Reti e Laboratorio III
Modulo Laboratorio III
AA. 2023-2024

docente: Laura Ricci

laura.ricci@unipi.it

Lezione 4

Stream based IO: richiami

12/10/2023

JAVA I/O

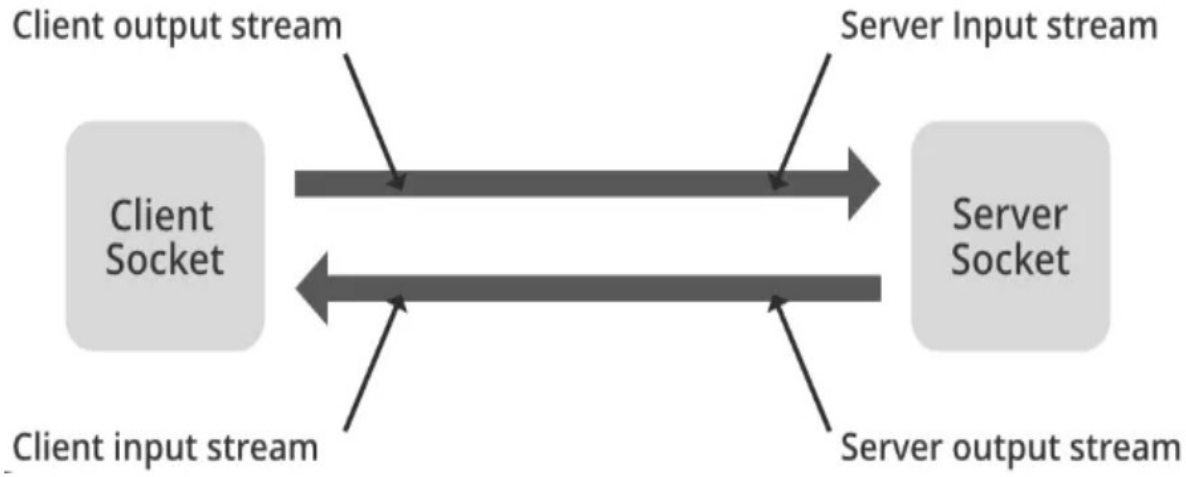
- definizione di un insieme di astrazioni per la gestione dell'I/O: una delle parti più complesse di un linguaggio

- diversi tipi di device di input/output: se il linguaggio dovesse gestire ogni tipo di device come caso speciale, la complessità sarebbe enorme
  - necessità di astrazioni opportune per rappresentare una device di `I/O`

- in `JAVA`, la prima astrazione definita è basata sul concetto di stream (o flusso)

- altre astrazioni per l'`I/O`
  - `File:` per manipolare descrittori di files
  - `Channels (NIO)`
- perchè importanti per questo corso? Le connessioni `TCP` possono essere modellate in `JAVA` con streams

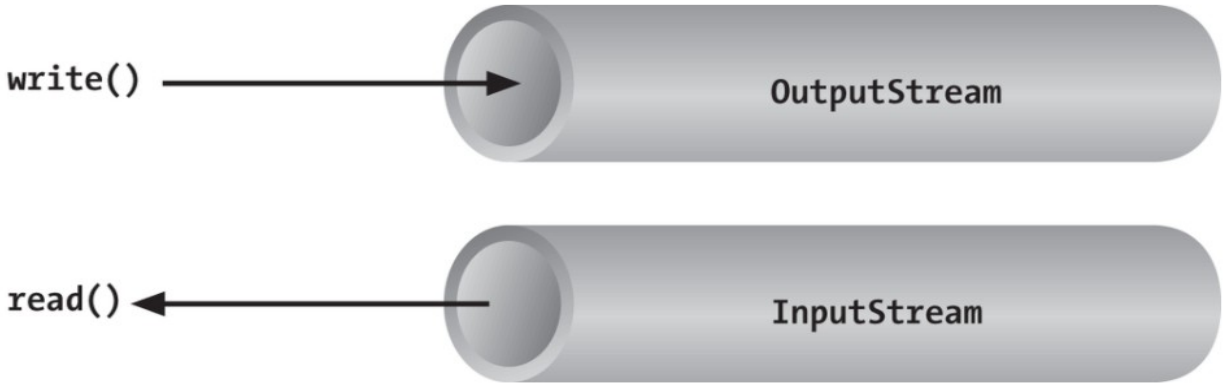
STREAM E RETI



- socket: endpoint per inviare/ricevere dati
  - astrazione che "maschera" complessità della rete
- stream: astrazione che modella la connessione tramite un socket TCP
  - dopo che il protocollo TCP verrà introdotto nel modulo di teoria vedremo socket + stream nel laboratorio

L'ASTRAZIONE DEGLI STREAM

- uno stream rappresenta una connessione tra un programma JAVA ed un dispositivo esterno (file, buffer di memoria, connessione di rete,...)
- un flusso di informazione di lunghezza illimitata



- un "tubo" tra una sorgente ed una destinazione (dal programma ad un dispositivo e viceversa)
- l'applicazione inserisce dati o li legge ad/da un capo dello stream
- i dati fluiscono da/verso la destinazione

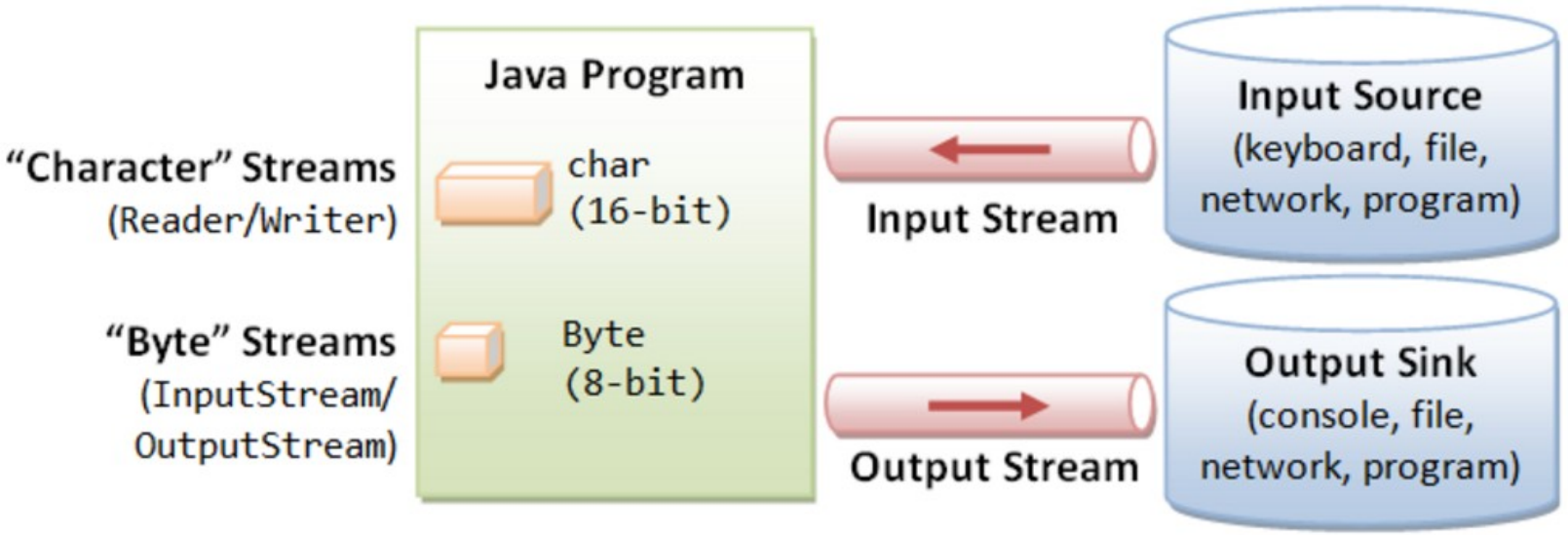
JAVA STREAMS: CARATTERISTICHE GENERALI

- accesso sequenziale
- mantengono l'ordinamento FIFO
- one way: read only oppure write only (a parte i file ad accesso random)
  - se un programma ha bisogno di dati in input ed output, è necessario aprire due stream, in input ed in output
- bloccanti: quando un'applicazione legge un dato dallo stream (o lo scrive) si blocca finchè l'operazione non è completata
- non è richiesta una corrispondenza stretta tra letture/scritture
  - una unica scrittura inietta 100 bytes sullo stream
  - i byte vengono letti con due write successive 'all'altro capo dello stream', la prima legge 20 bytes, la seconda 80 bytes)

IL PACKAGE JAVA.IO: OBIETTIVI

- fornire un'astrazione che incapsuli tutti i dettagli del dispositivo sorgente/ destinazione dei dati
- fornire un modo semplice e flessibile per aggiungere ulteriori funzionalità quelle fornite dallo "stream base"
- un approccio "a livelli"
  - alcuni stream di base per connettersi a dispositivi "standard": file, connessioni di rete, console,.....
  - altri stream sono pensati per "avvolgere" i precedenti ed aggiungere ulteriori funzionalità
  - così è possibile configurare lo stream con tutte le funzionalità che servono senza doverle re-implementare più volte

LE CLASSI PRINCIPALI: CARATTERI E BYTE



Internal Data Formats:

- Text (char): UCS-2
- int, float, double, etc.

External Data Formats:

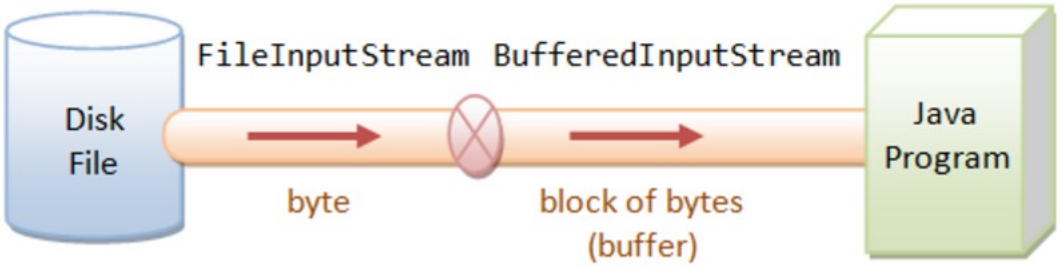
- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

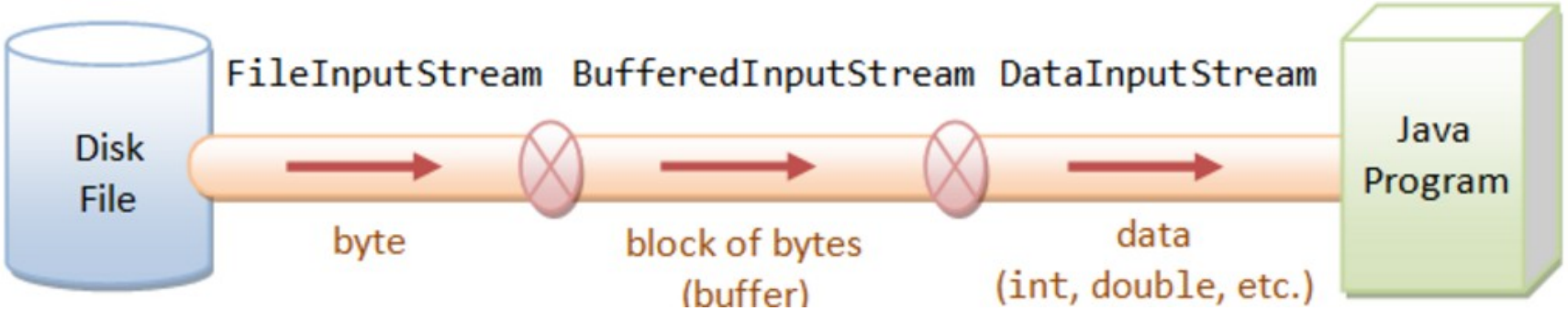
JAVA FILTER

- `InputStream` and `OutputStream` operano su "row bytes"
- classi filtro compiono trasformazioni sui dati a basso livello. Tipi di filtri:
- filter Stream: trasformazioni effettuate
  - crittografia
  - compressione
  - buffering
  - traduzione dei dati in un formato a più alto livello
- Readers/Writes
  - orientati al testo e permettono di decodificare bytes in caratteri
- I filtri possono essere organizzati in catena. Ogni elemento della catena
  - riceve dati dallo stream o dal filtro precedente
  - passa i dati al programma o al filtro successivo

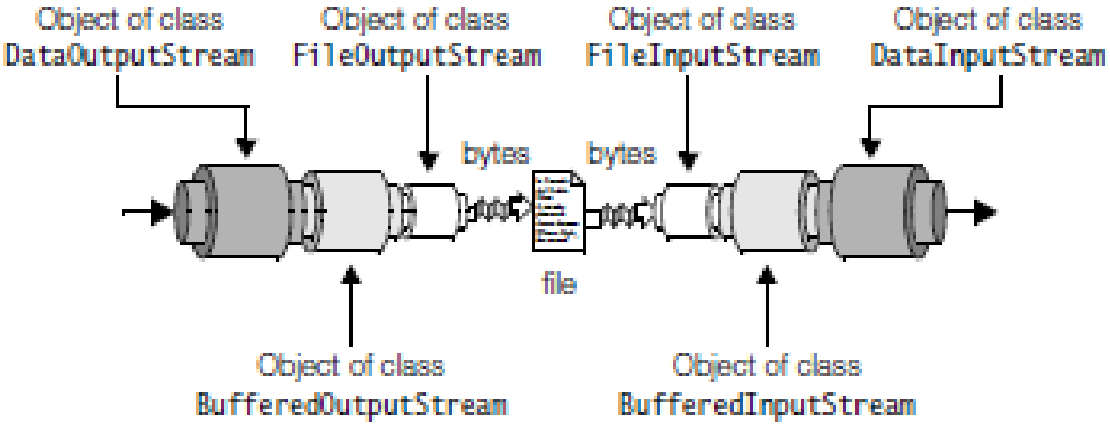
JAVA FILTER

- implementano una bufferizzazione per stream di input e di output,
- i dati vengono scritti e letti in blocchi di bytes, invece che un solo blocco per volta
- miglioramento significativo della performance





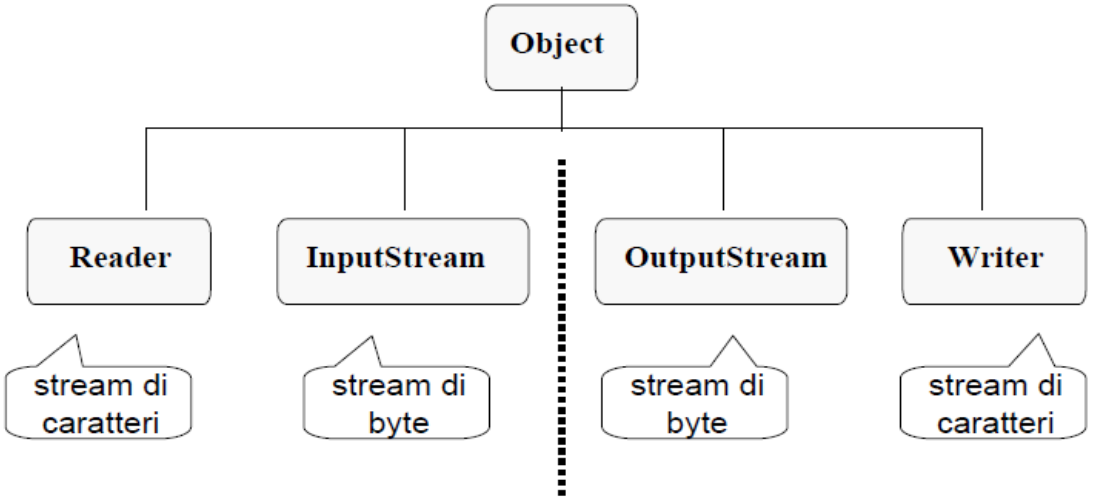
JAVA FILTER



- nell'esempio
  - stream di base è il `FileOutputStream`
  - viene "avvolto" in un `BufferedOutputStream`: byte raggruppati in blocchi, migliori prestazioni
  - viene "avvolto" in un `DataOutputStream`: trasforma tipi di dato strutturati in byte

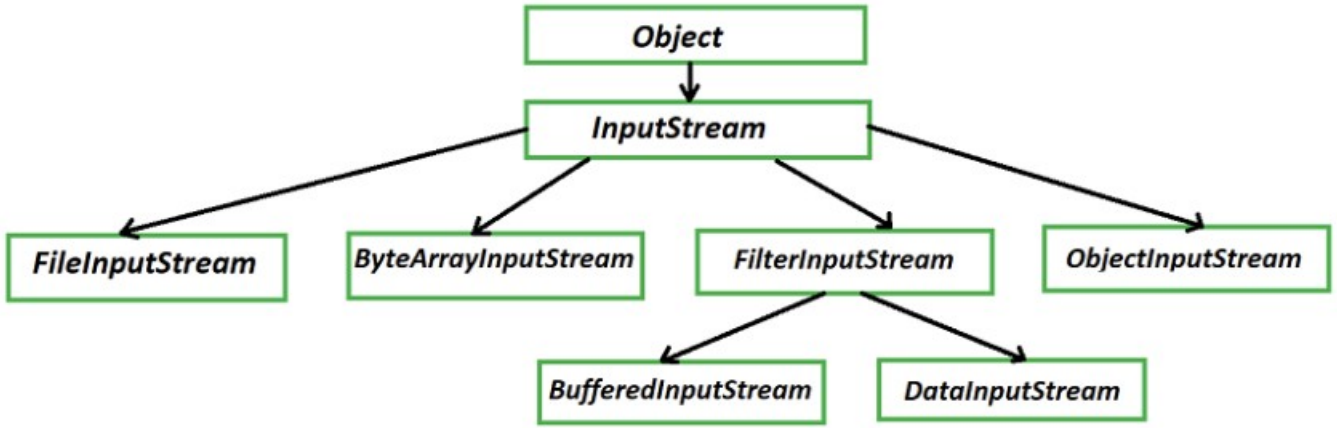
IL PACKAGE JAVA.IO

- `java.io` distingue fra:
  - stream di byte (analoghi ai file binari del C)
  - stream di caratteri (analoghi ai file di testo del C)
- modellate da altrettante classi base astratte:
  - stream di byte: `InputStream` e `OutputStream`
  - stream di caratteri: `Reader` e `Writer`
- i metodi sono simili per le due classi, per cui parleremo di stream di byte



STREAM DI BYTE

- la classe base `InputStream` definisce il concetto generale di "canale di input" che lavora a byte
  - il costruttore apre lo stream
  - `read()` legge uno o più byte
  - `close()` chiude lo stream
- `InputStream` è una classe astratta
  - il metodo `read()` dovrà essere realmente definito dalle classi derivate
  - un metodo specifico per ogni sorgente dati



STREAM DI BYTE: LEGGERE 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 può essere passato come parametro al costruttore di `FileInputStream`

```
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);
         }
```

- in alternativa si può passare al costruttore un oggetto `File` (o un `FileDescriptor`) costruito in precedenza

STREAM DI BYTE: LEGGERE DA FILE

- si usa il metodo `read()`
  - permette di leggere uno o più byte dal file
  - 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

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

STREAM DI BYTE: SCRIVERE SU FILE

- metodi analoghi per la apertura/scrittura 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`, o in alternativa si può passare al costruttore un oggetto `File` costruito in precedenza
- 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

JAVA: FILTER STREAMS

- `FilterInputStream` and `FilterOutputStream` con diverse sottoclassi
  - `BufferedInputStream` e `BufferedOutputStream` implementano filtri che bufferizzano l'input da/l'output verso lo stream sottostante
    - i dati vengono scritti e letti in blocchi di bytes, invece che un solo blocco per volta miglioramento significativo della performance
  - `DataInputStream` and `DataOutputStream` implementano filtri che permettono di " formattare" i dati presenti sullo stream

COPYING A FILE .JPEG

```
import java.io.*;
public class FileCopyNoBuffer{
   public static void main(String[] args) {
      String inFileStr  = "relax.jpg"; String outFileStr = "relax_new.jpg";
      long startTime, elapsedTime;  // for speed benchmarking
      File fileIn = new File(inFileStr);
      System.out.println("File size is " + fileIn.length() + " bytes");
      FileInputStream  in;  FileOutputStream out;
      try
          { in  = new FileInputStream(inFileStr);
            out = new FileOutputStream(outFileStr);
            startTime = System.nanoTime();
            int byteRead;
            while ((byteRead = in.read()) != -1)
                  { out.write(byteRead);}
            elapsedTime = System.nanoTime() - startTime;
            System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + "msec");
       }  catch (IOException ex) { ex.printStackTrace(); }}}
```

```
File size is 16473 bytes
Elapsed Time is 54.2873 msec
```

JAVA: FILTER STREAMS

cosa accade sostituendo

```
FileInputStream  in  = new FileInputStream(inFileStr);
FileOutputStream out = new FileOutputStream(outFileStr)
```

con

```
BufferedInputStream in = new BufferedInputStream(new
                                 FileInputStream(inFileStr));
BufferedOutputStream out = new BufferedOutputStream(new
                                 FileOutputStream(outFileStr)))
```

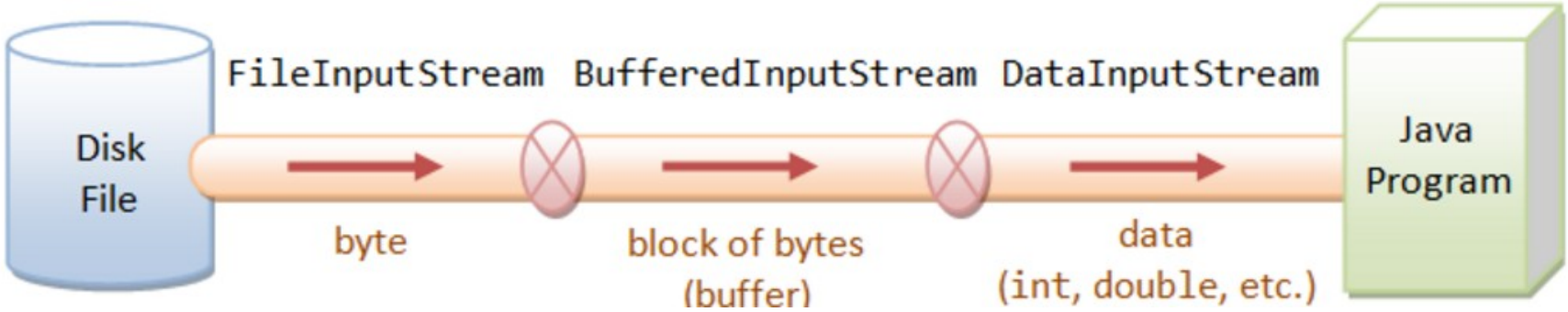
?

i tempi di esecuzione del programma si abbassano notevolmente

```
File size is 16473 bytes
Elapsed Time is 1.2581 msec
```

JAVA: FORMATTED DATA STERAM

```
mport java.io.*;
public class TestDataIOStream {
   public static void main(String[] args) {
      String filename = "data-out.dat";
      // Write primitives to an output file
      try (DataInputStream in =
              new DataInputStream(
                 new BufferedInputStream(
                    new FileInputStream(filename)))) {
```



JAVA: FORMATTED DATA STERAM

```
mport java.io.*;
public class TestDataIOStream {
   public static void main(String[] args) {
      String filename = "data-out.dat";
      // Write primitives to an output file
      try (DataInputStream in =
              new DataInputStream(
                 new BufferedInputStream(
                    new FileInputStream(filename)))) {
            System.out.println("byte:    " + in.readByte());
            System.out.println("short:   " + in.readShort());
            System.out.println("int:     " + in.readInt());
            System.out.println("long:    " + in.readLong());
            System.out.println("float:   " + in.readFloat());
            System.out.println("double:  " + in.readDouble());
            System.out.println("boolean: " + in.readBoolean());...}
```

ASSIGNMENT 4

- scrivere un programma che dato in input una lista di directories, comprima tutti i file in esse contenuti, con l'utility *gzip*
- ipotesi semplificativa:
  - zippare solo i file contenuti nelle directories passate in input,
  - non considerare ricorsione su eventuali sottodirectories
- il riferimento ad ogni file individuato viene passato ad un task, che deve essere eseguito in un threadpool
- individuare nelle `API JAVA` la classe di supporto adatta per la compressione
- `NOTA:` l'utilizzo dei threadpool è indicato, perchè I task presentano un buon mix tra I/O e computazione
  - I/O heavy: tutti i file devono essere letti e scritti
  - CPU-intensive: la compressione richiede molta computazione
- facoltativo: comprimere ricorsivamente I file in tutte le sottodirectories