Reti e Laboratorio III
Modulo Laboratorio III
AA. 2025-2026

docente: Laura Ricci

laura.ricci@unipi.it

Lezione 6

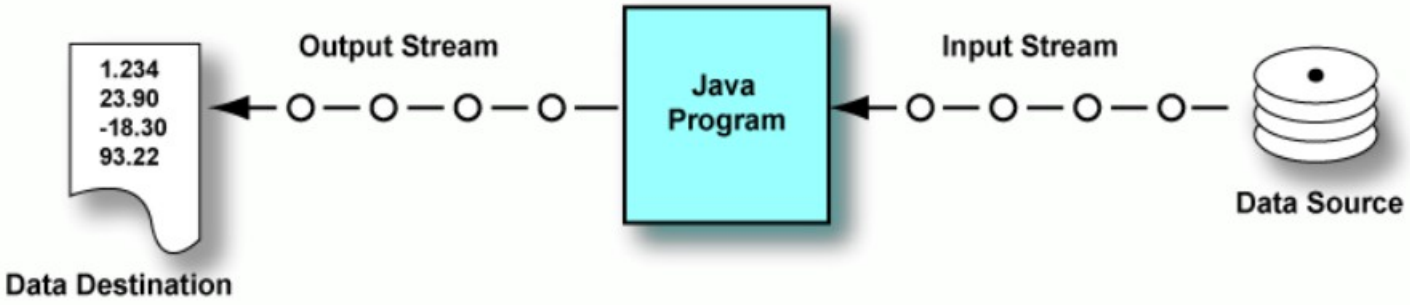
Stream based IO
Stream di byte e di caratteri
27/10/2025

INPUT/OUTPUT IN JAVA

- tipi di `I/O`:
  - *file system*: files e directories
  - *connessioni di rete*
  - *keyboard*: `System.in`
  - *console*: `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

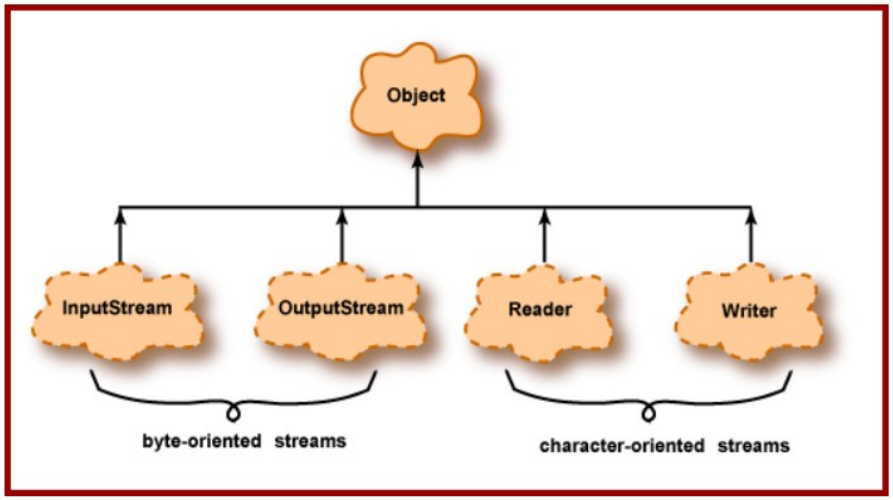
L'ASTRAZIONE DEGLI STREAM (RECAP)

- uno stream(flusso) rappresenta una a sequenza di dati che scorre da una sorgente a una destinazione



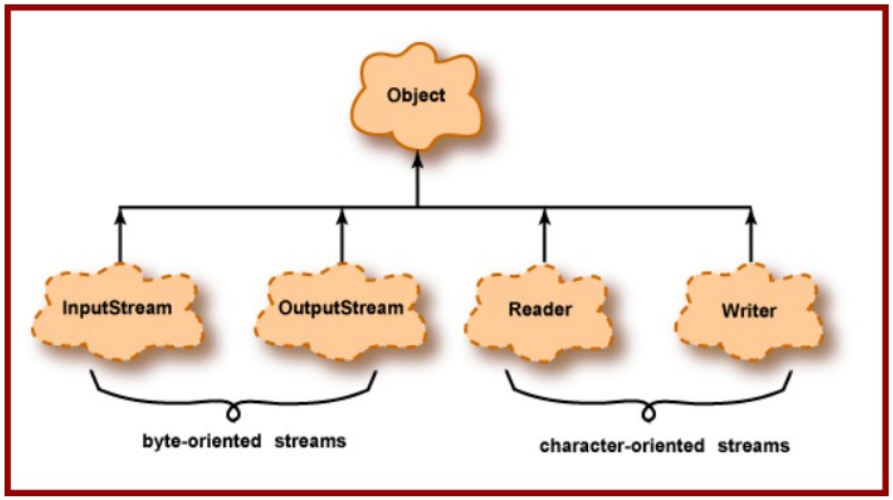
- nella figura, ogni cerchio rappresenta un "chunk di dati", l'interpretazione di cosa sia un chunck dipende dal tipo di stream
- principali proprietà degli stream
  - accesso sequenziale
  - ordinamento `FIFO`
  - one way
  - bloccanti

IL PACKAGE JAVA IO



- il diagramma mostra il livello più alto della gerarchia del pacchetto `java.io`
  - "nuvole tratteggiate" rappresentano classi astratte
  - fungono da classi base per gli stream specializzati, a seconda del dispositivo di input/output
- gli stream possono essere orientati ai byte o ai caratteri, e ciascun tipo comprende sia stream di input sia di output

IL PACKAGE JAVA IO



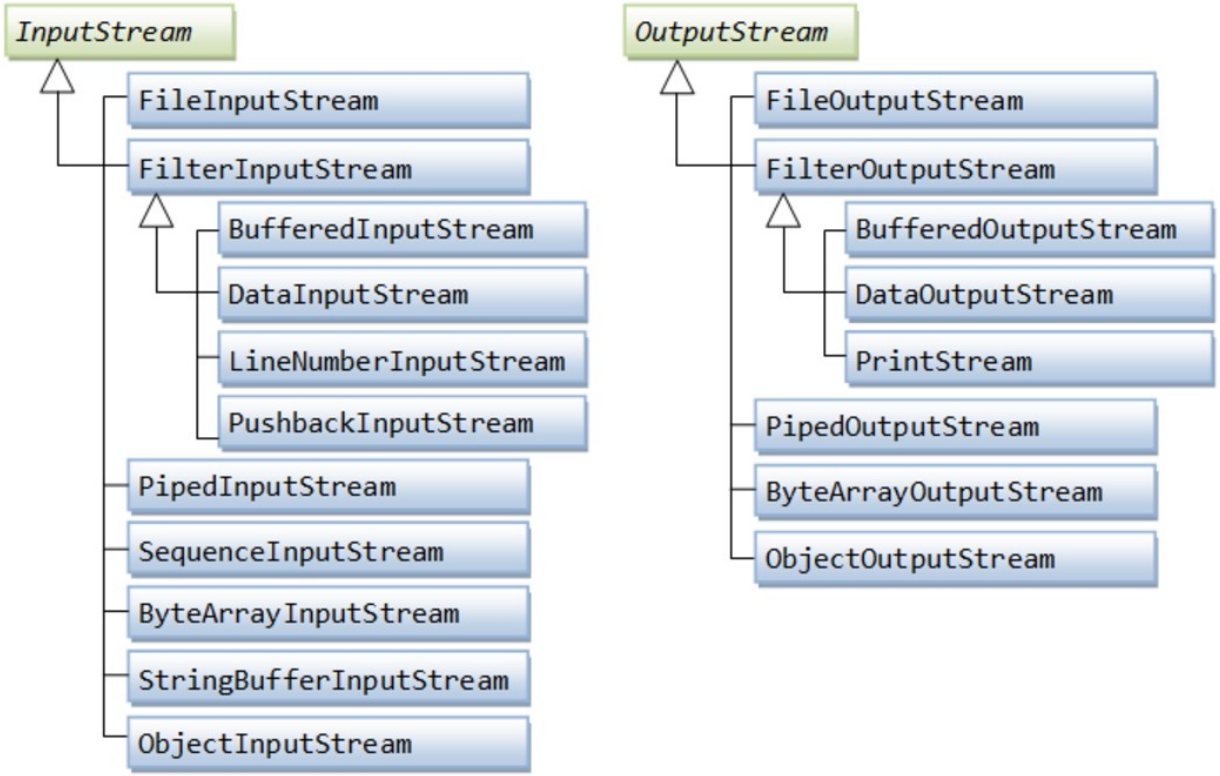
- stream orientati ai byte
  - possono gestire byte "grezzi"
    - dati non strutturati
    - letti/scritti byte a byte, senza alcuna traduzione
  - ideale per leggere/scrivere raw data: un'immagine, la codifica di un video
- stream orientati ai caratteri
  - destinati alla gestione di dati testuali
  - trasformano i dati tra il formato interno di `Java` (caratteri a `16` bit) e il formato esterno usato per la memorizzazione o il trasferimento

NOTA SUI PROGRAMMI SEGUENTI

- nei programmi presentati in seguito gli stream non vengono chiusi, per semplificare la presentazione
- tuttavia ogni stream deve essere chiuso, dopo essere stato utilizzato
- alla fine della lezione, vedremo un metodo elegante per chiudere automaticamente tutti gli stream utilizzati in una `try...catch`

STREAM DI BYTE

- nei programmi gli stream non vengono chiusi, per ragioni di semplificazione del programma mostrato
- tuttavia, gli stream vanno sempre chiusi, dopo averli utilizzati
- alla fine della lezioni vedremo un metodo elegante per chiudere automaticamente tutti gli stream utilizzati in un blocco try



STREAM DI BYTE: COPIARE UN FILE .JPEG

```
package FileCopyNoBuffer;
import java.io.*;
public class FileCopyNoBuffer{
   public static void main(String[] args) {
      String inFileStr  = "JAVA-Logo.jpg"; String outFileStr = "JAVA-Logo-new.jpg";
      long startTime, elapsedTime;  // for speed benchmarking
      InputStream  in; OutputStream out;
      int count = 0;
      try
          { in  = new FileInputStream(inFileStr);
            out = new FileOutputStream(outFileStr);
            startTime = System.nanoTime();
            int byteRead;
            while ((byteRead = in.read()) != -1)
                   { out.write(byteRead);
                     count++;}
            elapsedTime = System.nanoTime() - startTime;
            System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + "msec");
            System.out.println("File size"+count);}
        catch (IOException ex) { ex.printStackTrace();
        }}}
```

```
Elapsed Time is 147.1514 msec
File size 40084
```

STREAM DI BYTE: COPIARE UN FILE .JPEG

- la gerarchia delle classi

  `java.lang.Object`

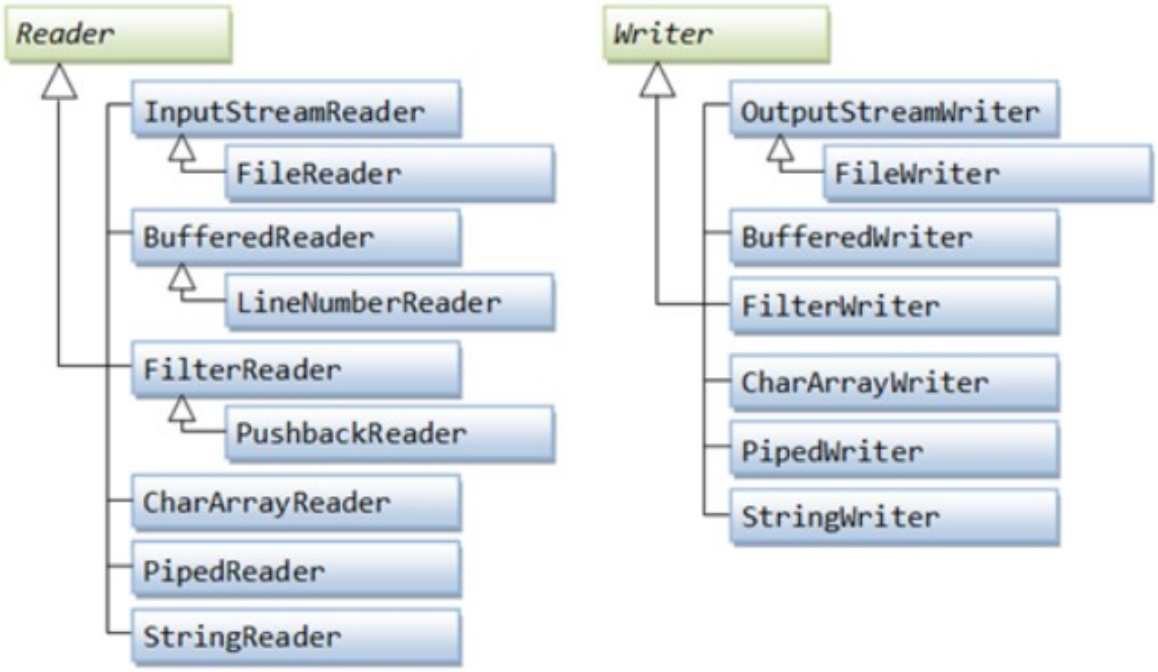
  ↳ `java.io.InputStream` (classe astratta)

  ↳ `java.io.FileInputStream`

- `FileInputStream` estende la classe astratta `InputStream` e fornisce implementazione concreta dei suoi metodi astratti
- ad esempio:
  - `read()`
    - legge un singolo byte dal file
    - restituisce un valore tra `0` e `255`, oppure `-1` se si è raggiunta la fine del file.
  - `read(byte[ ] b)`
    - legge una serie di byte e li inserisce nell'array `b` (fino a riempirlo, oppure una parte se il file termina)
    - restituisce il numero effettivo di byte letti, oppure `-1` se il file è terminato.
  - `close()`
    - chiude il file e rilascia le risorse di sistema

STREAM DI CARATTERI: READER E WRITER

- le classi base `Reader` e `Writer` definiscono il concetto generale di "canale di input/output", lavorano su caratteri, e sono classi astratte
- anche in questo caso, un metodo specifico per ogni sorgente dati



DA BYTE A CARATTERI

- un client in genere comunica con un server TCP scambiandosi testo, non byte grezzi
- tuttavia, quando si apre una Socket, si ottengono due flussi:

  `socket.getInputStream()` → `InputStream`, per leggere dati (in `byte`)

  `socket.OutputStream()` → `OutputStream` → per scrivere dati (in `byte`)
- poiché vogliamo leggere/scrivere testo, dobbiamo:
  - convertire l'`InputStream` in un `Reader` (`byte` → caratteri)
  - convertire l'`OutputStream` in un `Writer` (caratteri → `byte`)
- questo avviene usando delle "classi ponte"
  - `InputStreamReader` (per la ricezione)
  - `OutputStreamWriter` (per l'invio)

L'IMPORTANZA DELLA CODIFICA

- una codifica è la regola che mappa sequenze di byte ↔ caratteri `Unicode`.
- cosa è il sistema `Unicode`?
  - uno standard universale che assegna un numero univoco a ogni carattere di tutte le lingue del mondo, più simboli, emoji, ideogrammi, ecc.
  - questi numeri vengono chiamati code point e si scrivono in notazione U+xxxx
  - esempio: il carattere "a" corrisponde al numero `61` (`U+0061`), il carattere "焗" è il numero `3DB8` (`U+3DB8`), …
  - ma non è un formato di memorizzazione! definisce i numeri (i code point), ma non come vengono memorizzati nei file o trasmessi in rete
- per questo servono le codifiche (`encodings`) come `UTF-8`, `UTF-16`, `UTF-32`
- `Java` utilizza `UTF-16` come codifica interna dei caratteri
- soprattutto nelle applicazioni di rete, occorre considerare la possibilità di codifiche diverse quando un testo viene trasferito tra client e server

QUANDO UTF-8 INCONTRA ISO-8859-1… MOJIBAKE!

- Mojibake è un termine giapponese 文字化け
  - "moji" = carattere
  - "bake" = trasformazione
  - indica la visualizzazione errata di caratteri testuali quando la codifica usata per leggere un testo non corrisponde a quella usata per scriverlo
- creiamo un mojibake utilizzando
  - tcpbin.com: server TCP echo pubblico, per test e debug di connessioni TCP.
  - funzione principale: ritornare esattamente i dati ricevuti, byte per byte, al client che li invia
- il client JAVA che sviluppiamo utilizza
  - Reader/Writer
  - caratteri diversi in fase di invio della stringa al server echo e in fase di ricezione

QUANDO UTF-8 INCONTRA ISO-8859-1… MOJIBAKE!

```
package Mojibake;
import java.io.*;
import java.net.*;
import java.nio.charset.StandardCharsets;
public class MojibakeclientClient {
    public static void main(String[] args) {
        String host = "tcpbin.com";  // public echo server
        int port = 4242;             // echo port
        try (Socket socket = new Socket(host, port)) {
           System.out.println("Connected to " + host + ":" + port);
           // Reader using UTF-8 encoding
           BufferedReader reader = new BufferedReader(
               new InputStreamReader(socket.getInputStream(), StandardCharsets.UTF_8)
           );
           BufferedWriter writer = new BufferedWriter(
               new OutputStreamWriter(socket.getOutputStream(),StandardCharsets.ISO_8859_1)
```

QUANDO UTF-8 INCONTRA ISO-8859-1… MOJIBAKE!

```
// Send a UTF-8 encoded message

        String message = "Buongiorno! ☕ 🌸 Caffè e fiori rendono la giornata migliore.";
        writer.write(message + "\n");
        writer.flush();
        System.out.println("Sent: " + message);
        // Read the echo from the server
        String response = reader.readLine();
        System.out.println("Received: " + response);

    } catch (IOException e) {
        System.err.println("Error communicating with server:");
        e.printStackTrace();
    }
  }
}
```

Javadoc | Declaration | Console ×

<terminated> Utf8EchoClient [Java Application] C:\Users\ricci\.p2\pool\plugins\org.eclipse.justj.op

```
Connected to tcpbin.com:4242
Sent: Buongiorno! ☕ 🌸 Caffè e fiori rendono la giornata migliore.
Received: Buongiorno! ?? Caff? e fiori rendono la giornata migliore.
```

IL PACKAGE JAVA.IO: FILTRI

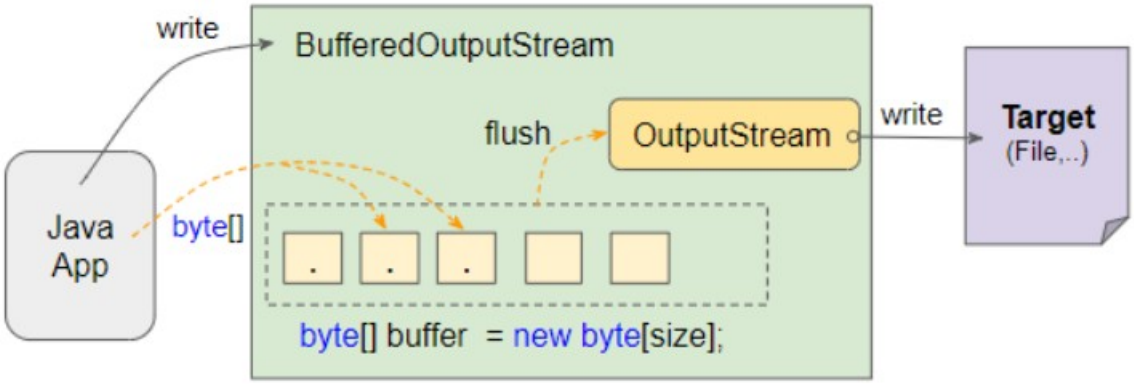
- obiettivi del package:
  - fornire un'astrazione che incapsuli tutti i dettagli del dispositivo sorgente/ destinazione dei dati
  - fornire un modo semplice e flessibile per aggiungere ulteriori funzionalità quelle fornite dallo "stream base"
- un approccio "a livelli"
  - alcuni stream di base per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

IL PACKAGE JAVA.IO: FILTRI

- `InputStream` and `OutputStream` operano su "raw bytes"
- classi filtro compiono trasformazioni sui dati a basso livello. Tipi di filtri:
- filter Stream: trasformazioni effettuate
  - buffering
  - traduzione dei dati in un formato a più alto livello
  - crittografia
  - compressione
- 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 BUFFERED OUTPUTSTREAM/WRITER

- filtri che implementano una bufferizzazione per stream di output,
- dati scritti a blocchi di bytes, invece che un byte per volta
- miglioramento significativo della performance

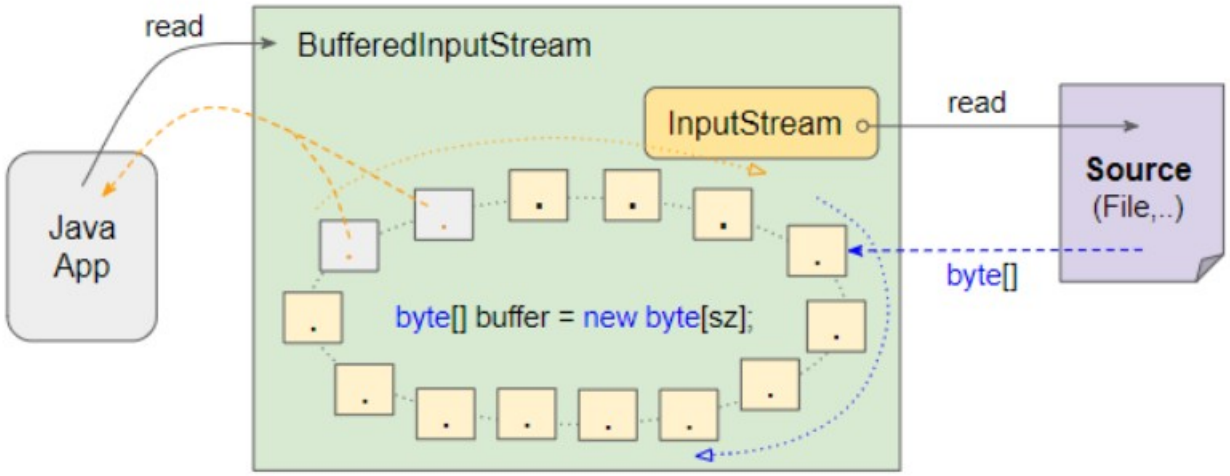



```
FileOutputStream outputFile = new FileOutputStream("output.data");

BufferedOutputStream bufferedOutput = new BufferedOutputStream(outputFile);
```

JAVA BUFFERED INPUTSTREAM/READER

- filtri che implementano una bufferizzazione per stream di input
- dati letti a blocchi di bytes, invece che un solo byte per volta
- miglioramento significativo della performance




```
FileInputStream outputFile = new FileInputStream("input.data");

BufferedInputStream bufferedOutput = new BufferedInputStream(outputFile);
```

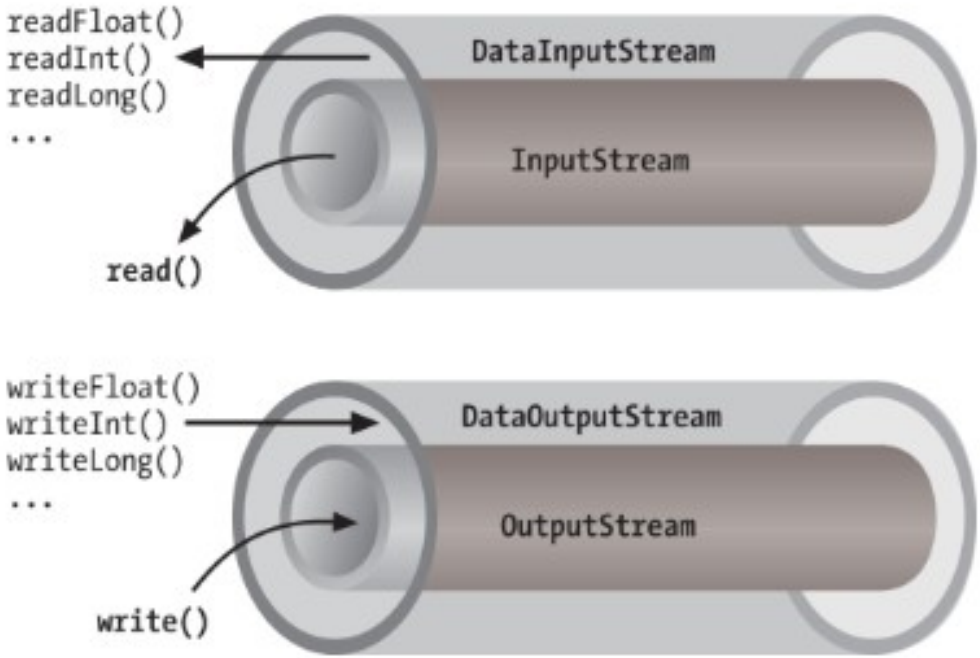
COPIARE UN FILE .JPEG: USARE BUFFERIZZAZIONE

```
package FileCopyNoBuffer;
import java.io.*;
public class FileCopyNoBuffer{
   public static void main(String[] args) {
      String inFileStr  = "JAVA-Logo.jpg"; String outFileStr = "JAVA-Logo-new.jpg";
      long startTime, elapsedTime;  // for speed benchmarking
      int count = 0;
      BufferedInputStream in;
      BufferedOutputStream out;
      try
          { in = new BufferedInputStream(new  FileInputStream(inFileStr));
            out = new BufferedOutputStream(new FileOutputStream(outFileStr));
            startTime = System.nanoTime();
            int byteRead;
            while ((byteRead = in.read()) != -1)
                   { out.write(byteRead); count++;}
            elapsedTime = System.nanoTime() - startTime;
            System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + "msec");
            System.out.println("file size"+count);}
            catch (IOException ex) { ex.printStackTrace(); }}}
```

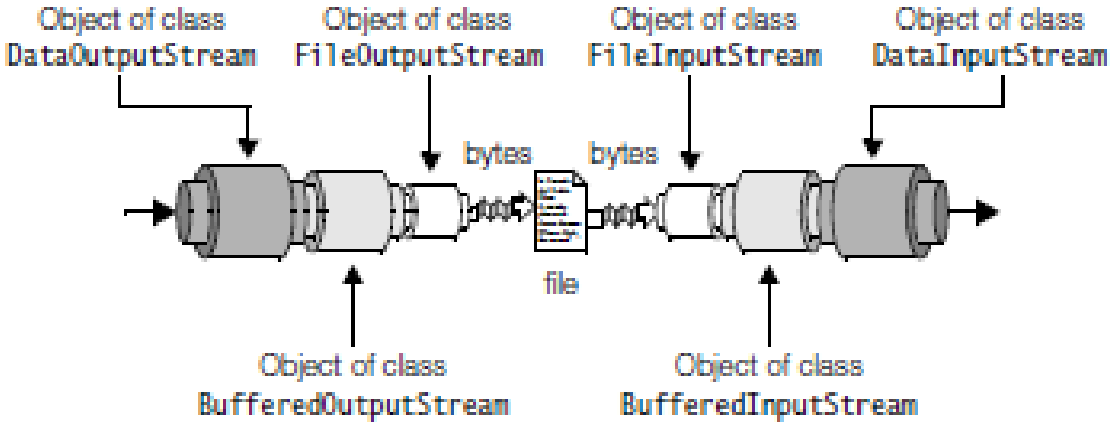
```
Elapsed Time is 11.0092msec
File size 40084
```

DATAINPUT/DATAOUTPUT STREAM

- servono per leggere/scrivere dati primitivi `Java` (`int`, `float`, `double`, `boolean`, `char`, ecc.) in formato binario, invece di testo.
- filtri che avvolgono (wrap) un flusso di byte (`InputStream` / `OutputStream`)



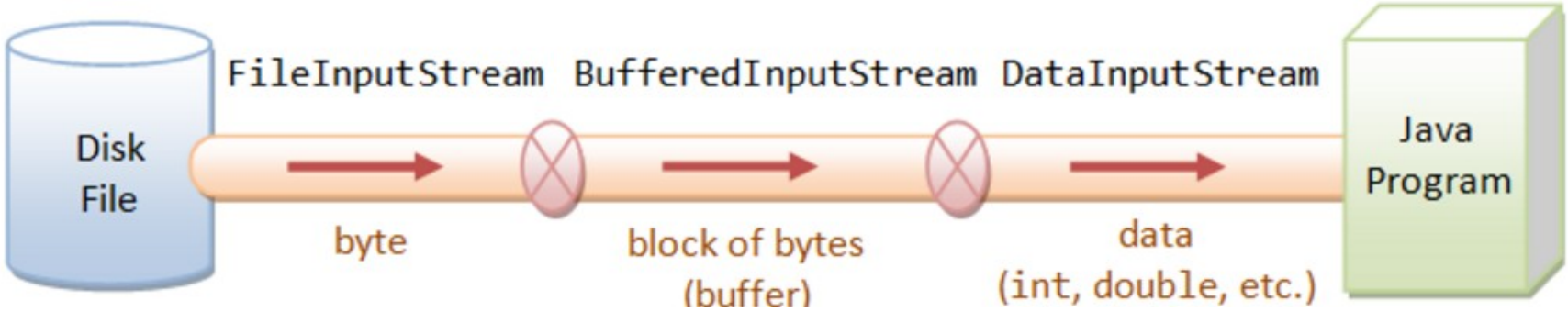
CONCATENARE I FILTRI



- nell'esempio
  - stream di base è il `FileInputStream`
  - viene "avvolto" in un `BufferedInputStream`: byte raggruppati in blocchi, migliori prestazioni
  - viene "avvolto" in un `DataInputStream`: trasforma i byte in tipi di dato strutturati

CONCATENARE I FILTRI

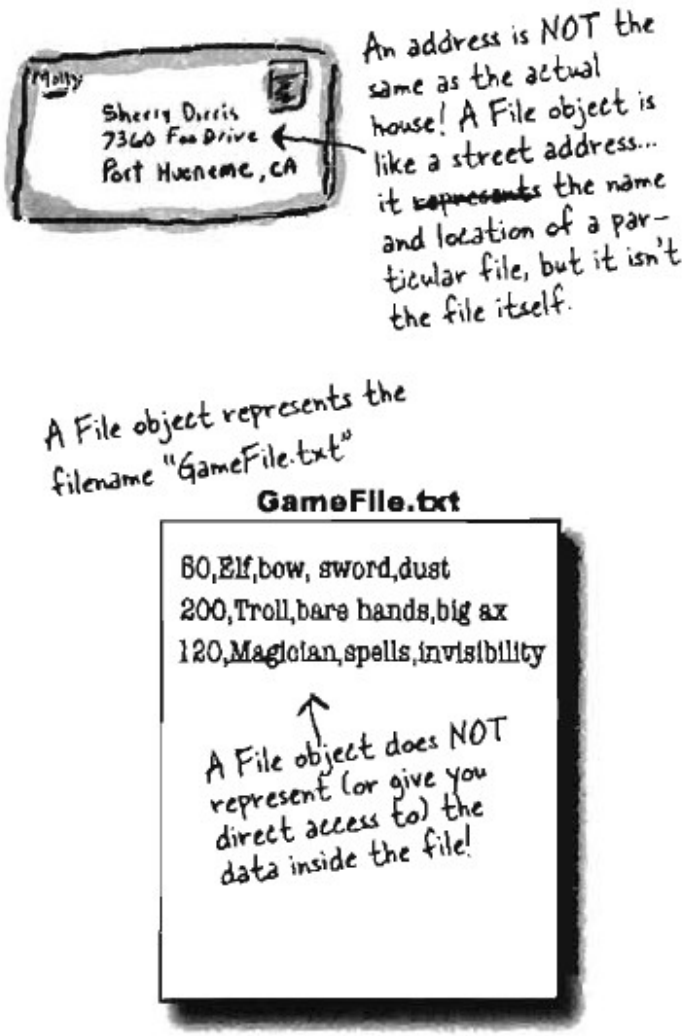
```
mport java.io.*;
public class TestDataIOStream {
   public static void main(String[] args) {
      String filename = "data-in.dat";
      // Write primitives to an output file
      try (DataInputStream in =
              new DataInputStream(
                 new BufferedInputStream(
           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



JAVA: FORMATTED DATA STERAM

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

UTILIZZARE I FILE: I DESCRITTORI



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:
  - una stringa, come visto in precedenza
  - un oggetto di tipo File
  - ...

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);}
            }
        }
    }
```

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` 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", alla fine del blocco

```
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 solleva?
- 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