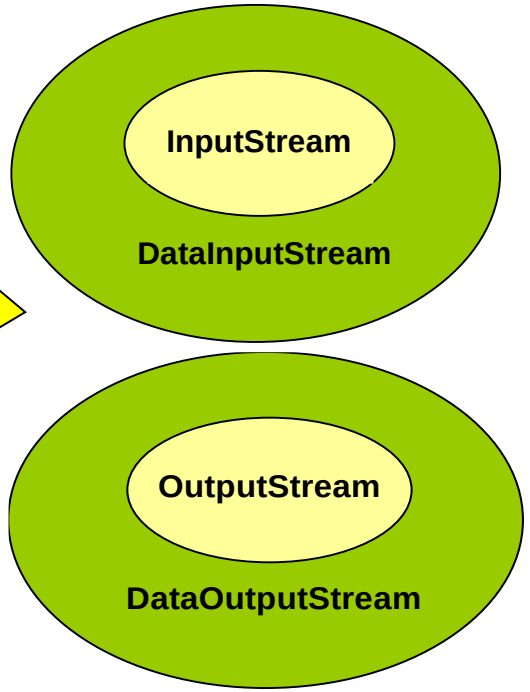


Stream atti a fungere da «nucleo» (si agganciano a un dispositivo fisico)

ADAPTER STREAMS

- Gli stream di adattamento hanno lo scopo di *adattare uno stream pre-esistente* per *creare un'entità più evoluta*.
- Si riconoscono perché il *costruttore* prende come argomento *un* ***`InputStream`*** *o un* ***`OutputStream`***

Ad esempio, per leggere o scrivere dei valori primitivi **`int`**, **`float`**, etc. è molto comodo adattare lo stream base tramite un **`DataInputStream`** / **`DataOutputStream`**



STREAM INCAPSULANTI per INPUT

Alcuni stream notevoli per l'input binario:

- **`DataInputStream`**
  (implementa `DataInput`)
  - definisce metodi per leggere *valori primitivi:* **`readInteger`**, **`readFloat`**, ecc.
  - lanciano **`EOFException`** se lo stream termina in modo inatteso

- **`ObjectInputStream`**
  (implementa `ObjectInput` e `DataInput`)
  - definisce il metodo **`readObject`** per leggere *oggetti serializzati*
  - ma contiene anche gli stessi **`readInteger`**, **`readFloat`**, ecc. di cui sopra per comodità
  - lanciano **`EOFException`** se lo stream termina in modo inatteso

STREAM INCAPSULANTI per OUTPUT

Alcuni stream notevoli per l'output binario:

- **`DataOutputStream`**
  (implementa `DataOutput`)
  - definisce metodi per scrivere *valori primitivi:* **`writeInteger`**, **`writeFloat`**, ecc.
  - lanciano **`EOFException`** se lo stream termina in modo inatteso

- **`ObjectOutputStream`**
  (implementa `ObjectOutput` e `DataOutput`)
  - definisce il metodo **`writeObject`** per scrivere *oggetti serializzati*
  - ma contiene anche gli stessi **`writeInteger`**, **`writeFloat`**, ecc. di cui sopra per comodità
  - lanciano **`EOFException`** se lo stream termina in modo inatteso

STREAM INCAPSULANTI per OUTPUT

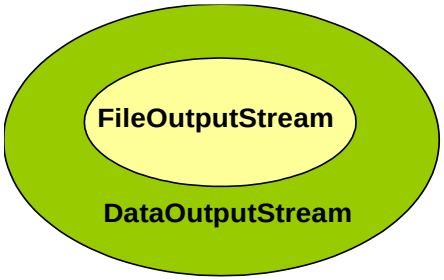
Ce ne sarebbe anche un altro, ma è *obsoleto*:

- **`PrintStream`**
  - definisce metodi per scrivere *stringhe:* **`print`**, **`println`**, ecc.
  - *è deprecato da anni: non va assolutamente usato!*
  - esistono alternative molto migliori → **`PrintWriter`**
  - però non può essere del tutto dimenticato, perché purtroppo è il tipo di **`System.out`** e **`System.err`** !
  - introdotti in Java 1.0, quei due stream non possono essere modificati per retrocompatibilità… *ma potremo adattarli* ☺

ESEMPIO 1 (1/3)

OBIETTIVO: scrivere dei valori primitivi su file binario

- Per scrivere su un file binario serve un nucleo costituito da un **`FileOutputStream`** che però, di suo, consente di scrivere solo dei byte → *scomodo*
- Per scrivere valori primitivi è decisamente più pratico un **`DataOutputStream`**, che ha metodi nativi per farlo
- Costruiamo perciò la cipolla che ci serve: incapsuliamo il nucleo **`FileOutputStream`** in un **`DataOutputStream`**



ESEMPIO 1 (2/3)

```
import java.io.*;
public class Esempio1 {
 public static void main(String args[]){
   FileOutputStream fs = null;
   try {
    fs = new FileOutputStream("Prova.dat");
   }
  catch(IOException e){
     System.out.println("Apertura fallita");
     System.exit(1);
   }
   // continua
   ...
```

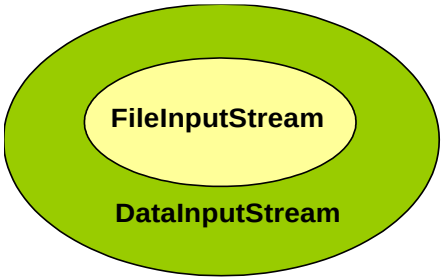
ESEMPIO 1 (3/3)

```
    ...
    DataOutputStream os =
                        new DataOutputStream(fs);
    float   f1 = 3.1415F;  char   c1 = 'X';
    boolean b1 = true;     double d1 = 1.4142;
    try {
      os.writeFloat(f1);  os.writeBoolean(b1);
      os.writeDouble(d1); os.writeChar(c1);
      os.writeInt(12);    os.close();
    } catch (IOException e){
      System.out.println("Scrittura fallita");
      System.exit(2);
    }
  }
}
```

ESEMPIO 2 (1/3)

OBIETTIVO: rilettura di valori primitivi da file binario

- Per leggere su un file binario serve un nucleo costituito da un **`FileInputStream`** che però, di suo, consente di leggere solo dei byte → *scomodo*
- Per leggere valori primitivi è decisamente più pratico un **`DataInputStream`**, che ha metodi nativi per farlo
- Costruiamo perciò la cipolla che ci serve: incapsuliamo il nucleo **`FileInputStream`** in un **`DataInputStream`**



ESEMPIO 2 (2/3)

```
import java.io.*;
public class Esempio2 {
 public static void main(String args[]){
   FileInputStream  fin = null;
   try {
     fin = new FileInputStream("Prova.dat");
   }
   catch(FileNotFoundException e){
     System.out.println("File non trovato");
     System.exit(3);
   }
  // continua...
```

ESEMPIO 2 (3/3)

```
  ...
  DataInputStream is = new DataInputStream(fin);
  float f2; char c2; boolean b2;    double d2;  int
  i2;
  try {
   f2 = is.readFloat(); b2 = is.readBoolean();
   d2 = is.readDouble(); c2 = is.readChar();
   i2 = is.readInt();    is.close();
   System.out.println(f2 + ", " + b2 + ", "
   + d2 + ", " + c2 + ", " + i2);
  } catch (IOException e){
    System.out.println("Errore di input");
    System.exit(4);
  }
}
```

Solo chi l'ha scritto può sapere *cosa* ha scritto e *in che ordine*: **l'informazione è negli occhi di chi guarda!**

SERIALIZZAZIONE DI OGGETTI

In molti casi è utile e necessario poter *salvare interi oggetti* per poi successivamente *ricostruirli*.

- *Serializzare* significa *salvare un oggetto* in una opportuna *rappresentazione binaria* → **`ObjectOutputStream`**
- *Deserializzare* significa *ricostruire un oggetto* a partire dalla sua rappresentazione binaria → **`ObjectInputStream`**

Tuttavia, poiché gli oggetti possono contenere *dati critici e riservati*, di cui si perde il controllo se escono dalla JVM, *non è opportuno che questo sia fattibile sempre e comunque.*

Le classi *serializzabili* devono essere *esplicitamente marcate come tali* dal progettista.

SERIALIZZAZIONE DI OGGETTI

In molti casi è utile e necessario poter *salvare interi oggetti* per poi successivamente *ricostruirli*.

- *Serializzare* significa *salvare un oggetto* in una opportuna *rappresentazione binaria* → **`ObjectOutputStream`**
- *Deserializzare* significa *ricostruire un oggetto* a partire dalla sua rappresentazione binaria → **`ObjectInputStream`**

Tuttavia, poiché gli oggetti possono contenere *dati critici e riservati*, di cui si perde il controllo se escono dalla JVM, *non è opportuno che questo sia fattibile sempre e comunque.*

Per questo, le classi serializzabili devono *implementare l'interfaccia (vuota)* **`Serializable`**

L'INTERFACCIA `Serializable`

- L'interfaccia **`Serializable`** è un *marcatore vuoto:* implementarla *non costa nulla* perché non dichiara metodi
  - infatti, i metodi per salvare/ricostruire un oggetto *comunque complesso* da uno stream di byte sono già inclusi in **`java.io`**
- Tuttavia, obbligando le classi da serializzare a contenere la frase **`implements Serializable`**:
  - si abilita il controllo di coerenza da parte del compilatore
  - si forza il progettista a esprimere esplicitamente il suo *design intent*
- Inoltre, per *distinguere versioni successive* della stessa classe si stabilisce di aggiungere un *numero di versione:* un long di nome **`SerialVersionUID`**

ESEMPIO: `Point` serializzabile

```
public class Punto2D implements java.io.Serializable {
  float x, y;
  private static final long serialVersionUID = 1;
  public Punto2D(float xx, float yy){ x = xx; y = yy; }
  public float getX(){ return x; }
  public float getY(){ return y; }
}
```

- Il numero long **`SerialVersionUID`** **deve** essere privato, statico e final (non modificabile in alcun modo)
- Per convenzione, solitamente si parte da 1

SERIALIZZAZIONE DI OGGETTI

I due metodi chiave:

- il metodo **`writeObject`** serializza un oggetto
  - formalmente il metodo accetta qualunque **`Object`**, ma se non è serializzabile *lancia eccezione*
- il metodo **`readObject`** lo ricostruisce (de-serializza)
  - formalmente restituisce un **`Object`**, che deve esistere al momento dell'esecuzione (altrimenti, eccezione)
  - in realtà però l'oggetto restituito sarà di un *tipo specifico*, *che solo noi possiamo conoscere* perché solo noi sappiamo cosa avevamo salvato in precedenza
    → *necessità di un down-cast* in assegnamento

ESEMPIO: SALVATAGGIO DI Point

```
Punto2D p = new Punto2D(3.2F, 1.5F);
try {
  FileOutputStream f = new FileOutputStream("xy.bin");
  ObjectOutputStream os = new ObjectOutputStream(f);
  os.writeObject(p);
  os.flush();
  os.close();
}
catch (IOException e){
  ...
}
```

Fa tutto lui!

ESEMPIO: RILETTURA DI Point

```
Punto2D p = null;
try {
  FileInputStream f = newFileInputStream("xy.bin");
  ObjectInputStream is = new ObjectInputStream(f);
  p = (Punto2D) is.readObject();
  is.close();
}
catch (IOException | ClassNotFoundException e){
  ...
}
// se tutto è ok
System.out.print("x,y = " + p.getX()+", " + p.getY());
```

DownCast (**readObject** restituisce un **Object**)

ALCUNE TIPICHE DOMANDE (1)

- Come si serializza una sequenza di oggetti?
  - basta scriverli con **`writeObject`** uno dopo l'altro, *in un ordine noto prestabilito dal progettista.*
  - oppure… *salva direttamente l'intera sequenza!* (array, lista…)
- Come si rilegge una sequenza di oggetti?
  - *poiché sono stati scritti in un ordine prestabilito,* basta rileggerli con **`readObject`** *nello stesso ordine*
    → Attenzione: per farlo *bisogna sapere a priori quanti sono*, altrimenti **`readObject`** lancia **`EOFException`**
    Per questo occorre *progettare bene il formato del file*: prima un intero col numero totale di elementi, poi gli elementi!
  - ..ma *se hai salvato l'intera sequenza, basta rileggere un singolo mega-oggetto:* pensa a tutto **`readObject`**!

ALCUNE TIPICHE DOMANDE (2)

- Cosa succede se non dichiaro il numero di versione?
  - ne viene calcolato uno di default, che però dipende da dettagli della classe che potrebbero variare in diversi compilatori
    → rischio di `InvalidClassException` inattesa
  - ergo, sempre meglio definire `serialVersionUID` esplicitamente
  - NB: gli array non sono soggetti al check di versione
- Cosa succede se il numero di versione è diverso?
  - eccezione! se in fase di lettura il numero di versione è diverso, *la classe è stata modificata* e quindi lo stream ha perso di significato

```
java.io.InvalidClassException: Punto2D;
local class incompatible:
stream classdesc serialVersionUID = 8,
local class serialVersionUID = 3
```

Java 7: la stream factory

LA FABBRICA DEGLI STREAM

- Finora abbiamo creato gli stream direttamente (con **`new`**)
  - massimo controllo, ma anche massima visibilità di ogni dettaglio
- Java 7 affianca alla creazione diretta quella *indiretta*, tramite *factory methods della classe* **`Files`**
  - metodi con SIGNATURE UNIFORME
  - astrazione **`OpenOption`** per esprimere le opzioni di apertura

```
newInputStream( Path p, OpenOption... options)
newOutputStream(Path p, OpenOption... options)
```

Notare la notazione *varargs* **`OpenOption...`** per indicare un *numero variabile* di argomenti (da zero in su)

Rivediamo alcuni degli esempi precedenti in questa nuova prospettiva.

ESEMPIO 1 – *NIO version*

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.io.OutputStream;
import java.io.DataOutputStream;
import java.io.IOException;

public class NioEsempio1 {
 public static void mai...
   OutputStream fs = null;
   try {
    fs = Files.newOutputStream(Paths.get("Prova.dat"));
   }
   catch(IOException e){
     System.out.println("Apertura fallita");
     System.exit(1);
   }
   // continua come prima
```

Le due classi factory/libreria di **`java.nio.file`** più le altre classi di **`java.io`**

Il factory method restituisce **`OutputStream`** (NON **`FileOutputStream`**)

ESEMPIO 2 – *NIO version*

```
// import varie analoghe

public class NioEsempio2 {
 public static void main(String
   InputStream fin = null;
   try {
     fin = Files.newInputStream(Paths.get("Prova.dat"));
   }
   catch(FileNotFoundException e1){
     System.out.println("File non trovato");
     System.exit(3);
   }
   catch(IOException e2){
     System.out.println("Errore di input");
     System.exit(4);
   }
   // continua come prima
```

Il factory method restituisce **`InputStream`** (NON **`FileInputStream`**)

Il factory method può lanciare anche **`IOException`** (ci sono le opzioni..) oltre a **`FileNotFoundException`**

Java 7: il nuovo costrutto *`try`-with-resources*

LA PROLIFERAZIONE DEI `try/catch`

- Poiché le eccezioni dell'I/O sono a controllo obbligatorio, la pratica ha mostrato una *proliferazione di* ***`try/catch`*** che porta spesso a codice poco leggibile
- Per questo, Java 7 ha introdotto una *sintassi specifica* per *incapsulare la creazione dello stream nel blocco* ***`try`***

```
try (apertura file e altre risorse) {
        ...
}
```

Anche più di una: il costrutto accetta una *lista* di aperture di risorse, separate da '`;`'

- Il costrutto *chiude automaticamente i file* e le altre risorse all'uscita dal blocco **`try`**
  - a tal fine, le risorse coinvolte devono implementare **`AutoCloseable`**

IL COSTRUTTO *try-with-resources*

Sintassi classica

```
// dichiarazione stream
try {
 // apertura stream
 // operazioni sullo stream
}
catch(IOException e){
  ...
}
finally{
   // chiusura stream
}
```

Sintassi *try-with-resources*

nessuna dichiarazione

```
try(apertura stream){
 // operazioni sullo stream
}
catch(IOException e){
  ...
}
```

nessun finally per garantire la chiusura

- meno "boilerplate code"
- miglior leggibilità globale
- evidenza ciò che conta

ESEMPIO ORIGINALE

```
public static void main(String args[]){
  FileInputStream is = null;
  try {
    is = new FileInputStream(args[0]);
  }
  catch(FileNotFoundException e){
    System.out.println("File non trovato");
    System.exit(1);
  }
  try {
    int x = is.read(), n = 0;
    while (x>=0) {
       System.out.print(" " + x); n++; x = is.read();
    }
    System.out.println("\nTotale byte: " + n);
  }
  catch(IOException ex){
    System.out.println("Errore di input");
    System.exit(2);
  }
}
```

solo apertura stream

parte operativa

ESEMPIO REVISIONATO

```
public static void main(String args[]){
```

PARTE ELIMINATA

apertura stream inserita direttamente nella parte "operativa"

```
  try(InputStream is = Files.newInputStream(Paths.get(args[0]))) {
    int x = is.read(), n = 0;
    while (x>=0) {
       System.out.print(" " + x); n++; x = is.read();
    }
    System.out.println("\nTotale byte: " + n);
  }
  catch(IOException ex){
    System.out.println("Errore di input");
    System.exit(2);
  }
}
```

chiusura stream automatica ☺

ESEMPIO ORIGINALE

```
public static void main(String args[]){
  FileOutputStream os = null;
  try {
    os = new FileOutputStream(args[0]);
  }
  catch(FileNotFoundException e){
    System.out.println("Imposs. aprire file");
    System.exit(1);
  }
  try {
   for (int x=0; x<10; x+=3) {
      System.out.println("Scrittura di " + x);
      os.write(x);
   }
  } catch(IOException ex){
    System.out.println("Errore di output");
    System.exit(2);
  }
}
```

solo apertura stream

parte operativa

ESEMPIO REVISIONATO

```
public static void main(String args[]){
```

PARTE ELIMINATA

apertura stream inserita direttamente nella parte "operativa"

```
   try(OutputStream os = Files.newOutputStream(Paths.get(args[0]))){
    for (int x=0; x<10; x+=3) {
        System.out.println("Scrittura di " + x);
        os.write(x);
    }
   } catch(IOException ex){
     System.out.println("Errore di output");
     System.exit(2);
   }
}
```

chiusura stream automatica ☺

CONCLUSIONE

Il costrutto ***`try`-with-resources***

- migliora nettamente la leggibilità del codice
- automatizza parte della gestione delle risorse
- *garantisce la chiusura pulita* delle risorse utilizzate
- MA proprio per questo *non* si deve usare se le risorse sono *usate anche dopo l'uscita* dal blocco **`try`**
  - caso di buon uso tipico: si apre un file, lo si incapsula in un reader e poi lo si usa dentro al blocco try stesso → OK
  - caso di cattivo uso: si apre un file, lo si incapsula in un reader *destinato a essere usato successivamente*
    → al momento dell'uso il reader è già stato chiuso: ECCEZIONE!