PACKAGE DI I/O

Dott. Denis Ferraretti

denis.ferraretti@unife.it

lezioni

- Mercoledì 24 febbraio 2010 ore 11.00 - 13.30
- Lunedì 1 marzo 2010 ore 11.00 - 13.30

Tutte le lezioni sono nel
laboratorio di Informatica Grande.

argomenti

- Gli STREAM: concetti di base
- STREAM di byte
- STREAM di caratteri
- Gestione di File (classe File)
- Serializzazione

Il package `java.io`

- Il package **`java.io`** definisce i concetti base per gestire l'I/O da *qualsiasi sorgente* e verso *qualsiasi destinazione*.

- Obiettivi del package:
  - fornire **un'astrazione** che incapsuli tutti i dettagli di una sorgente dati o di un dispositivo di output.
  - fornire un modo **semplice e flessibile** per aggiungere ulteriori funzionalità a quelle fornite dal canale "base"

concetto base: gli stream

- I programmi in JAVA comunicano (cioè effettuano operazioni di I/O) tramite gli **STREAM.**

- **Uno STREAM è un'astrazione di alto livello che produce o consuma informazioni: rappresenta una connessione ad un canale di comunicazione.**

- Gli stream possono sia leggere da un canale di comunicazione che scrivere su tale canale: quindi si parla di stream di input e stream di output.

concetto base: gli stream

- Uno STREAM rappresenta quindi un'estremità di un canale di comunicazione **a un senso solo**.
- Le classi di STREAM forniscono metodi per leggere da un canale o per scrivere su un canale.
- Quindi un *output stream* scrive dei dati su un canale di comunicazione, mentre un *input stream* legge dati da un canale di comunicazione.

ATTENZIONE!

Non esistono delle classi di stream che forniscano funzioni sia per leggere che per scrivere su un canale.

Se si desidera sia leggere che scrivere su uno stesso canale di comunicazione si dovranno aprire due stream (uno di input e uno di output) collegati allo stesso canale.

concetto base: gli stream

- Gli STREAM si comportano in modo omogeneo, indipendentemente da dispositivo fisico con cui sono collegati (**da qui il concetto di astrazione ad alto livello**).

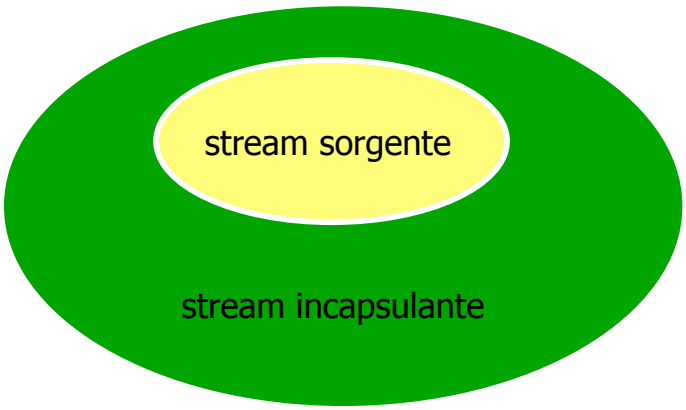
- Le stesse classi e gli stessi metodi di I/O posso essere applicati a qualunque dispositivo.

- Uno STREAM di input può essere utilizzato per leggere da un file su disco, da tastiera o dalla rete; allo stesso modo uno STREAM di output può fare riferimento alla console (a video), a un file, o ancora ad una connessione di rete.

concetto base: l'approccio

L'approccio *"a cipolla"*

- **alcuni tipi di stream rappresentano sorgenti di dati o dispositivi di uscita [stream sorgente]**
  - **file, connessioni di rete,**
  - **array di byte, ...**

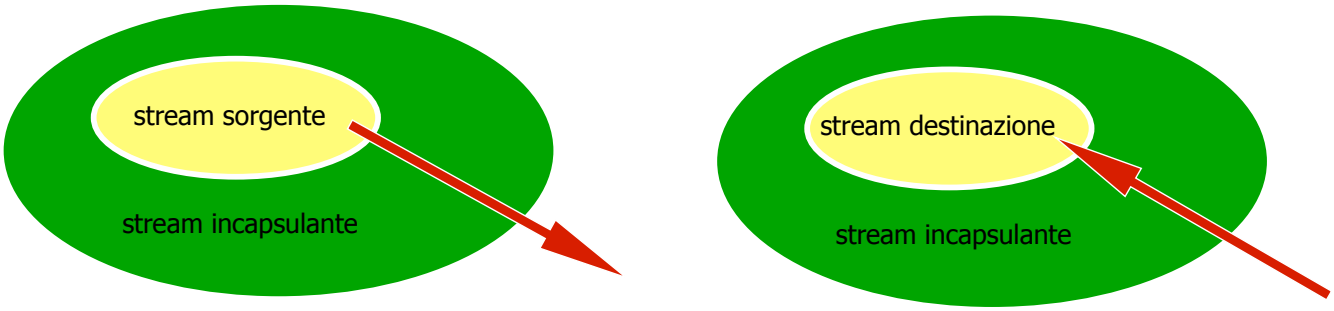


- **gli altri tipi di stream sono pensati per *"avvolgere" i precedenti* per aggiungere ulteriori funzionalità [stream incapsulante]**

concetto base: l'approccio

- Così, è possibile configurare il canale di comunicazione *con tutte e sole le funzionalità che servono*...
- **senza doverle replicare e reimplementare più volte.**

Massima flessibilità, minimo sforzo.



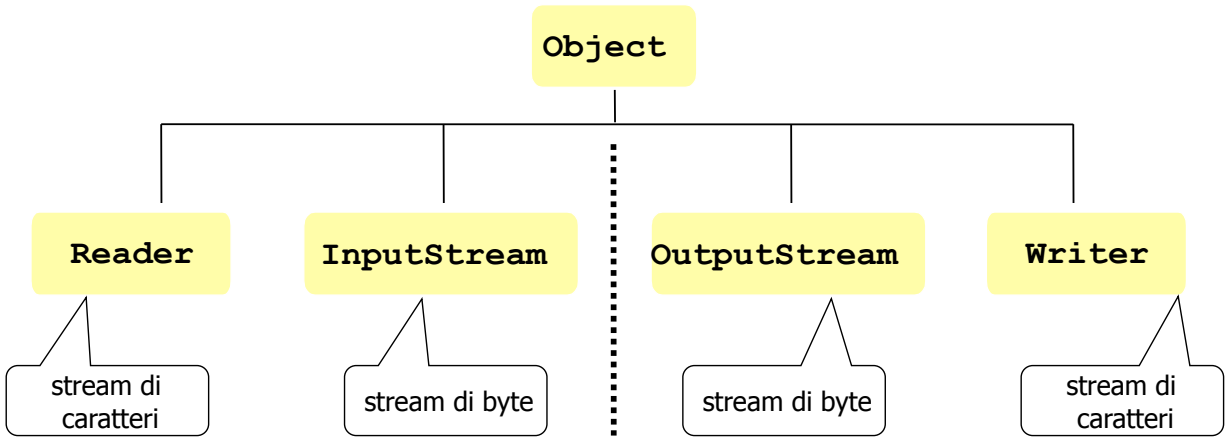
il package `java.io`

Il package **`java.io`** distingue fra:

- ***stream di byte*** (analoghi ai *file binari* del C)
- ***stream di caratteri*** (analoghi ai *file di testo* del C)

Questi concetti si traducono in **altrettante classi base astratte**:

- stream di byte: **`InputStream`** e **`OutputStream`**
- stream di caratteri: **`Reader`** e **`Writer`**



stream di byte

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

- Il costruttore apre lo stream
  - **`read()`** legge uno o più byte
  - **`close()`** chiude lo stream

Attenzione!

`InputStream` è una classe astratta, quindi il metodo **`read()`** dovrà essere realmente definito dalle classi derivate, in modo specifico alla specifica sorgente dati.

stream di byte

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

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

Attenzione!

`OutputStream` è una classe astratta, quindi il metodo **`write()`** dovrà essere realmente definito dalle classi derivate, in modo specifico allo specifico dispositivo di uscita.

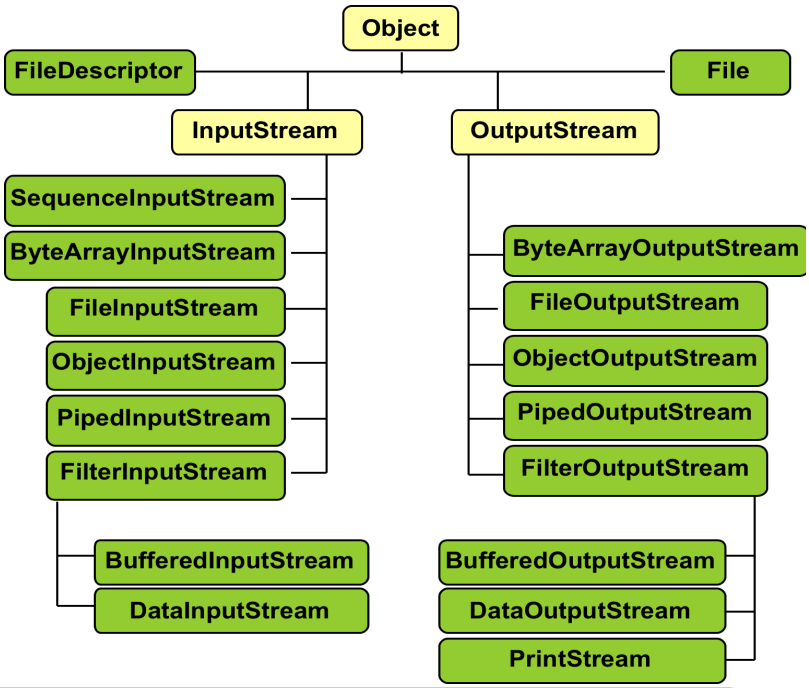
stream di byte

- Dalle classi base astratte **si derivano varie classi concrete**, specializzate per fungere da:
  - sorgenti per input da file
  - dispositivi di output su file
  - stream di incapsulamento, cioè pensati per aggiungere a un altro stream nuove funzionalità.

Esempio:

I/O bufferizzato, filtrato,…

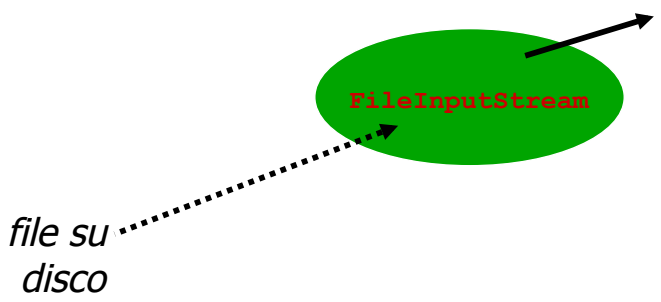
I/O di numeri, di oggetti,...



stream di byte – input da file

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

- Il **nome del file** da aprire è passato come parametro al costruttore di `FileInputStream`
- In alternativa si può passare al costruttore **un oggetto File** (o un FileDescriptor) costruito in precedenza



stream di byte – input da file

- Per aprire un **file binario in lettura** si crea un oggetto di classe **`FileInputStream`**, specificando il nome del file all'atto della creazione.
- Per leggere dal file si usa poi il metodo **`read()`** che permette di leggere uno o più 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.

Poiché è possibile che le operazioni su stream falliscano per varie cause, tutte le operazioni possono sollevare eccezioni.

Necessità di utilizzare blocchi **`try / catch`**

input da file – esempio

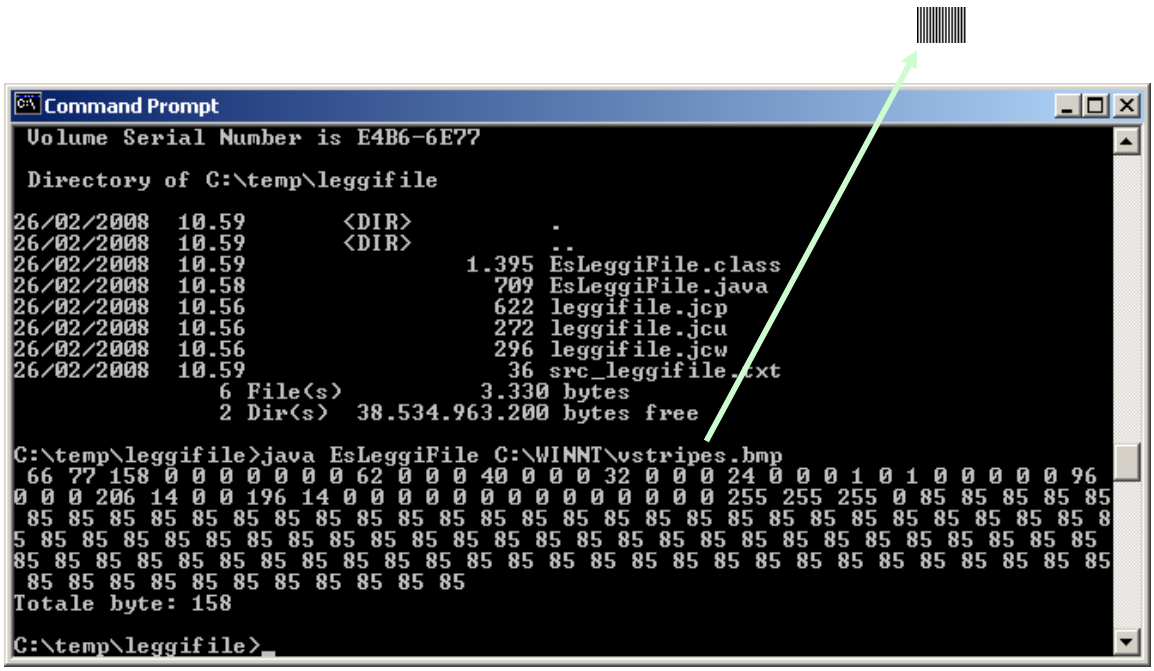
```
import java.io.*;
public class EsLeggiFile1 {
   public EsLeggiFile1(String filename) throws
    FileNotFoundException, IOException {

    FileInputStream is = new FileInputStream(filename);
    int x = is.read();
    int n = 0;
    while (x >= 0) {
        System.out.print(" " + x);
        n++;
        x = is.read();
    }
    System.out.println("\nTotale byte: " + n);
    is.close();
   }
...
```

quando lo stream termina, `read()` restituisce -1

input da file – esempio

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

    try {
       EsLeggiFile1 oggetto = new EsLeggiFile1(args[0]);
    }
    catch (FileNotFoundException ex) {
        System.out.println("File non trovato");
            System.exit(1);
        }
        catch(IOException ex) {
            System.out.println("Errore di input");
            System.exit(2);
    }
   }
}
```

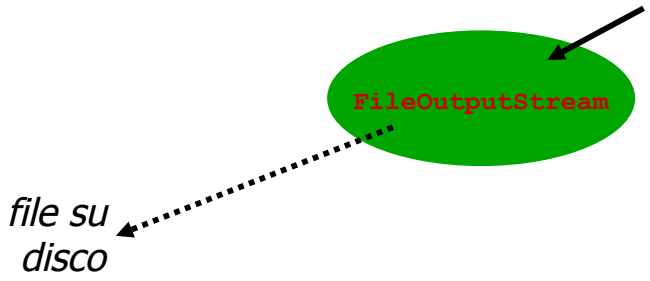

input da file – esempio



stream di byte – output su file

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

- Il nome del file da aprire è passato come parametro al costruttore di **`FileOutputStream`**
- In alternativa si può passare al costruttore un **oggetto File** (o un FileDescriptor) costruito in precedenza



stream di byte – output su file

- Per aprire un **file binario in scrittura** si crea un oggetto di classe **`FileOutputStream`**, specificando il nome del file all'atto della creazione
- Un secondo parametro opzionale, di tipo **`boolean`**, permette di chiedere l'apertura in modo append
- Per scrivere sul file si usa il metodo **`write()`** che permette di scrivere uno o più byte
  - **scrive l'intero [0, 255] passatogli come parametro**
  - **non restituisce nulla**

Poiché è possibile che le operazioni su stream falliscano per varie cause, tutte le operazioni possono sollevare eccezioni.

Necessità di utilizzare blocchi **`try / catch`**

stream di byte – output su file

```
import java.io.*;

public class EsScriviFile1 {

    public EsScriviFile1(String filename)
       throws FileNotFoundException, IOException {

        FileOutputStream os = new FileOutputStream(filename);

        for(int x=0; x<10; x+=3) {
       System.out.println("Scrittura di "+x);
       os.write(x);
        }
        os.close();
    }
...
```

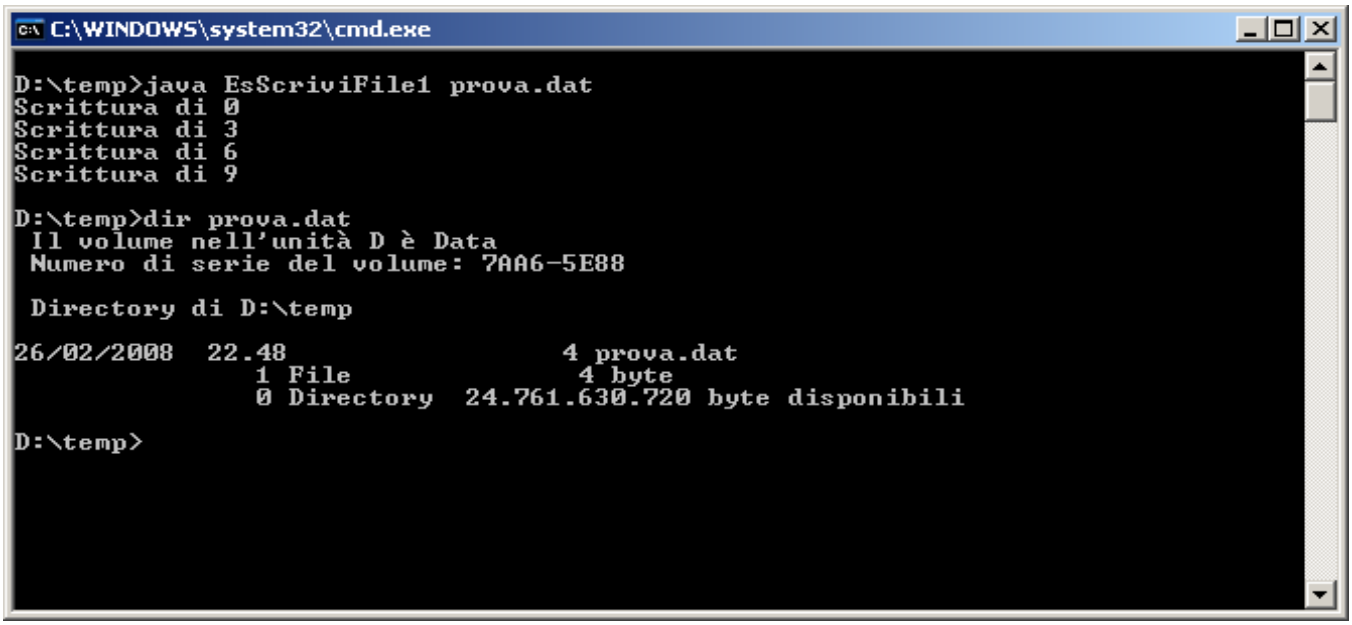
Per aprirlo in modalità append:
`FileOutputStream(filename,true);`

stream di byte – output su file

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

     try {
         EsScriviFile1 oggetto = new EsScriviFile1(args[0]);
     }
     catch(FileNotFoundException e) {
       System.out.println("Impossibile aprire file");
       System.exit(1);
     }
     catch(IOException ex) {
       System.out.println("Errore di output");
       System.exit(2);
     }
  }
}
```

STRUMENTI JAVA PER LO SVILUPPO DI INTERFACCE UTENTE E SERVIZI DI RETE E LORO APPLICAZIONE

stream di byte – output su file

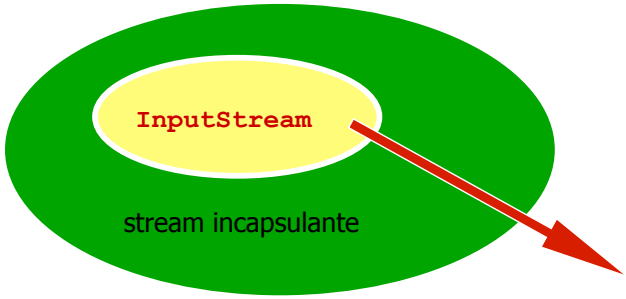


Esperimenti

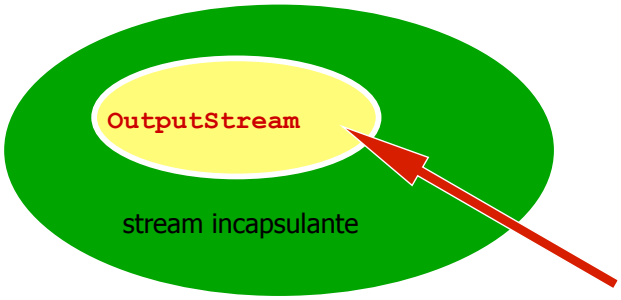
- *Provare a rileggere il file con il programma precedente*
- *Aggiungere altri byte riaprendo il file in modo append*

stream di incapsulamento

- Gli STREAM di incapsulamento hanno come scopo quello di **"avvolgere"** un altro STREAM per creare un'entità con funzionalità più evolute.



Il loro costruttore ha quindi come parametro un **`InputStream`** o un **`OutputStream`** già esistente.



stream di incapsulamento - INPUT

- **`BufferedInputStream`**

  aggiunge un buffer e ridefinisce **`read()`** in modo da avere una lettura bufferizzata

- **`DataInputStream`**

  definisce metodi per leggere i tipi di dati standard in forma binaria: **`readInt()`**, **`readFloat()`**, ...

- **`ObjectInputStream`**

  definisce un metodo per leggere oggetti **"serializzati"** (salvati) da uno stream, offre anche metodi per leggere i tipi primitivi di Java

stream di incapsulamento - OUTPUT

- **`BufferedOutputStream`**

  aggiunge un buffer e ridefinisce **`write()`** in modo da avere una scrittura bufferizzata

- **`DataOutputStream`**

  definisce metodi per scrivere i tipi di dati standard in forma binaria: **`writeInt()`**, **`writeFloat()`** ...

- **`PrintStream`**

  definisce metodi per stampare come stringa valori primitivi (es. **`print(int)`**) e classi standard (es. **`print(Object)`**)

- **`ObjectOutputStream`**

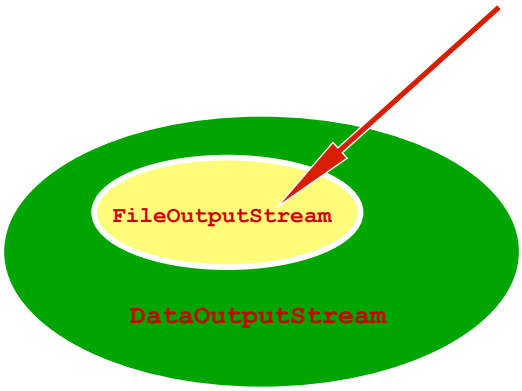
  definisce un metodo per scrivere oggetti **"serializzati"**; offre anche metodi per scrivere i tipi primitivi di Java

esempio - OUTPUT

Scrittura di dati su file binario

- Per scrivere su un file binario occorre un **`FileOutputStream`**, che però consente solo di scrivere un byte o un array di byte
- Volendo scrivere dei **`float, int, double, boolean`**, … **è molto più pratico un `DataOutputStream`**, che ha metodi idonei

- **Si incapsula `FileOutputStream` dentro un `DataOutputStream`**



esempio - OUTPUT

```
import java.io.*;
public class EsScriviFile2 {
  public EsScriviFile2(String filename)
      throws FileNotFoundException, IOException {

    FileOutputStream fos=new FileOutputStream(filename);
    DataOutputStream dos=new DataOutputStream(fos);
    float   f = 3.1415F;   char    c = 'X';
    boolean b = true;      double  d = 1.4142;
    int     i = 12;
    dos.writeFloat(f);     dos.writeChar(c);
    dos.writeBoolean(b);   dos.writeDouble(d);
    dos.writeInt(i);    dos.close();
  }
. . .
```

esempio - OUTPUT

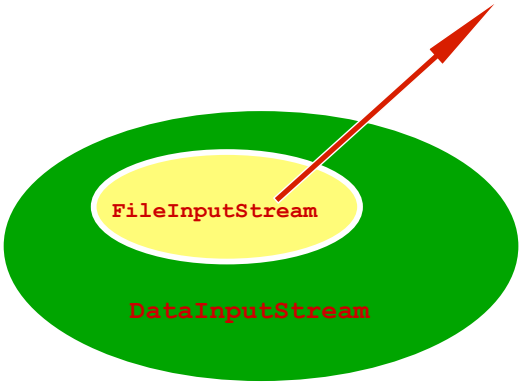
```
public static void main(String[] args) {

  try {
    EsScriviFile2 oggetto = new EsScriviFile2(args[0]);
  }
  catch(FileNotFoundException e) {
    System.out.println("Impossibile aprire file");
    System.exit(1);
  }
  catch(IOException ex) {
    System.out.println("Errore di output");
    System.exit(2);
   }
 }
}
```


esempio - INPUT

Lettura di dati da file binario

- Per leggere da un file binario occorre un **`FileInputStream`**, che però consente solo di leggere un byte o un array di byte
- Volendo leggere dei **`float, int, double, boolean,`** … **è molto più pratico un `DataInputStream`**, che ha metodi idonei

- **Si incapsula `FileInputStream` dentro un `DataInputStream`**



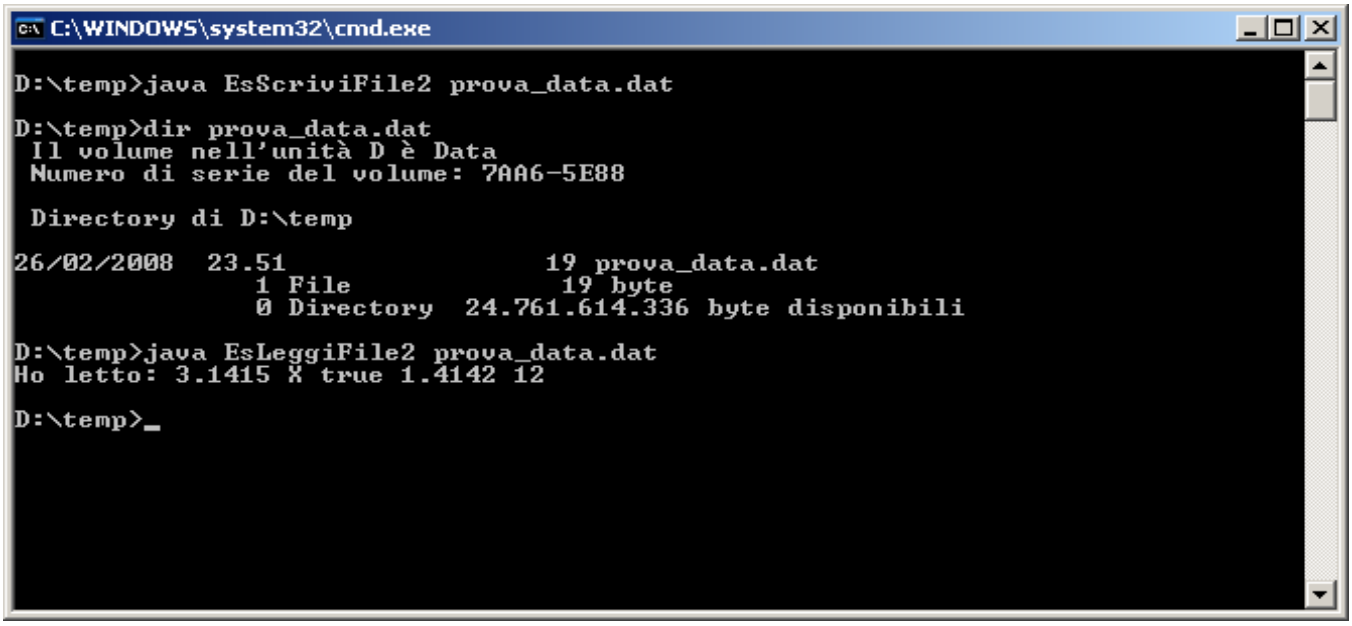
esempio - INPUT

```
import java.io.*;
public class EsLeggiFile2 {
  public EsLeggiFile2(String filename)
       throws FileNotFoundException, IOException {

     FileInputStream fis = new FileInputStream(filename);
     DataInputStream dis = new DataInputStream(fis);
     float   f = dis.readFloat();
     char    c = dis.readChar();
     boolean b = dis.readBoolean();
     double  d = dis.readDouble();
     int     i = dis.readInt();
     System.out.println("Ho letto:"+f+" "+c+" "+b+" "+d+" "+i);
     dis.close();
  }
. . .
```


esempio - INPUT

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

        try {
            EsLeggiFile2 oggetto = new EsLeggiFile2(args[0]);
        }
        catch(FileNotFoundException e) {
           System.out.println("Impossibile aprire file");
           System.exit(1);
        }
        catch(IOException ex) {
           System.out.println("Errore di output");
           System.exit(2);
      }
    }
}
```

esempio - risultato

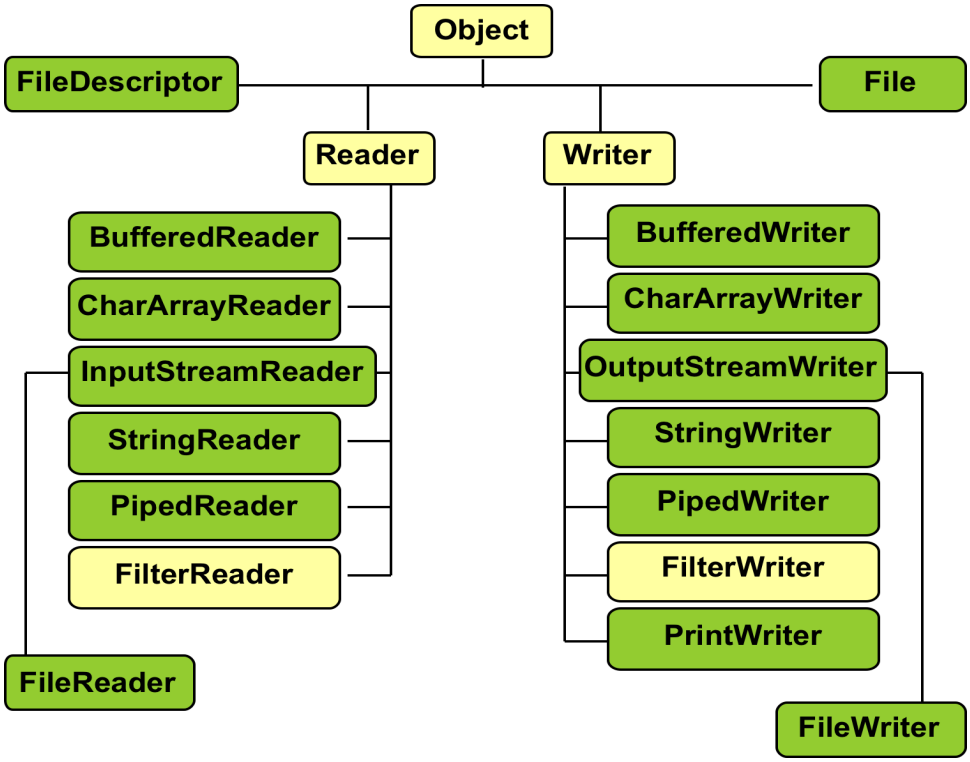


stream di caratteri

- Le classi per l'I/O da stream di caratteri (**`Reader`** e **`Writer`**) sono *più efficienti* di quelle a byte.
- Hanno nomi analoghi e struttura analoga

- Convertono correttamente la codifica **UNICODE** di Java in quella locale
  - specifica della piattaforma in uso (tipicamente ASCII)...
  - ...e della lingua in uso (essenziale per l'internazionalizzazione).

stream di caratteri



stream di caratteri

Cosa cambia rispetto agli stream binari ?

- Il file di testo si apre costruendo un oggetto **`FileReader`** o **`FileWriter`**, rispettivamente
- **`read()`** e **`write()`** leggono/scrivono **un `int`** che rappresenta un carattere **UNICODE**

Un carattere UNICODE è lungo due byte.

- **`read()`** restituisce un valore tra 0 e 65535
- oppure -1 in caso di fine stream

Occorre dunque un cast esplicito per convertire il carattere **UNICODE** in **`int`** e viceversa.

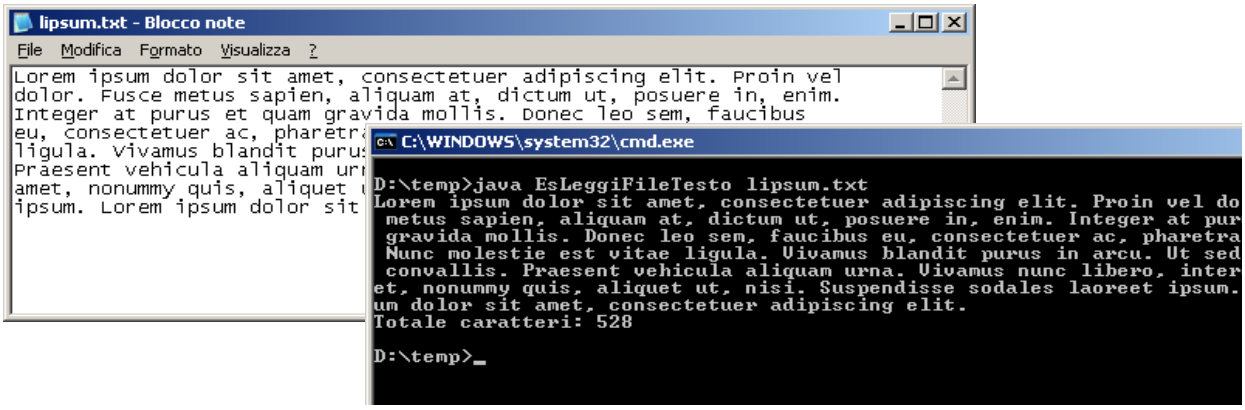
esempio - input da file

```
import java.io.*;
public class EsLeggiFileTesto {
    public EsLeggiFileTesto(String filename)
                throws IOException {
        FileReader fr = new FileReader(filename);
        int n = 0;
        int x = fr.read();
        while (x >= 0) {
            char ch = (char) x;
            System.out.print(ch);
            n++;
            x = fr.read();
        }
       System.out.println("\nTotale caratteri: " + n);
        fr.close();
    }
```

Cast esplicito da `int` a `char` - Ma solo se è stato *davvero* letto un carattere (cioè se non è stato letto -1)

esempio - input da file

```
  public static void main(String[] args) {
    try {
      EsLeggiFileTesto oggetto =
          new EsLeggiFileTesto(args[0]);
    }
    catch(IOException ex) {
      System.out.println("Errore di I/O.");
      System.exit(1);
    }
  }
}
```



stream di incapsulamento

- Le operazioni di I/O con caratteri in genere coinvolgono tanti caratteri alla volta.

- In genere l'unità di base per lavorare con i caratteri è la **LINEA:** una stringa di caratteri con un terminatore di linea alla fine (in win "\r\n" carriage-return/line-feed).

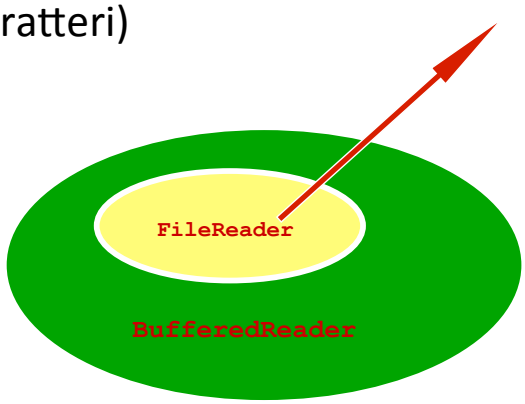
- Anche nel caso degli stream di caratteri, possiamo utilizzare gli **stream di incapsulamento**, che estendono le funzionalità dei `Reader` e `Writer` fornendo metodi idonei al trattamento delle linee di testo.

stream di incapsulamento - INPUT

`BufferedReader` lettura bufferizzata

- Fornisce i metodi:
  - **`readLine()`** ↰
    - restituisce una stringa contenente il testo presente nella linea escluso il terminatore
    - **`null`** se si raggiunge la fine dello stream
  - **`read()`** (singolo carattere) e array di caratteri)

- **Si incapsula `FileReader` dentro un `BufferedReader`**

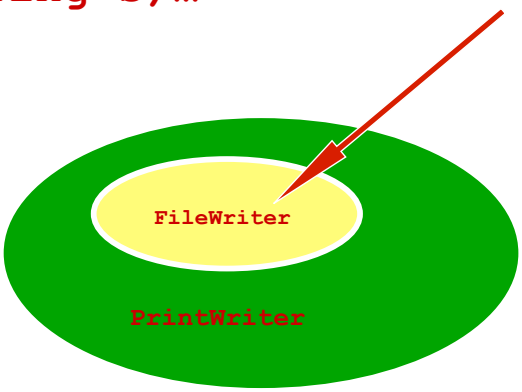


stream di incapsulamento - OUTPUT

`PrintWriter` scrittura "formattata"

- Fornisce i metodi per scrivere i tipi primitivi e non, su uno stream di testo.
- Ad esempio: **`print(boolean b), print(double d), print(String s)`**…
- O la loro versione con il terminatore: **`println(boolean b), println(double d), println(String s)`**…

- **Si incapsula `FileWriter` dentro un `PrintWriter`**



esempio – INPUT/OUTPUT

```
import java.io.*;
public class CopyLines {

    public CopyLines(String filename) throws IOException {

        FileReader fr = new FileReader(filename);
        BufferedReader input = new BufferedReader(fr);
        FileWriter fw = new FileWriter("Copia_di_" + filename);
        PrintWriter output = new PrintWriter(fw);

        String linea = input.readLine();
        while (linea != null) {
            output.println(linea);
            linea = input.readLine();
        }
        input.close();
        output.close();
    }
. . .
```

esempio – INPUT/OUTPUT

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

        try {
            CopyLines cp = new CopyLines(args[0]);
        }
        catch(IOException ex) {
            System.out.println("Errore di I/O.");
            System.exit(1);
        }
    }
}
```


stream di caratteri

UN PROBLEMA

- Gli stream di byte **sono antecedenti** e **di livello più basso** rispetto agli stream di caratteri
  - Un carattere **UNICODE** viene espresso a livello macchina come sequenza di due byte
  - Gli stream di byte esistono da Java 1.0, quelli di caratteri esistono invece da Java 1.1
- Varie classi esistenti fin da Java 1.0 usano quindi **stream di byte anche quando dovrebbero usare in realtà stream di caratteri**
- Conseguenza: i caratteri rischiano di non essere sempre trattati in modo coerente

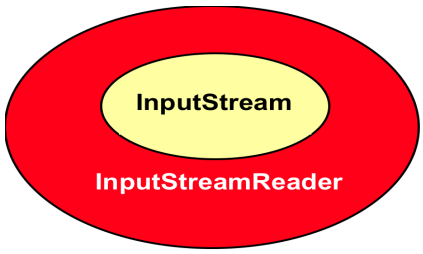
stream di caratteri

SOLUZIONE

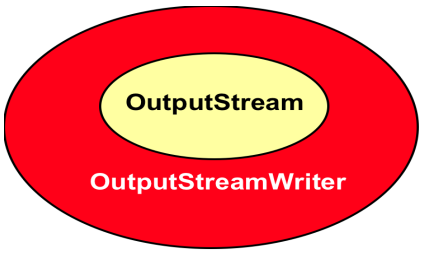
- Occorre dunque poter **reinterpretare** uno stream di byte come **`Reader / Writer`** **quando opportuno** (cioè quando trasmette caratteri)
- Esistono **due classi "incapsulanti"** progettate proprio per questo scopo:
  - **`InputStreamReader`** che reinterpreta un **`InputStream`** come un **`Reader`**
  - **`OutputStreamWriter`** che reinterpreta un **`OutputStream`** come un **`Writer`**

stream di caratteri

- **`InputStreamReader`** ingloba un **`InputStream`** e lo fa apparire all'esterno come un **`Reader`**
- **`OutputStreamWriter`** ingloba un **`OutputStream`** e lo fa apparire fuori come un **`Writer`**



da fuori, si vede un ***`Reader`***



da fuori, si vede un ***`Writer`***

il caso dell'I/O da console

- Video e tastiera sono rappresentati dai due oggetti statici **`System.in`** e **`System.out`**

- Poiché esistono fin da Java 1.0 (quando **`Reader`** e **`Writer`** non esistevano), essi **sono formalmente degli stream di byte...**
  - **`System.in`** è formalmente un **`InputStream`**
  - **`System.out`** è formalmente un **`OutputStream`**

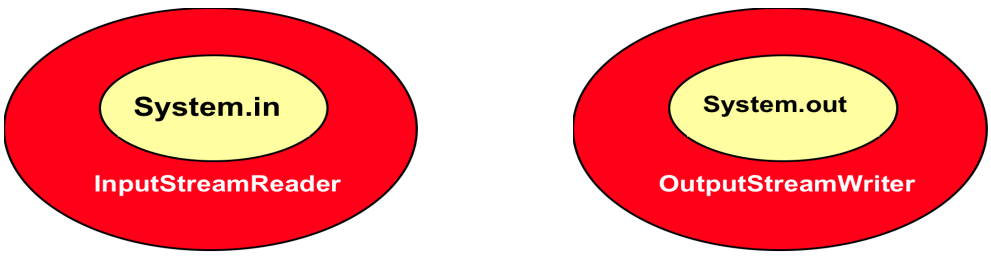
  ...ma **in realtà sono stream di caratteri!**

  Per assicurare che i caratteri UNICODE siano correttamente interpretati occorre quindi incapsularli rispettivamente in un Reader e in un Writer.

il caso dell'I/O da console

- **`System.in`** può essere **"interpretato come un `Reader`"** incapsulandolo dentro a un **`InputStreamReader`**

- System.out può essere **"interpretato come un `Writer`"** incapsulandolo dentro a un **`OutputStreamWriter`**



il caso dell'I/O da console

IN PRATICA

```
InputStreamReader tastiera =
    new InputStreamReader(System.in);

OutputStreamWriter video =
    new OutputStreamWriter(System.out);
```

MA! Per quanto detto precedentemente, è meglio scrivere:

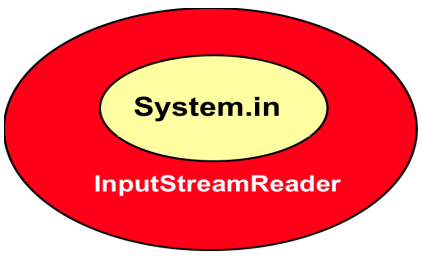
```
BufferedReader tastiera =
   new BufferedReader(new InputStreamReader(System.in));

PrintWriter video =
   new PrintWriter(new OutputStreamWriter(System.out));
```

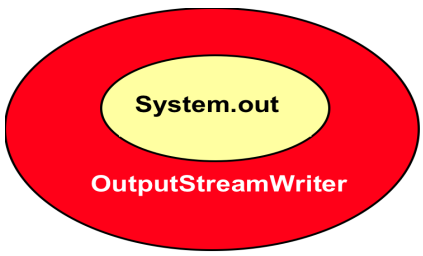
Equivalente a:
`System.out.println(…)`

il caso dell'I/O da console

- **`System.in`** può essere **"interpretato come un `Reader`"** incapsulandolo dentro a un **`InputStreamReader`**

- System.out può essere **"interpretato come un `Writer`"** incapsulandolo dentro a un **`OutputStreamWriter`**





esempio – scrittura su file

- Scrivere su un file di testo le linee inserite da console, fino a quando non viene inserita la linea vuota.

- Bisogna incapsulare **`System.in`** in un **`InputStreamReader`** che a sua volta è incapsulato da un **`BufferedReader`**
- Si può inoltre incapsulare **`FileWriter`** in un **`PrintWriter`**

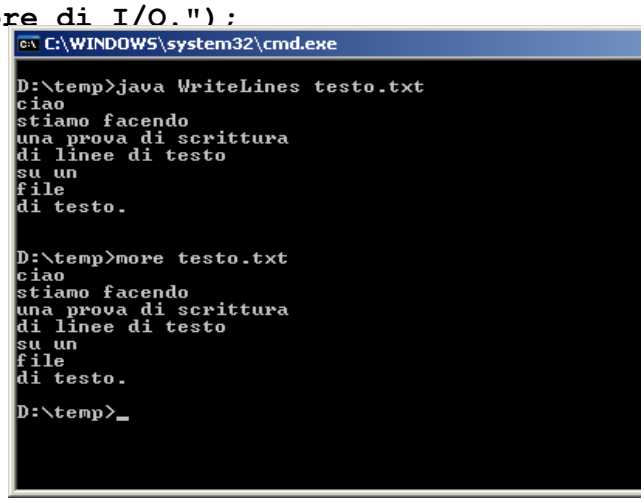
- **Leggere linee di testo e inserirle nel file fino a quando non si incontra una linea vuota.**

esempio – scrittura su file

```
import java.io.*;
public class WriteLines {

    public WriteLines(String filename) throws IOException {

        PrintWriter output =
        new PrintWriter(new FileWriter(filename));
        BufferedReader input =
        new BufferedReader(new InputStreamReader(System.in));

        String linea = input.readLine();
        while (!linea.equals("")) {
            output.println(linea);
            linea = input.readLine();
        }
        input.close();
        output.close();
    }
. . .
```

esempio – scrittura su file

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

        try {
            WriteLines cp = new WriteLines(args[0]);
        }
        catch(IOException ex) {
            System.out.println("Errore di I/O.");
            System.exit(1);
        }
    }
}
```



classe `File`

- La classe **`File`** fornisce l'accesso a file e directory in modo indipendente dal sistema operativo.
- Mette a disposizione una serie di metodi per ottenere informazioni su un certo file o directory, e per visualizzare e modificarne gli attributi.

- A proposito di indipendenza…

Ogni sistema operativo utilizza convenzioni diverse per separare le varie directory in un *path.* Esempio: in Linux " / ", in Windows " \ ";

- Costruttore:

`public File(String path)`

classe `File` – metodi utili

- **`public String getName()`**
  restituisce il nome dell'oggetto
- **`public String getAbsolutePath()`**
  restituisce il percorso assoluto dell'oggetto
- **`public boolean exist()`**
  restituisce vero se l'oggetto File esiste
- **`public boolean isDirectory()`**
  restituisce vero se l'oggeto File è una direcotry
- **`public long length()`**
  restituisce la lunghezza in byte dell'oggetto
- **`public boolean renameTo(File dest)`**
  rinomina l'oggetto
- **`public boolean delete()`**
  cancella l'oggetto File
- **`public boolean mkdir()`**
  crea una directory che corrisponde all'oggetto File
- **`public String[] list()`**
  restituisce un vettore contenente il nome di tutti file della directory associata all'oggetto File

classe `RandomAccessFile`

- Consente l'accesso in modo RANDOM (cioè non sequenziale), ai file.
- Consente l'accesso ad un file contemporaneamente in scrittura e il lettura.
- Implementa le interfacce DataInput e DataOutput, rendendo possibile la scrittura in file di tutti gli oggetti e i tipi primitivi.

- Costruttori:

```
public RandomAccessFile(String file, String mode)
public RandomAccessFile(File file, String mode)
```

- Il parametro `mode` è di tipo `String` e specifica la modalità di accesso al file: `"r"` oppure `"rw"`

- **Oltre ai metodi `read/write`, offre i metodi:**

```
long getFilePointer(), seek(long pos), skipBytes(long pos )
```

serializzazione di oggetti

- **Serializzare un oggetto** significa salvare un oggetto scrivendo **una sua rappresentazione binaria su uno stream di byte**

- Analogamente, **deserializzare un oggetto** significa ricostruire un oggetto a partire dalla sua rappresentazione binaria letta da uno stream di byte

- Le classi `ObjectOutputStream` e `ObjectInputStream` offrono questa funzionalità per qualunque tipo di oggetto.

serializzazione di oggetti

Le due classi principali sono:

- `ObjectInputStream`
  - **legge oggetti** serializzati salvati su stream, tramite il metodo `readObject()`
  - offre anche metodi per leggere i tipi primitivi di Java

- `ObjectOutputStream`
  - **scrive un oggetto** serializzato su stream, tramite il metodo `writeObject()`
  - offre anche metodi per scrivere i tipi primitivi di Java

serializzazione di oggetti

- Una classe che voglia essere **"serializzabile"** **deve implementare** l'interfaccia `Serializable`

- È una **interfaccia vuota**, che serve come marcatore
  (il compilatore rifiuta di compilare una classe che usa la serializzazione senza implementare tale interfaccia)

- Vengono **scritti / letti tutti i dati non static e non transient dell'oggetto, inclusi quelli ereditati** (anche se privati o protetti)

serializzazione di oggetti

- Se un oggetto **contiene riferimenti ad altri oggetti**, si invoca ricorsivamente `writeObject()` su ognuno di essi
  - si serializza quindi, in generale, un intero grafo di oggetti
  - l'opposto accade quando si deserializza

- Se uno stesso oggetto è referenziato più volte nel grafo, **viene serializzato una sola volta**, affinché `writeObject()` non cada in una ricorsione infinita.

esempio - serializzazione

ESEMPIO di classe serializzabile

```
public class PuntoCartesiano
      implements java.io.Serializable {
  private int x;
  private int y;

  public PuntoCartesiano(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public PuntoCartesiano() { x = y = 0; }

   public float getAscissa()   { return x; }
   public float getOrdinata() { return y; }
}
```


esempio - serializzazione

SCRITTURA SU FILE…

```
import java.io.*;

public class ScriviPunto {

   public ScriviPunto(String filename)
                          throws IOException {

     PuntoCartesiano point = new PuntoCartesiano(3, 10);
     FileOutputStream fos =
             new FileOutputStream(filename);
     ObjectOutputStream oos =
             new ObjectOutputStream(fos);
     oos.writeObject(point);
     oos.flush();
     oos.close();
   }
```

……

esempio - serializzazione

...SCRITTURA SU FILE

```
...
public static void main(String[] args) {
        try {
            ScriviPunto sp = new ScriviPunto(args[0]);
        }
        catch(IOException ex) {
            System.out.println("Errore di I/O.");
       System.exit(1);
        }
    }
}
```


esempio - serializzazione

LETTURA DA FILE...

```
import java.io.*;
public class LeggiPunto {
  public LeggiPunto(String filename)
              throws IOException, ClassNotFoundException {

    FileInputStream fis = new FileInputStream(filename);
    ObjectInputStream ois = new ObjectInputStream(fis);

    PuntoCartesiano point = (PuntoCartesiano)ois.readObject();
    ois.close();

    System.out.println("Punto (" + point.getAscissa() +
       ", " + point.getOrdinata() + ")");
    }
.....
```

Il cast è necessario, perché `readObject()` restituisce un `Object`

esempio - serializzazione

…LETTURA DA FILE

```
   ……
   public static void main(String[] args) {
      try {
         LeggiPunto lp = new LeggiPunto(args[0]);
      }
      catch(IOException ex) {
         System.out.println("Errore I/O: " + ex.getMessage());
         System.exit(1);
      }
       catch(ClassNotFoundException ex) {
         System.out.println("Errore: " + ex.getMessage());
         System.exit(2);
      }

   }
}
```