Reti e Laboratorio 3
Modulo Laboratorio 3
AA. 2024-2025

docente: Laura Ricci

laura.ricci@unipi.it

Lezione 6

Stream based IO: richiami

25/10/2024

INPUT/OUTPUT IN JAVA

- I/O: reperire informazioni da una sorgente esterna o inviare ad una sorgente esterna
  - *file system*: files e directories
  - *connessioni di rete*
  - *keyboard*: `System.in`, `System.out`, `System.err`
  - *in-memory buffers* (array)
    - "vista" di un buffer di memoria come una sorgente o destinazione esterna.
      - un programma legge da un file csv.
      - per ottimizzare l'accesso ai dati si legge tutto il file in un buffer, in memoria centrale.
      - l'interfaccia verso il modulo che gestisce i dati deve rimanere la solita

- di particolare interesse per il corso:
  - connessioni di rete modellate come streams
  - in-memory buffers per la generazione di pacchetti UDP.

JAVA I/O

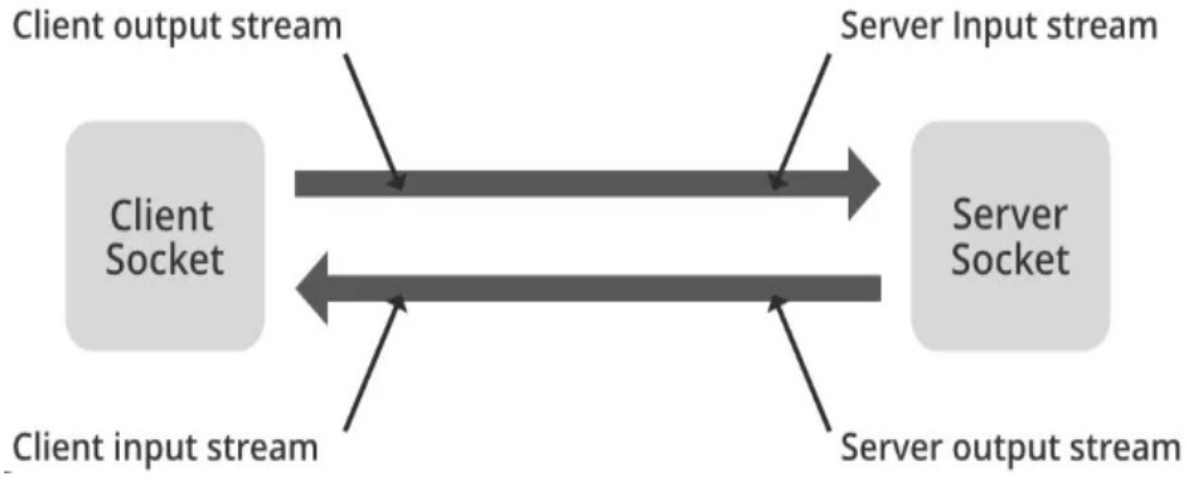
- definizione di un insieme di astrazioni per la gestione dell'I/O: una delle parti più complesse di un linguaggio

- 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)` (non lo vedermo)
- perchè importanti per questo corso? Le connessioni `TCP` possono essere modellate in `JAVA` con streams

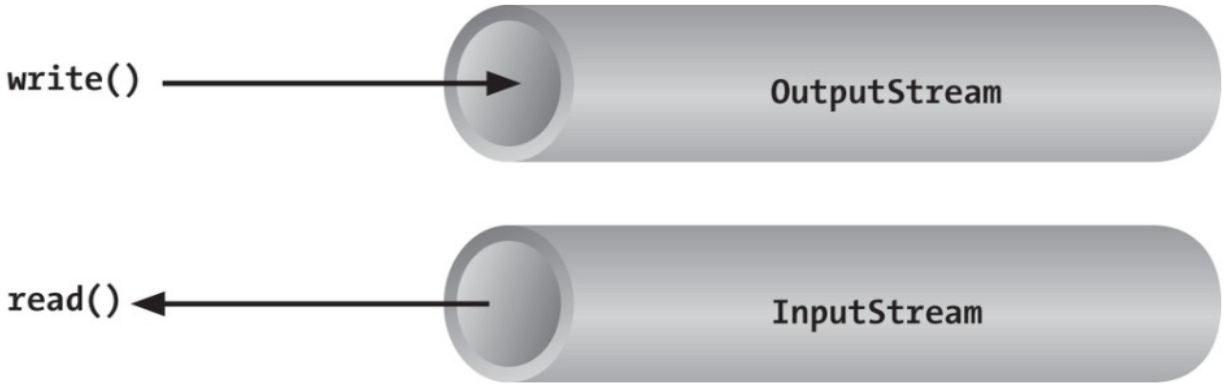
STREAM E RETI



- 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/legge dati ad/da un capo dello stream
- i dati fluiscono da/verso l'altra estremità

JAVA STREAMS: CARATTERISTICHE GENERALI

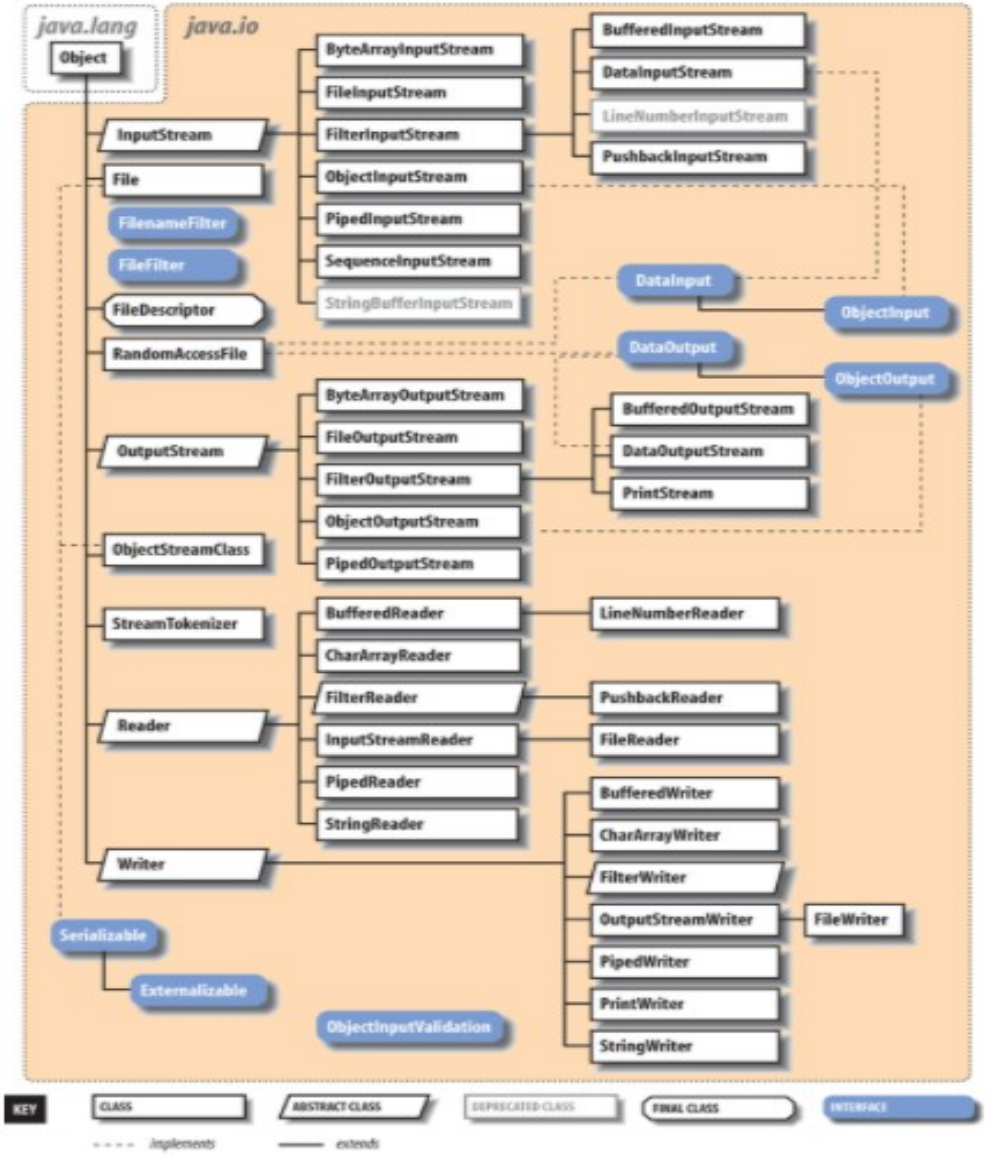
- accesso sequenziale
- mantengono l'ordinamento FIFO
- one way: read only oppure write only (a parte i file ad accesso random)
  - se un programma ha bisogno di dati in input ed output, è necessario aprire due stream, uno in input, l'altro in output
- bloccanti: quando un'applicazione legge un dato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

IL PACKAGE JAVA.IO: OBIETTIVI

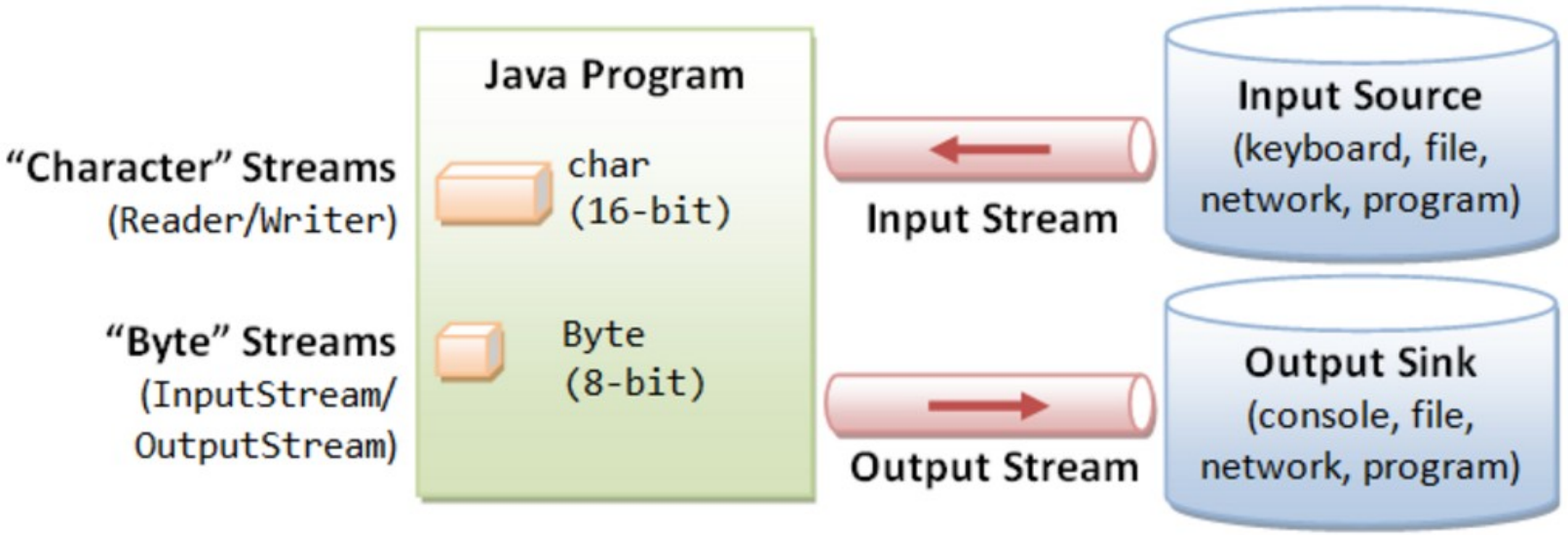
- fornire un'astrazione che incapsuli tutti i dettagli del dispositivo sorgente/ destinazione dei dati
- fornire un modo semplice e flessibile per aggiungere ulteriori funzionalità quelle fornite dallo "stream base"
- un approccio "a livelli"
  - alcuni stream di base per connettersi a dispositivi "standard": 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

LA GIUNGLA DELLE CLASSE IN JAVA IO

- 4 classi astratte fondamentali:
  - `InputStream, OutputStream, Reader, Writer`
- `Input/OutputStream`:
  - leggono e scrivono bytes
  - dati copiati byte a byte
    - non strutturati
    - senza alcuna traduzione
  - ideale per leggere/scrivere raw data in binario
    - un'immagine
    - la codifica di un video.
- `Reader/Writer` leggono/scrivono caratteri



LE CLASSI PRINCIPALI: CARATTERI E BYTE



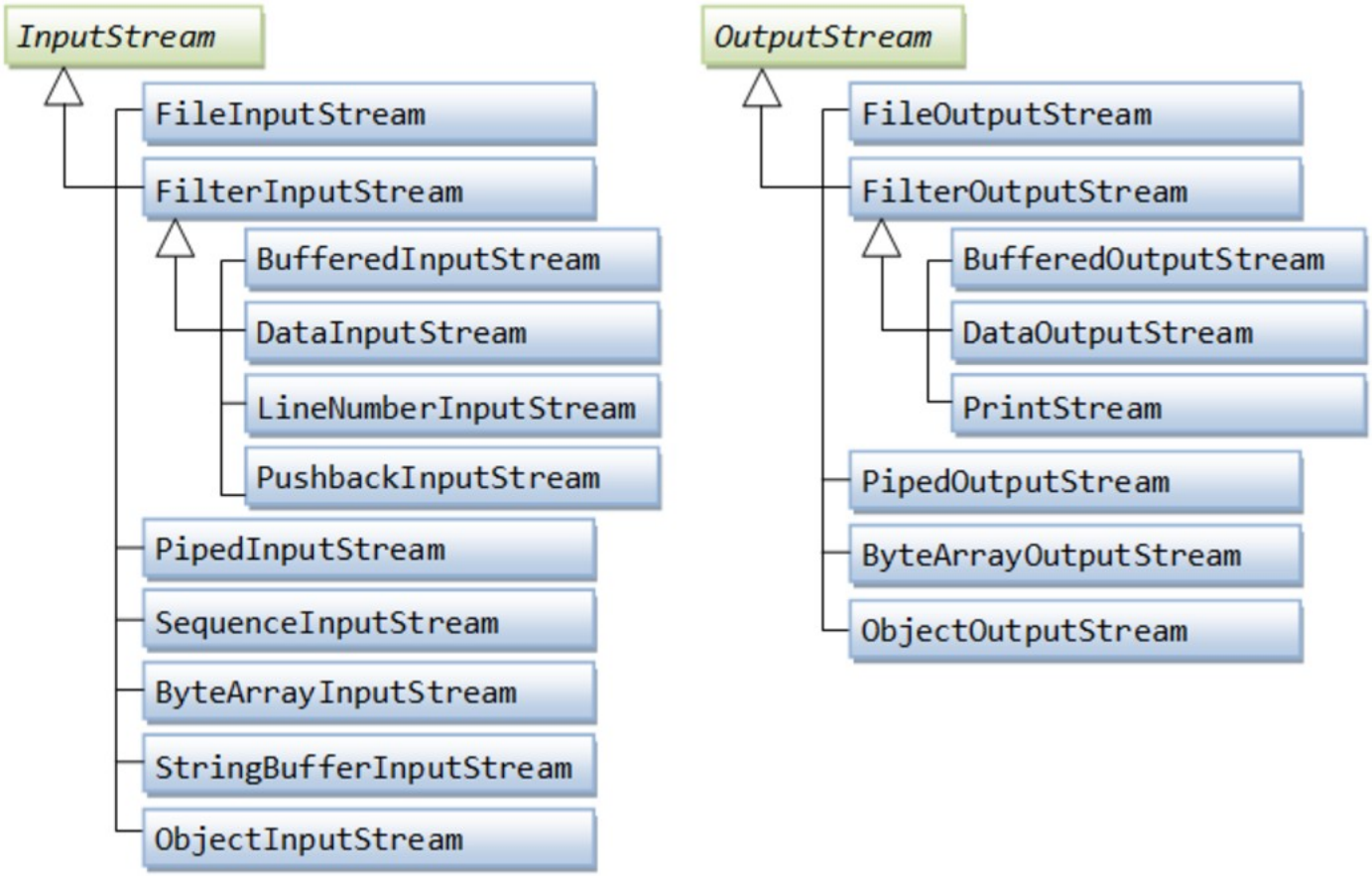
Internal Data Formats:

- 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)

BYTE STREAM: GERARCHIA DI CLASSI



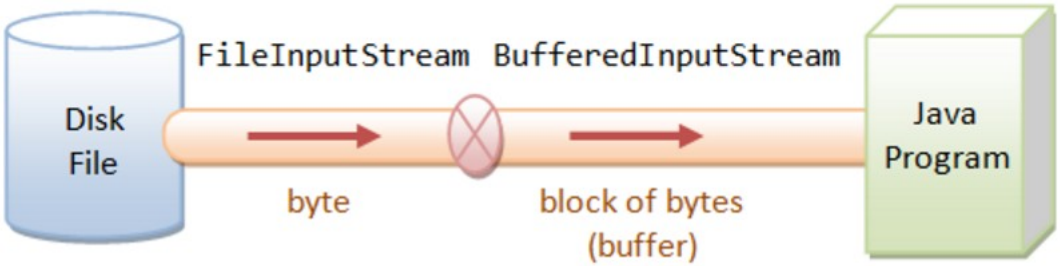
Input/OutputStream: sono classi astratte, diverse implementazioni

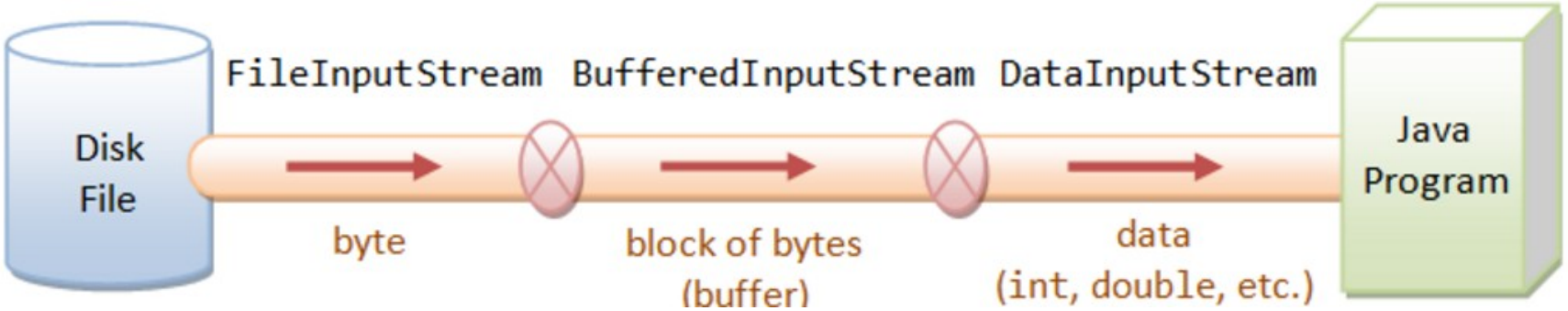
JAVA FILTER

- `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
- Reader/Write
  - 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

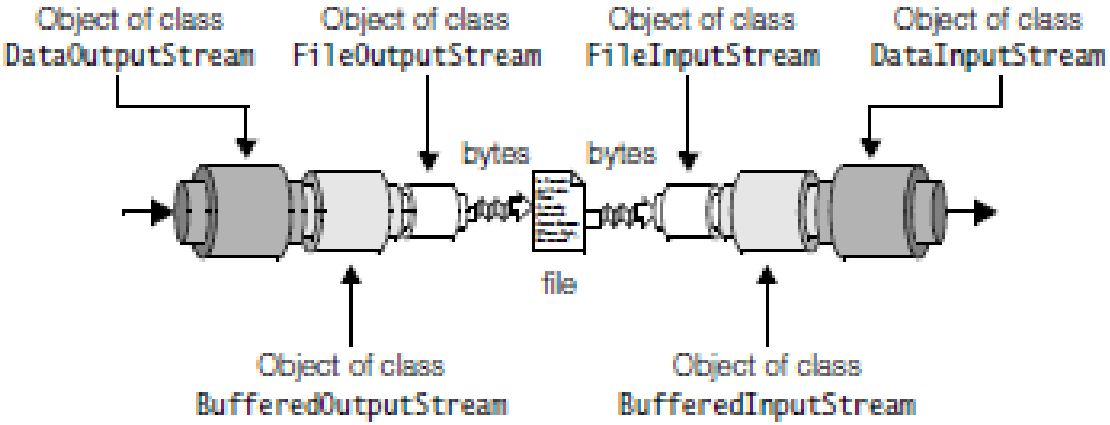
JAVA FILTER

- 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





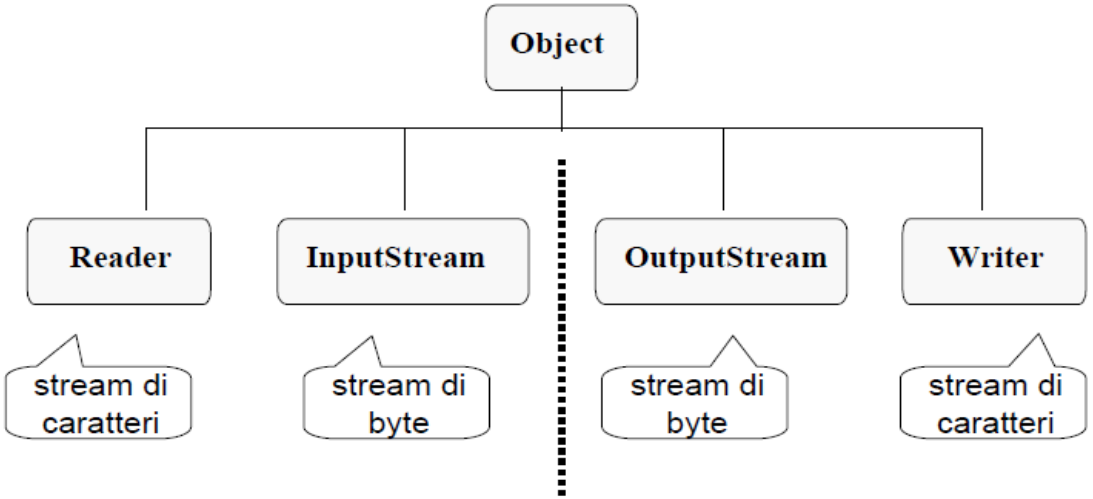
JAVA FILTER



- 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

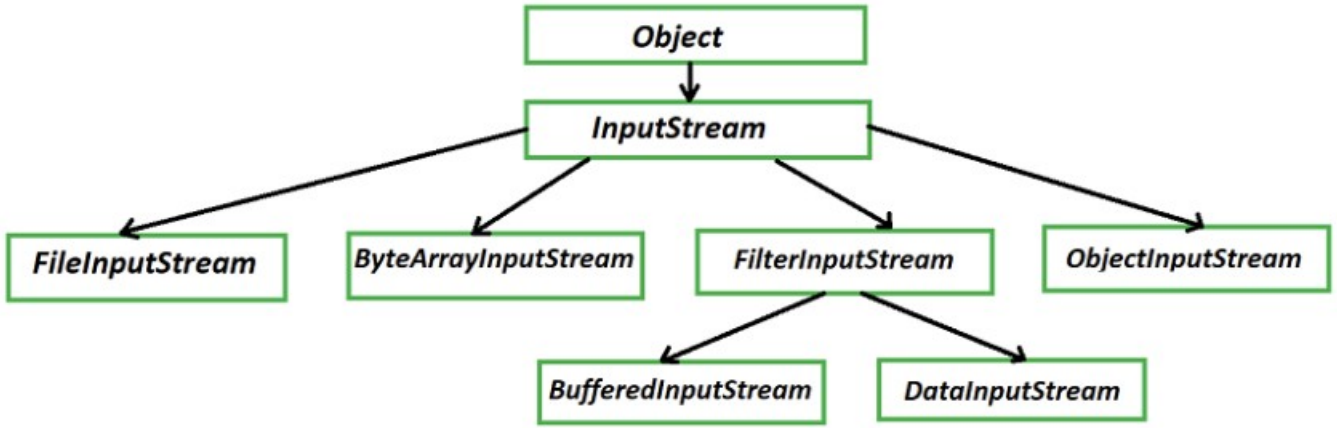
IL PACKAGE JAVA.IO

- `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



LA CLASSE FILE: DESCRITTORE DI FILE

un' istanza della classe `File` descrive:

- path per l'individuazione del file o della directory
- non una semplice stringa, ma offre metodi
  - per verificare l'esistenza del path
  - per restituire meta-informazioni sul file,…
- quando si vuole stabilire una connessione (stream) con un file si può passare come parametro:
  - un oggetto di tipo File
  - una stringa
  - …..

LA CLASSE FILE: DESCRITTORE DI FILE

```
public class ListFiles {
    public static void main(String[] args) {
        File dir = new File(".");   // current working directory
        if (dir.isDirectory()) {
            // List only files that meet the filtering criteria
            String[] files = dir.list();
            for (String file : files) {
                    if (file.endsWith(".java"))
                      System.out.println(file);}
            }
        }
    }
```

STREAM DI BYTE: LEGGERE DA FILE

- `FileInputStream` è la classe derivata che rappresenta il concetto di sorgente di byte "agganciata" ad 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
- Attenzione: gli stream vanno chiusi quando non più utilizzati, le close possono essere omesse in questi esempi per mancanza di spazio

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 migliorando significativamente le 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)))
```

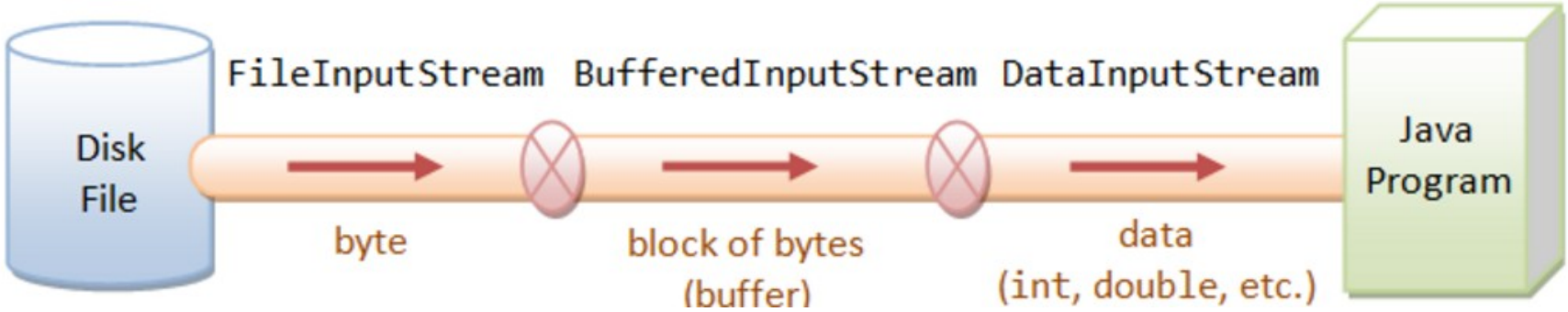
?

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());...}
```

TRY WITH RESOURCES

- introdotto in `JAVA 7`, aggiornato in `JAVA 9`
- chiusura sistematica ed automatica delle risorse di I/O usate da un programma
- un blocco `try` con uno o più argomenti tra parentesi.
  - argomenti: risorse che devono essere chiuse quando il `try block` termina
  - le variabili che rappresentano le risorse non devono essere riutilizzate
- generalizzazione: implementazione della `AutoCloseable` interface
- una soluzione al problema delle suppressed exceptions:
  - quando si verificano delle eccezioni sia nel blocco `try-with-resources` sia durante la chiusura della risorsa, solo l'ultima eccezione (quella generata in chiusura) verrebbe propagata
  - con il `try.. with resources`, la `JVM` sopprime l'eccezione generata nella chiusura automatica.

TRY WITH RESOURCES

- risorsa: file, stream, reader o socket
  - tecnicamente ogni oggetto che implementi l'interfaccia `AutoClosable`
- una certa risorsa viene chiusa "automaticamente", dopo che è stata utilizzata

```
try (FileWriter w = new FileWriter("file.txt")) {
     w.write("Hello World"); }
     // w.close() is called automatically
```

- in questo esempio, `w.close()` viene chiamata indipendentemente dal fatto che la `write` sollevi o meno una eccezione
- concettualmente simile ad aggiungere `w.close()` in un blocco `finally`
- possibile usare più risorse in un blocco `try with resources`, vengono chiuse in senso inverso rispetto all'ordine con cui sono state dichiarate

TRY WITH RESOURCES: ECCEZIONI

- nel seguente esempio

```
try (FileWriter w = new FileWriter("file.txt")) {
     w.write("Hello World"); }
     // w.close() is called automatically
```

- una eccezione può essere sollevata nei seguenti statement
  - `new FileWriter("file.txt")`
  - `w.write("Hello World")`
  - implicitamente da `w.close()`
- eccezione sollevata nel costruttore: nessun oggetto da chiudere, si propaga la eccezione senza eseguire la `write()`

TRY WITH RESOURCES: ECCEZIONI

- nel seguente esempio

```
try (FileWriter w = new FileWriter("file.txt")) {
     w.write("Hello World"); }
     // w.close() is called automatically
```

- eccezione sollevata nella `write()` : viene invocato `w.close()`, poi si propaga l'eccezione

TRY WITH RESOURCES: ECCEZIONI

- nel seguente esempio

```
try (FileWriter w = new FileWriter("file.txt")) {
     w.write("Hello World"); }
     // w.close() is called automatically
```

- eccezione sollevata nella chiamata implicita alla `close()` : viene propagata questa eccezione

TRY WITH RESOURCES: SUPPRESSED EXCEPTIONS

- nel seguente esempio

```
try (FileWriter w = new FileWriter("file.txt")) {
     w.write("Hello World"); }
     // w.close() is called automatically
```

- cosa accade se la `w.write()` solleva un'eccezione ed anche la chiamata implicita alla `w.close()` la sollva?
- la prima eccezione "vince" sulla seconda e la seconda viene soppressa

TRY WITH RESOURCES: SUPPRESSED EXCEPTIONS

```
import java.io.*;
public class trywithresources
  { public static void main (String args[])throws IOException {
    try(FileInputStream input = new FileInputStream(new  File("immagine.jpg"));
        BufferedInputStream bufferedInput = new  BufferedInputStream(input))
        {
          int data = bufferedInput.read();
          while(data != -1){
             System.out.print((char) data);
              data = bufferedInput.read();
          }}}}
```

- risolve il problema delle "suppressed exceptions"
  - eccezioni possono essere sollevate nel blocco `try`, oppure nel blocco `finally`,
  - un'eccezione rilevata nella `finally` sopprimerebbe l'eccezione rilevata nel blocco `try`
- con il `try with resources` viene propagata l'eccezione rilevata nel blocco `try`

ASSIGNMENT 6

- 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