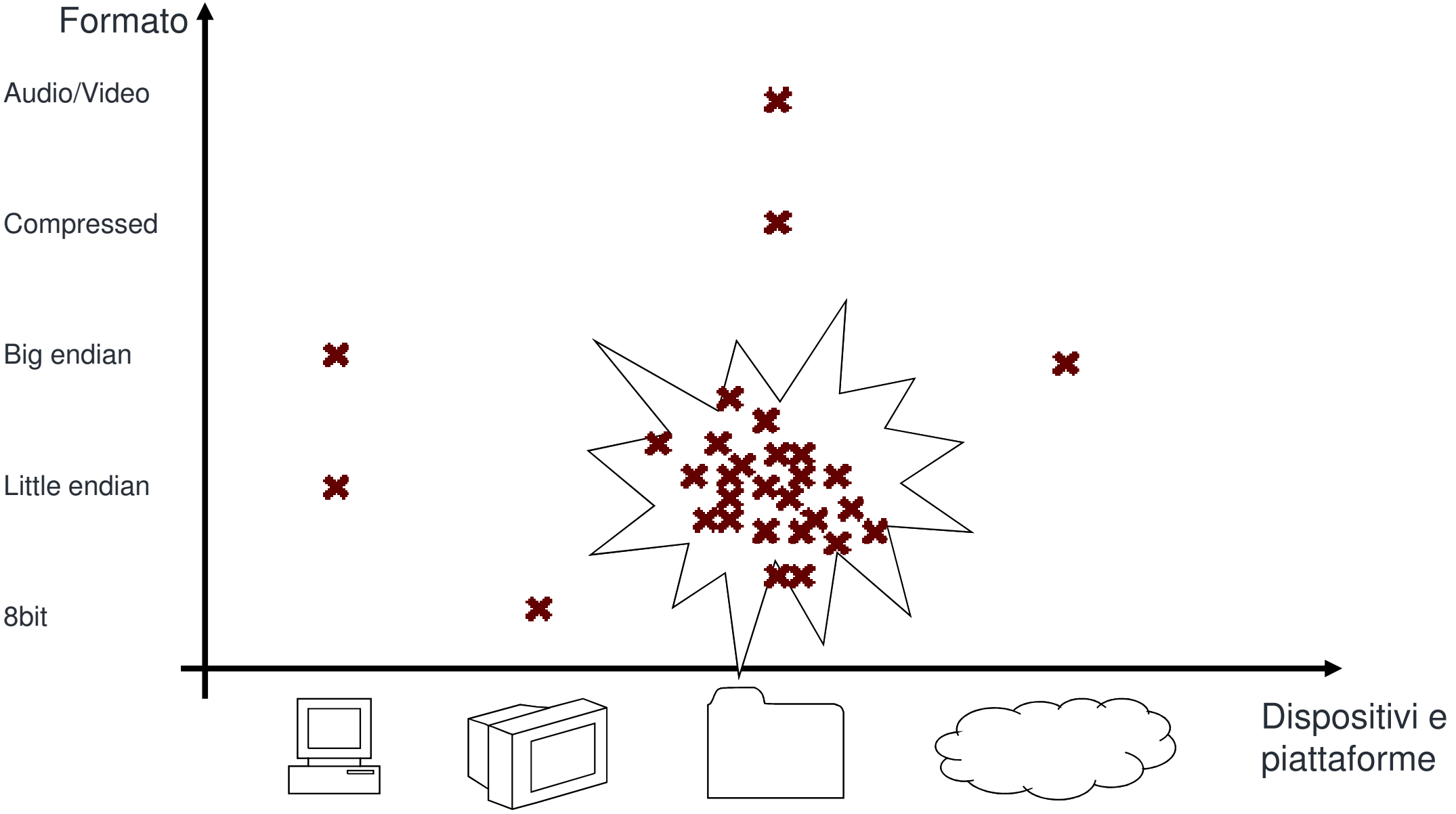
Programmazione Java

Davide Di Ruscio
Dipartimento di Informatica
Università degli Studi dell'Aquila

diruscio@di.univaq.it

Dimensioni nella gestione dell'I/O



Modalità di accesso ai dati

» Accesso sequenziale

L’accesso sequenziale si ha nel caso in cui i dati vengono letti o scritti dal dispositivo come una sequenza dove non è possibile tornare indietro

Dispositivi ad accesso sequenziale:

- Tastiera
- Modem
- Stampante
- Scheda audio

Modalità di accesso ai dati

» Accesso casuale (random/access)

L’accesso casuale si ha nel caso in cui è possibile muoversi all’interno dei dati memorizzati sul dispositivo e scrivere in una qualsiasi posizione

Dispositivi ad accesso sequenziale:

- Dischi
- Memorie

Approccio di Java

» I designer delle librerie della piattaforma *Java* hanno affrontato il problema della complessità della gestione dell'input/output creando una vasta gamma di classi

» Ogni classe incapsula le operazioni necessarie per gestire un tipo di dato o un dispositivo

» Componendo gli oggetti è possibile trattare tutte le dimensioni della gestione dell'input/output

Stream

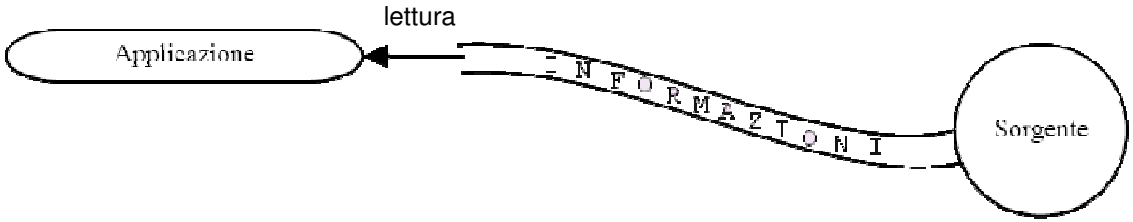
» La piattaforma Java utilizza il concetto di *stream* per trattare tutti i meccanismi di input/output

Uno *stream* è una rappresentazione mediante un oggetto di una qualsiasi sorgente di informazione capace di produrre o ricevere dati

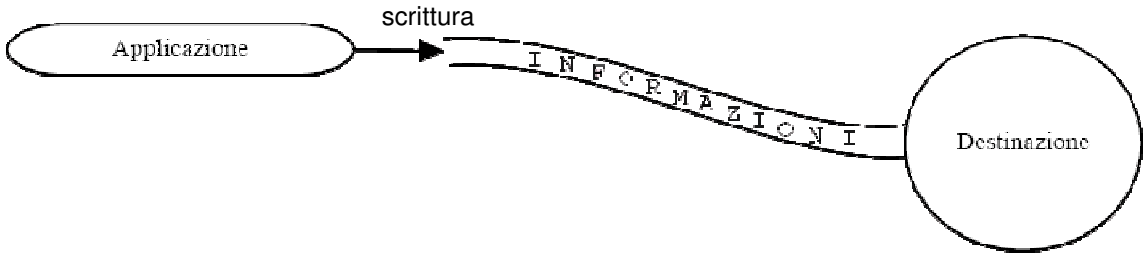
» Gli *stream* nascondono i dettagli di come vengono gestiti i dati dai reali dispositivi di input/output

Stream

» Per ricevere in ingresso dei dati, un programma apre uno stream su una sorgente di informazioni (file, memoria, connesioni di rete) e ne *legge sequenzialmente* le informazioni



» Analogamente un programma può inviare informazioni ad un destinatario, aprendo uno stream verso di esso e *scrivendo sequenzialmente* le informazioni in uscita



Stream: Reading e Writing

» Il processo di lettura (Reading) di informazioni può essere sintetizzato come segue:

```
open(stream)
while (more information)
 read(information)
close(stream)
```

» Similmente per il processo di scrittura (Wrinting):

```
open(stream)
while (more information)
 write(information)
close(stream)
```

Stream

» Gli *stream* vengono rappresentati e gestiti mediante:

- Interfacce
- Classi
- Eccezioni

» Gli *stream* gestiscono i dati secondo le regole dell'accesso sequenziale

Il package *java.io*

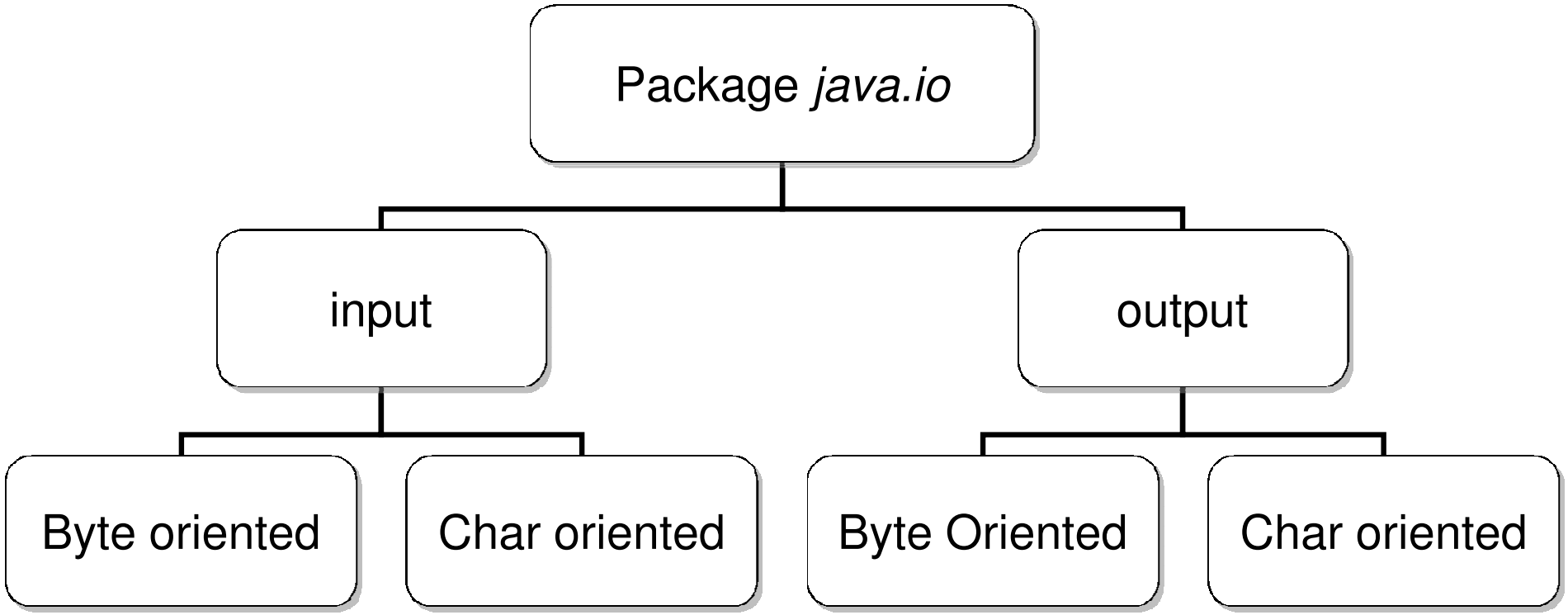
» Nella libreria standard della piattaforma *Java*, tutto ciò che riguarda la gestione degli *stream* è localizzato nel package *java.io*

» Tipicamente un programma che ha bisogno di utilizzare delle operazioni di input/output avrà nell'intestazione l'import di alcune classi presenti in tale package:

```
import java.io.*;
```

Il package *java.io*

» Le classi del package *java.io* sono logicamente suddivise a seconda della funzionalità che espletano e di come gestiscono a basso livello i dati



Il package *java.io*

» Byte oriented *stream*

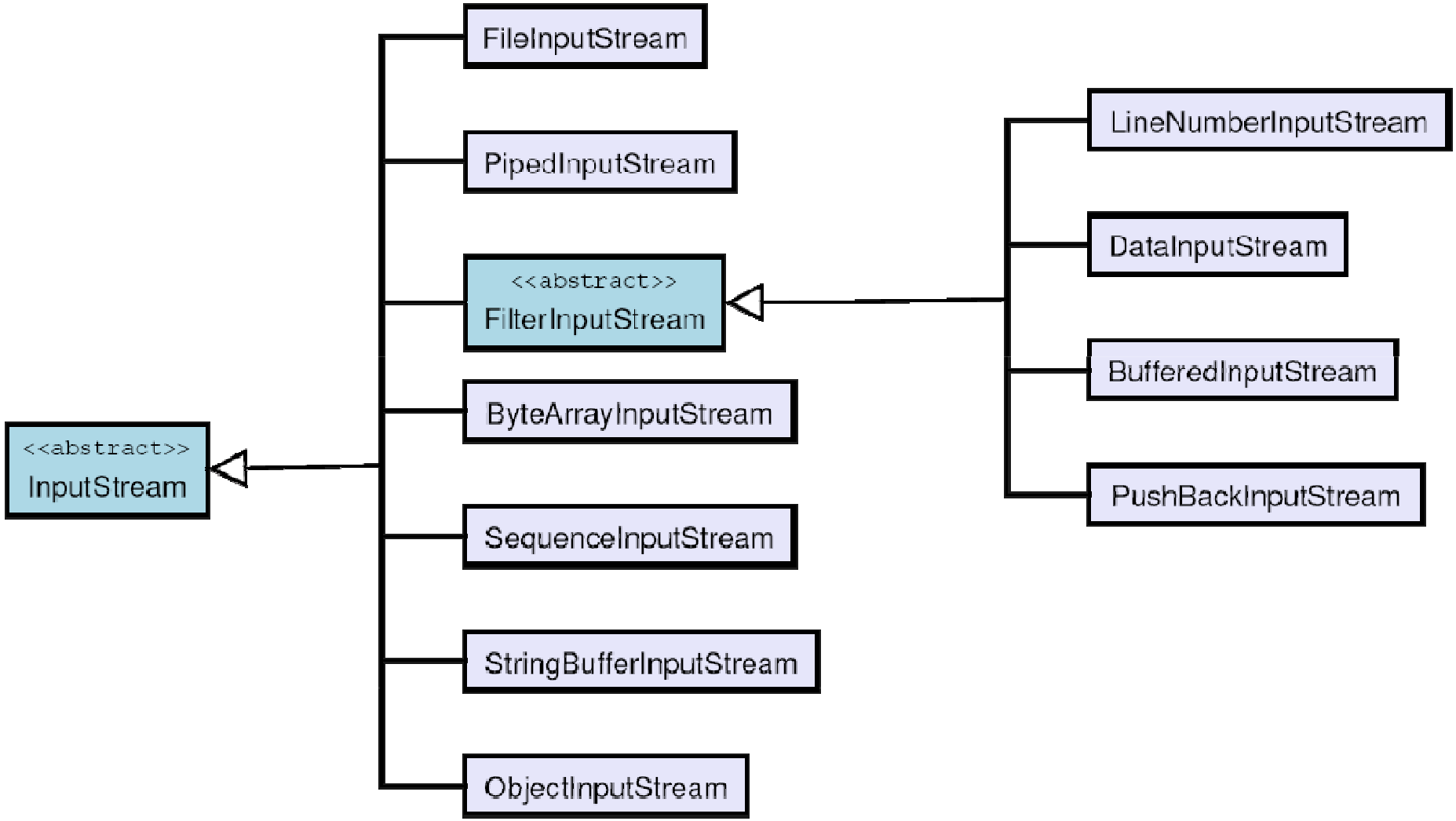
- Stream la cui unità atomica di memorizzazione è il byte.
- In questo caso si parla di I/O binario e viene usato in generale per i dati (es. i bit di un'immagine digitale o di un segnale sonoro digitalizzato).
- I flussi di byte sono divisi in flussi d'ingresso (**`InputStream`**) e flussi d'uscita (**`OutputStream`**)

» Char oriented *stream*

- Sono stati introdotti con la versione 2 della piattaforma e vengono utilizzati quando deve trattare del testo in formato *Unicode a 16bit*
- In questo caso si parla di I/O testuale (es. i caratteri ascii)
- I flussi di caratteri sono divisi in lettori (**`Reader`**) e scrittori (**`Writer`**)

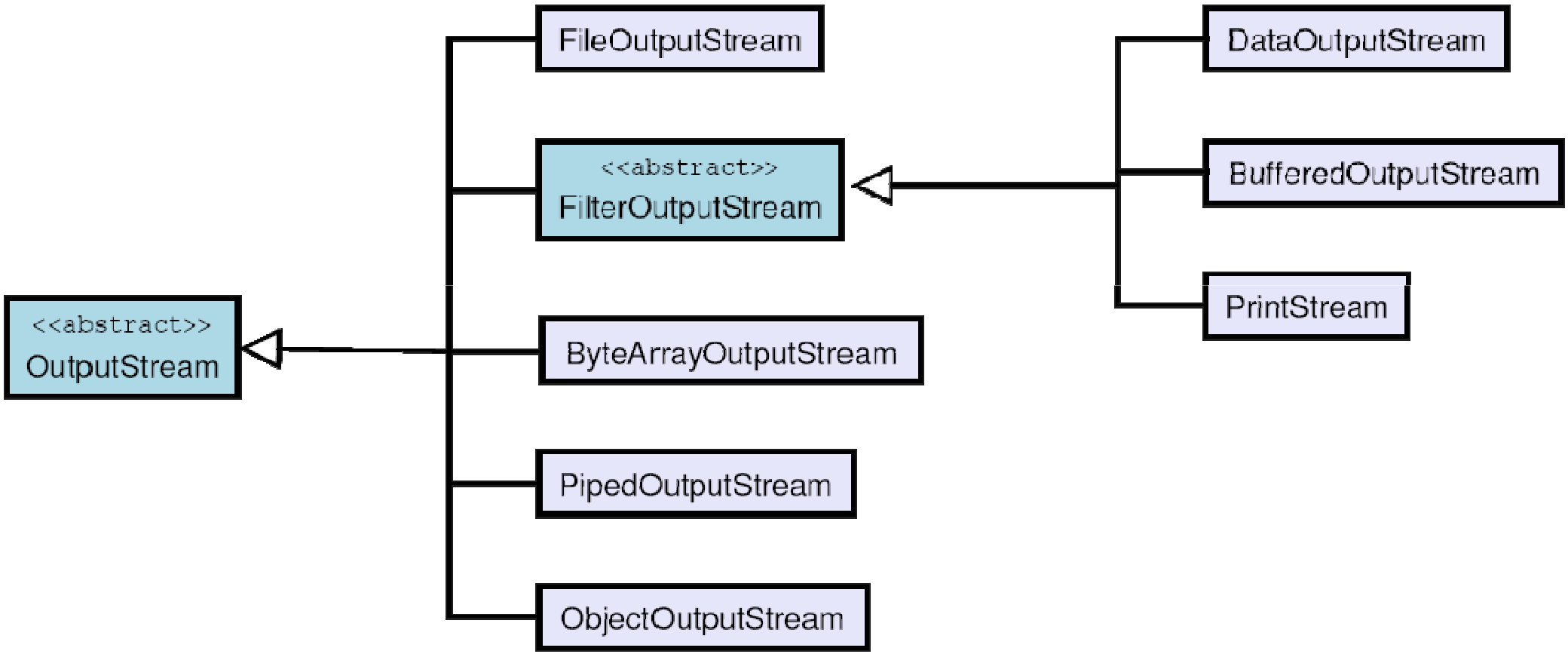
Classi per *stream* byte oriented

» La gerarchia **`InputStream`**



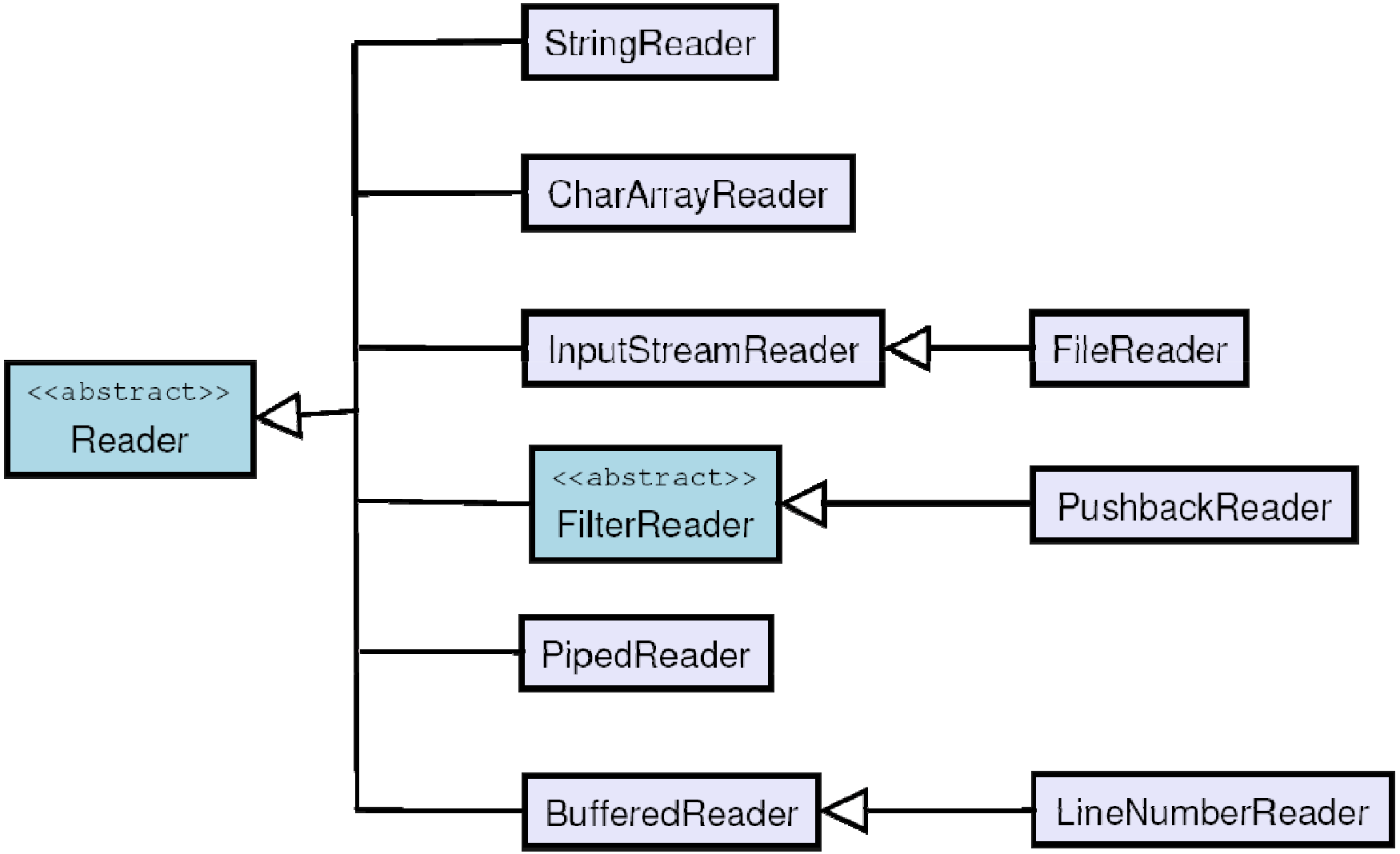
Classi per *stream* byte oriented

» La gerarchia **`OutputStream`**



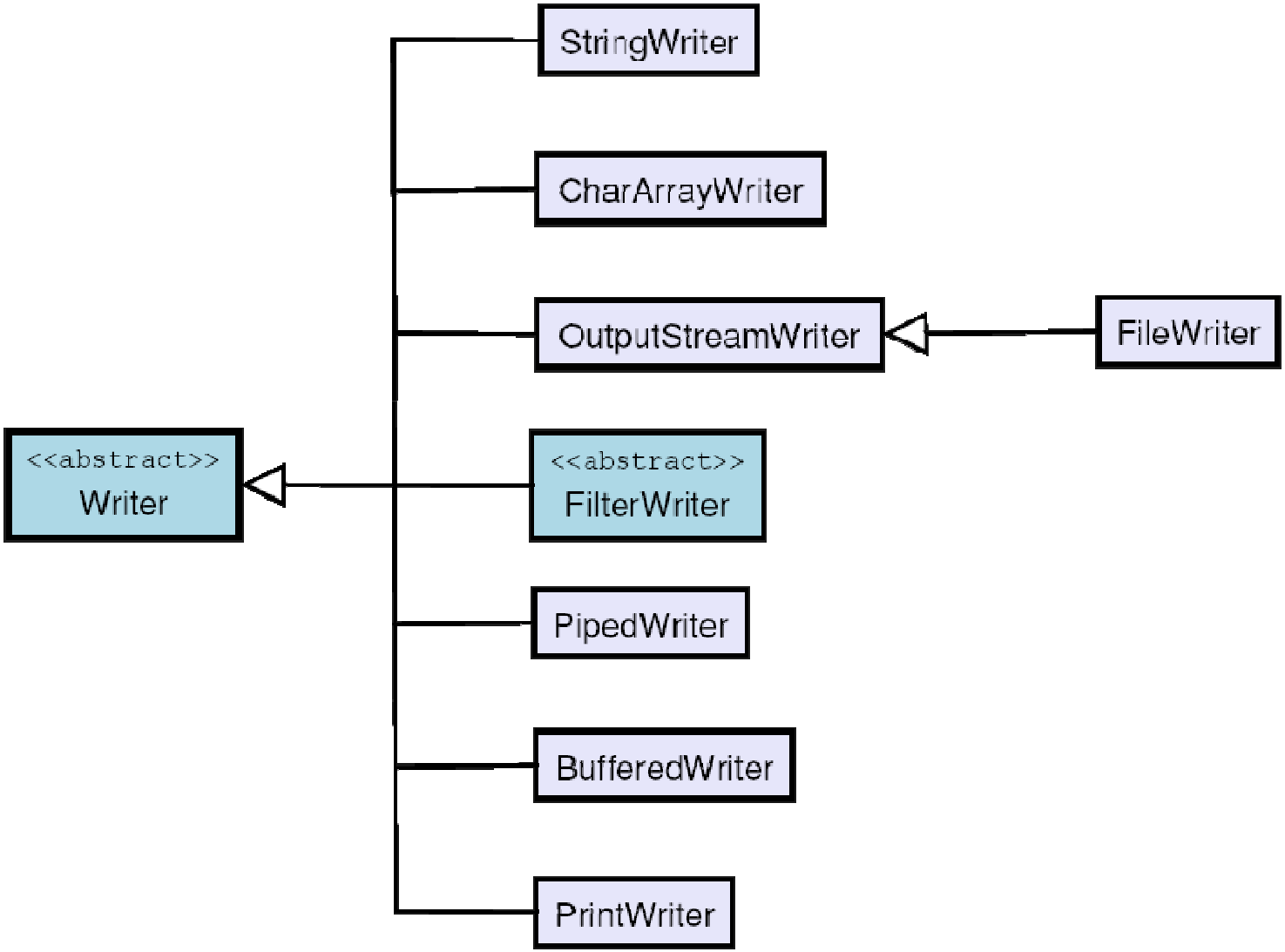
Classi per *stream* char oriented

» La gerarchia **`Reader`**



Classi per *stream* char oriented

» La gerarchia **`Writer`**



Utilizzi tipici di flussi di I/O

» Input bufferizzato da file

```
BufferedReader in = new BufferedReader(
      new FileReader("src/lezione13/IOStreamDemo.java"));
    String s, s2 = new String();
    while((s = in.readLine())!= null)
      s2 += s + "\n";
    in.close();
```

Si apre il file per l'input di caratteri

Utilizzi tipici di flussi di I/O

» Input bufferizzato da file

```
BufferedReader in = new BufferedReader(
      new FileReader("src/lezione13/IOStreamDemo.java"));
    String s, s2 = new String();
    while((s = in.readLine())!= null)
      s2 += s + "\n";
    in.close();
```

Per incrementare le prestazioni di lettura si usa BufferedReader

Utilizzi tipici di flussi di I/O

» Input bufferizzato da file

```
BufferedReader in = new BufferedReader(
      new FileReader("src/lezione13/IOStreamDemo.java"));
    String s, s2 = new String();
    while((s = in.readLine())!= null)
      s2 += s + "\n";
    in.close();
```

Ogni riga del file viene letta con il metodo readLine fino al raggiungimento della fine del file

Utilizzi tipici di flussi di I/O

» Input bufferizzato da file

```
BufferedReader in = new BufferedReader(
      new FileReader("src/lezione13/IOStreamDemo.java"));
    String s, s2 = new String();
    while((s = in.readLine())!= null)
      s2 += s + "\n";
    in.close();
```

Alla fine viene chiuso il file

Utilizzi tipici di flussi di I/O

» Input da memoria

```
StringReader in2 = new StringReader(s2);
    int c;
    while((c = in2.read()) != -1)
      System.out.print((char)c);
```

Viene creato uno StringReader a partire dalla stringa in s2

Utilizzi tipici di flussi di I/O

» Input da memoria

```
StringReader in2 = new StringReader(s2);
    int c;
    while((c = in2.read()) != -1)
      System.out.print((char)c);
```

Tutti i caratteri della stringa vengono letti uno di seguito all'altro mediante il metodo read() che restituisce il byte letto come un int.

Utilizzi tipici di flussi di I/O

» Input da memoria formattato

```
try {
      DataInputStream in3 = new DataInputStream(
        new ByteArrayInputStream(s2.getBytes()));
      while(true)
        System.out.print((char)in3.readByte());
    } catch(EOFException e) {
      System.err.println("End of stream");
    }
```

Il metodo getBytes() della classe String copia i caratteri della stringa in un array di byte. Ogni byte riceve gli 8 bit meno significativi del carattere corrspondente. Gli 8 bit più significativi vengono trascurati

Utilizzi tipici di flussi di I/O

» Input da memoria formattato

```
try {
      DataInputStream in3 = new DataInputStream(
        new ByteArrayInputStream(s2.getBytes()));
      while(true)
        System.out.print((char)in3.readByte());
    } catch(EOFException e) {
      System.err.println("End of stream");
    }
```

Viene creato un array di byte da poter poi gestire con …

Utilizzi tipici di flussi di I/O

» Input da memoria formattato

```
try {
      DataInputStream in3 = new DataInputStream(
        new ByteArrayInputStream(s2.getBytes()));
      while(true)
        System.out.print((char)in3.readByte());
    } catch(EOFException e) {
      System.err.println("End of stream");
    }
```

… la classe DataInputStream

Utilizzi tipici di flussi di I/O

» Input da memoria formattato

```
try {
      DataInputStream in3 = new DataInputStream(
        new ByteArrayInputStream(s2.getBytes()));
      while(true)
        System.out.print((char)in3.readByte());
    } catch(EOFException e) {
      System.err.println("End of stream");
    }
```

A questo punto è possibile leggere un byte alla volta dallo stream con il metodo readByte() della classe DataInputStream

Utilizzi tipici di flussi di I/O

» Output su file

```
try {
     BufferedReader in4 = new BufferedReader(
       new StringReader(s2));
     PrintWriter out1 = new PrintWriter(
       new BufferedWriter(new FileWriter("src/lezione13/IODemo.out")));
     int lineCount = 1;
     while((s = in4.readLine()) != null )
       out1.println(lineCount++ + ": " + s);
     out1.close();
   } catch(EOFException e) {
       System.err.println("End of stream");
   }
```

Viene creato un oggetti FileWriter per connettersi al file

Utilizzi tipici di flussi di I/O

» Output su file

```
try {
     BufferedReader in4 = new BufferedReader(
       new StringReader(s2));
     PrintWriter out1 = new PrintWriter(
       new BufferedWriter(new FileWriter("src/lezione13/IODemo.out")));
     int lineCount = 1;
     while((s = in4.readLine()) != null )
       out1.println(lineCount++ + ": " + s);
     out1.close();
   } catch(EOFException e) {
       System.err.println("End of stream");
   }
```

Si usa la bufferizzazione per migliorare le prestazioni delle operazioni di I/O

Utilizzi tipici di flussi di I/O

» Output su file

```
try {
     BufferedReader in4 = new BufferedReader(
       new StringReader(s2));
     PrintWriter out1 = new PrintWriter(
       new BufferedWriter(new FileWriter("src/lezione13/IODemo.out")));
     int lineCount = 1;
     while((s = in4.readLine()) != null )
       out1.println(lineCount++ + ": " + s);
     out1.close();
   } catch(EOFException e) {
       System.err.println("End of stream");
   }
```

Per la formattazione si usa PrintWriter. In questo modo il file creato è leggibile come un normale file di testo

Utilizzi tipici di flussi di I/O

» Output su file

```
try {
     BufferedReader in4 = new BufferedReader(
       new StringReader(s2));
     PrintWriter out1 = new PrintWriter(
       new BufferedWriter(new FileWriter("src/lezione13/IODemo.out")));
     int lineCount = 1;
     while((s = in4.readLine()) != null )
       out1.println(lineCount++ + ": " + s);
     out1.close();
   } catch(EOFException e) {
       System.err.println("End of stream");
   }
```

Lo stream viene letto una riga alla volta. Al termine dello stream di input, il metodo readLine() restituisce null.

Utilizzi tipici di flussi di I/O

» Output su file

```
try {
     BufferedReader in4 = new BufferedReader(
       new StringReader(s2));
     PrintWriter out1 = new PrintWriter(
       new BufferedWriter(new FileWriter("src/lezione13/IODemo.out")));
     int lineCount = 1;
     while((s = in4.readLine()) != null )
       out1.println(lineCount++ + ": " + s);
     out1.close();
   } catch(EOFException e) {
       System.err.println("End of stream");
   }
```

L'uso di stream bufferizzati non garantisce che il contenuto del buffer venga scritto fisicamente nel file nel momento stesso in cui I metodi di scrittura sono stati invocati. Il metodo close() forza la scrittura dei dati ancora in memoria e chiude il file.

Utilizzi tipici di flussi di I/O

» Memorizzazione e lettura dei dati

```
try {
      DataOutputStream out2 = new DataOutputStream(
        new BufferedOutputStream(
          new FileOutputStream("src/lezione13/Data.txt")));
      out2.writeDouble(3.14159);
      out2.writeUTF("That was pi");
      out2.writeDouble(1.41413);
      out2.writeUTF("Square root of 2");
      out2.close();
      DataInputStream in5 = new DataInputStream(
        new BufferedInputStream(
          new FileInputStream("src/lezione13/Data.txt")));
      // Must use DataInputStream for data:
      System.out.println(in5.readDouble());
      // Only readUTF() will recover the
      // Java-UTF String properly:
      System.out.println(in5.readUTF());
      // Read the following double and String:
      System.out.println(in5.readDouble());
      System.out.println(in5.readUTF());
    } catch(EOFException e) {
      throw new RuntimeException(e);
    }
```

- Per avere un output che possa essere letto da un altro stream è necessario usare DataOutputStream (e DataInputStream per leggerli)

- DataOutputStream e DataInputStream sono orientati ai byte quindi richiedono gli InputStream e OutputStrem

Utilizzi tipici di flussi di I/O

» Leggere e scrivere file ad accesso casuale

```
RandomAccessFile rf = new RandomAccessFile("src/lezione13/rtest.dat", "rw");
    for(int i = 0; i < 10; i++)
      rf.writeDouble(i*1.414);
    rf.close();
    rf = new RandomAccessFile("src/lezione13/rtest.dat", "rw");
    rf.seek(5*8);
    rf.writeDouble(47.0001);
    rf.close();
    rf = new RandomAccessFile("src/lezione13/rtest.dat", "r");
    for(int i = 0; i < 10; i++)
      System.out.println("Value " + i + ": " +
        rf.readDouble());
    rf.close();
```

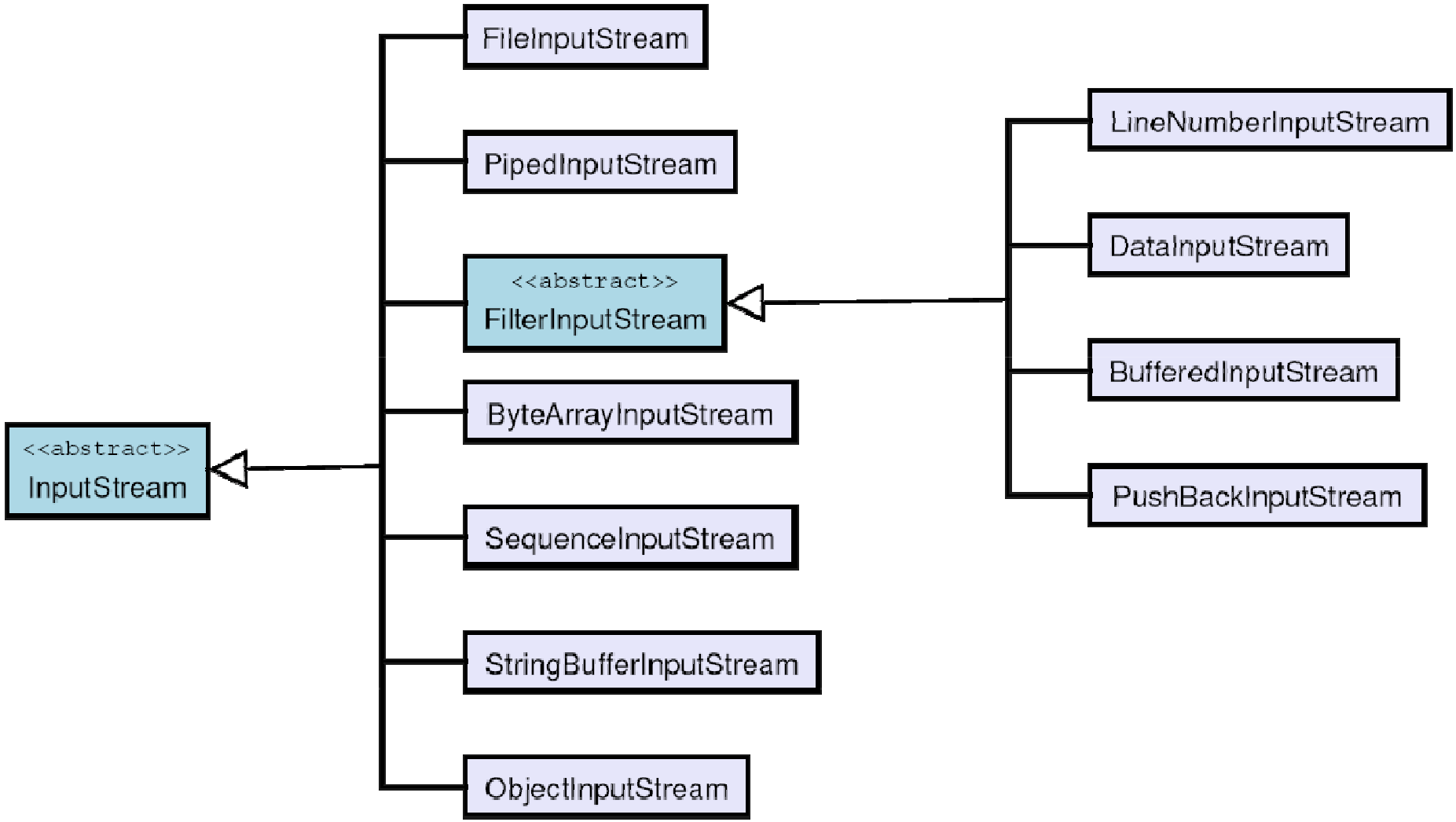
L'utilizzo di RandomAccessFile è simile all'utilizzo di una combinazione di un DataInputStream e di un DataOutputStrem

Vedere IOStreamDemo.java

Classi per la gestione degli stream byte oriented

Classi per *stream* byte oriented

» La gerarchia **`InputStream`**



La gerarchia `InputStream`

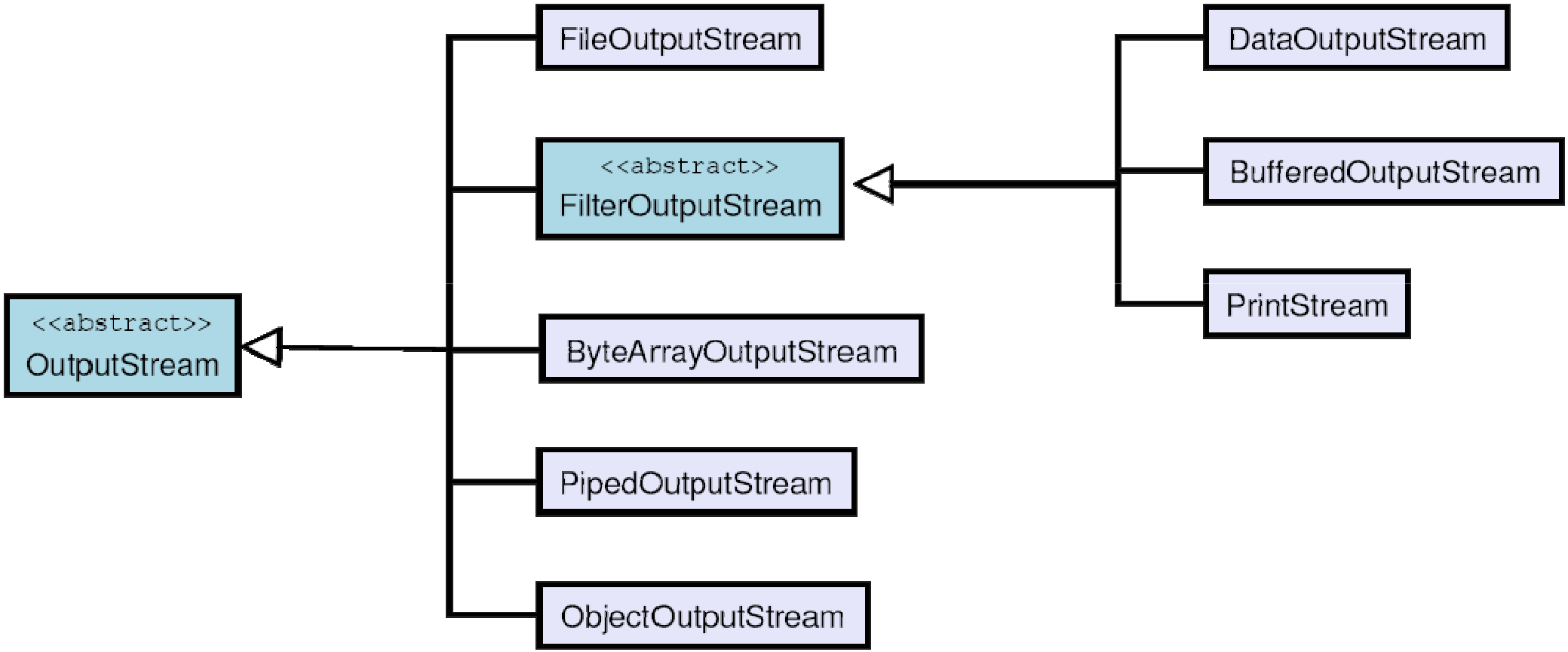
Class	Function
ByteArray-InputStream	Allows a buffer in memory to be used as an InputStream
StringBuffer-InputStream	Converts a String into an InputStream
File-InputStream	For reading information from a file
Piped-InputStream	Produces the data that's being written to the associated PipedOutput-Stream. Implements the "piping" concept.
Sequence-InputStream	Converts two or more InputStream objects into a single InputStream.
Filter-InputStream	Abstract class which is an interface for decorators that provide useful functionality to the other InputStream classes.

La gerarchia `Filter-InputStream`

Class	Function
Data-InputStream	Used in concert with DataOutputStream, so you can read primitives (int, char, long, etc.) from a stream in a portable fashion.
Buffered-InputStream	Use this to prevent a physical read every time you want more data. You're saying "Use a buffer."
LineNumber-InputStream	Keeps track of line numbers in the input stream; you can call getLineNumber( ) and setLineNumber(int).
Pushback-InputStream	Has a one byte push-back buffer so that you can push back the last character read.

Classi per *stream* byte oriented

» La gerarchia **`OutputStream`**



La gerarchia `OutputStream`

Class	Function
ByteArray-OutputStream	Creates a buffer in memory. All the data that you send to the stream is placed in this buffer.
File-OutputStream	For sending information to a file.
Piped-OutputStream	Any information you write to this automatically ends up as input for the associated PipedInput-Stream. Implements the “piping” concept.
Filter-OutputStream	Abstract class which is an interface for decorators that provide useful functionality to the other OutputStream classes.

La gerarchia `Filter-OutputStream`

Class	Function
Data-OutputStream	Used in concert with DataInputStream so you can write primitives (int, char, long, etc.) to a stream in a portable fashion.
PrintStream	For producing formatted output. While DataOutputStream handles the storage of data, PrintStream handles display.
Buffered-OutputStream	Use this to prevent a physical write every time you send a piece of data. You're saying "Use a buffer." You can call flush( ) to flush the buffer.

Classi per *stream* byte oriented

» La gerarchia **`InputStream`** contiene tutte le classi che gestiscono l'input da *stream* byte oriented

» La gerarchia **`OutputStream`** contiene tutte le classi che gestiscono l'output su *stream* byte oriented

Classi per *stream* byte oriented

» Canali tipici di input:

- Un array di byte (**`ByteArrayInputStream`**)
- Un oggetto String (**`StringBufferInputStream`**)
- Un file (**`FileInputStream`**)
- Una pipe che realizza lo scambio tra processi (thread) (**`PipedInputStream`**)
- Una sequenza da altri stream collettati insieme in un singolo stream (**`SequenceInputStream`**)
- Altre sorgenti, come le connessioni ad Internet

» In aggiunta la classe astratta **`FilterInputStream`** fornisce utili modalità di input per dati particolari come i tipi primitivi, meccanismi di bufferizzazione, ecc.

Classi per *stream* byte oriented

» La gerarchia **`OutputStream`** contiene tutte le classi che gestiscono l'output su *stream* byte oriented

Classi per *stream* byte oriented

» Canali tipici di output:

- **`ByteArrayOutputStream`**: crea un buffer in memoria e invia dati al buffer
- **`FileOutputStream`**: per inviare informazioni a un file
- **`PipedOutputStream`**: implementa il concetto di pipe
- **`ObjectOutputStream`**: per inviare oggetti al destinatario

» In aggiunta la classe astratta **`FilterOutputStream`** fornisce utili modalità di output per dati particolari come i tipi primitivi, meccanismi di bufferizzazione, ecc.

La classe `InputStream`

» La classe **`InputStream`** è la classe base della gerarchia. I metodi dichiarati in questa classe sono disponibili nelle varie sottoclassi

```
public class InputStream {
  ...
  int read();
  int read(byte[] b);
  int read(byte[] b, int off, int len);
  void close();
}
```

La classe `InputStream`

- **`public abstract int read() throws IOException`**
  Legge un singolo byte e lo restituisce sotto forma di intero (tra 0 e 255). Se non è disponibile (fine dello stream) viene restituito -1.

- **`int read(byte[] b)`**
  Legge dei byte dallo stream e li memorizza nell'array b. restituisce -1 se si è arrivati alla fine dello stream

- **`public int read(byte[] b, int off, int len) throws IOException`**
  Legge una sequenza di byte e la memorizza in un array di byte. Il numero massimo di byte letti è len. I byte vengono memorizzati a partire da b[off] sino a un massimo di b[off+len-1]. Restituisce -1 se si `e giunti alla fine del flusso e 0 se vengono letti zero byte

- **`public void close() throws IOException`**
  Rilascia tutte le risorse associate con questo *stream*

La classe `OutputStream`

» La classe **`OutputStream`** è la classe base della gerarchia. I metodi dichiarati in questa classe sono disponibili nelle varie sottoclassi

```
public class OutputStream {
  ...
  void write(int b);
  void write(byte[] b);
  void write(byte[] b, int off, int len);
  void close();
}
```

La classe `OutputStream`

- **`public abstract void write(int b) throws IOException`**
  Scrive b sotto forma di byte. Il byte viene passato come argomento di tipo int perchè spesso è il risultato di un'operazione aritmetica su un byte. Tuttavia solo gli 8 bit meno significativi del numero intero vengono scritti

- **`public void write(byte[] buf, int offset, int count) throws IOException`**
  Scrive una parte di array di byte, a partire da buff[offset] fino a un numero pari a count

- **`public void flush() throws IOException`**
  Effettua il flush (svuotamento) del flusso

- **`public void close() throws IOException`**
  Chiude il flusso di output. Andrebbe invocato per rilasciare le risorse (es. i file) associate allo stream.

Uso di `InputStream` e `OutputStream`

- Le classi **`InputStream`** e **`OutputStream`** sono dichiarate come **`abstract`** e, di conseguenza, non sono direttamente istanziabili

- Raramente i metodi **`read`** e **`write`** dichiarati dalle classi **`InputStream`** e **`OutputStream`** sono utilizzati direttamente ma servono soprattutto alle sottoclassi per recuperare i dati dagli *stream* soggiacenti

Sottoclassi: `ByteArrayInputStream`

» La classe **`ByteArrayInputStream`** consente di utilizzare una zona di memoria come sorgente per uno *stream* di input

» L'interfaccia offerta dalla classe **`ByteArrayInputStream`** è la stessa della classe **`InputStream`** a differenza del costruttore che accetta come parametro il buffer da cui recuperare i dati

Sottoclassi: `FileInputStream`

» La classe **`FileInputStream`** consente di utilizzare un file come *stream* di input

» L'interfaccia offerta dalla classe **`FileInputStream`** è la stessa della classe **`InputStream`** a differenza del costruttore che accetta come parametro il nome del file da utilizzare

Sottoclassi: `PipedInputStream`

- La classe **`PipeInputStream`** consente di implementare il meccanismo delle *pipe*. Viene utilizzata insieme alla **`PipedOutputStream`** che consente l'immissione di dati nella *pipe*

- L'interfaccia offerta dalla classe **`PipedInputStream`** è la stessa della classe **`InputStream`** a differenza del costruttore che accetta come parametro il nome del file da utilizzare

Sottoclassi: `SequenceInputStream`

» La classe **`SequenceInputStream`** consente di concatenare uno o più *stream* di input in un singolo *stream*

» L'interfaccia offerta dalla classe **`SequenceInputStream`** è la stessa della classe **`InputStream`** a differenza del costruttore che accetta come parametri gli *stream* da concatenare

Sottoclassi: `*OutputStream`

» Le classi descritte fino ad ora hanno i relativi corrispondenti per gestire gli *stream* di output:

- `ByteArrayOutputStream`
- `FileOutputStream`
- `PipedOutputStream`
- `SequenceOutputStream`

» Il comportamento implementato da queste classi è complementare a quello descritto per le rispettive classi di gestione degli *stream* di input

Flussi di byte da e verso file

» Per scrivere su un file dovremo far cooperare:

- Un oggetto che crea fisicamente un collegamento con il file (vedi la classe **`File`**)
- Un oggetto che gestisce lo stream (il canale di comunicazione verso il file) e che è in grado di inviare byte lungo lo stream (sarà un'istanza di una sottoclasse di **`OutputStream`**)

» In seguito vedremo anche come sia possibile "spezzare" informazioni complesse (ad esempio String, double o int) in singoli byte tali da poter essere inviati sullo stream

La classe File

- La classe File fornisce una rappresentazione astratta ed indipendente dal sistema dei pathname gerarchici (list, permessi, check esistenza, dimensioni, tipo, crea dir, rinomina, elimina)
  - Rappresenta solo il nome di un particolare file o il nome di gruppi di file in una directory
  - Gli oggetti di tipo File consentono di creare un collegamento con il file fisico
- Si ricordi che le interfacce utente e i sistemi operativi utilizzano pathname dipendenti dal sistema per attribuire un nome ai file e alle directory

La classe File: creazione e cancellazione

- `public boolean createNewFile() throws IOException`
  Crea un nuovo file vuoto con il nome specificato se e solo se tale file non esiste

- `public boolean mkdir()`
  Crea una directory con il nome denotato dal nome astratto

- `public boolean delete()`
  Cancella il file o la directory denotati dal nome astratto. Nel caso di directory, allora viene cancellata solo a patto che sia vuota

La classe File: metodi di test

- **`public boolean exists()`**
  Restituisce true se e solo se il file rappresentato dal nome astratto esiste
- **`public boolean setReadOnly()`**
  Rende il file o la directory accessibili solo in lettura. Restituisce true se e solo se l'operazione ha successo
- **`public boolean canWrite()`**
  Restituisce true se e solo se il file denotato dal nome astratto esiste e può essere scritto
- **`public boolean canRead()`**
  Restituisce true se e solo se il file denotato dal nome astratto esiste e pu`o essere letto

La classe File: informazioni

- **`public String getName()`**
  Restituisce il nome del file (o della directory) denotato dal nome astratto. E' solo l'ultimo nome nella nel pathname globale. Se il pathname è vuoto viene restituita la stringa vuota
- **`public String getPath()`**
  Restituisce il pathname completo del file
- **`public long lastModified()`**
  Restituisce un long che rappresenta la data dell'ultima modifica
- **`public long length()`**
  Restituisce la dimensione in bytes del file denotato dal nome astratto

(vedere DirList.java, MakeDirectories.java)

Esercizio

» Costruire la classe CpFile che effettui la copia di un file in un altro già esistente (sovrascrivendolo). Se il file destinatario non esiste solleva l'eccezione:

FileNotFoundException("Il file: "+args[0] ...)

(vedere CpFile.java)

Filtri

Filtri e design pattern

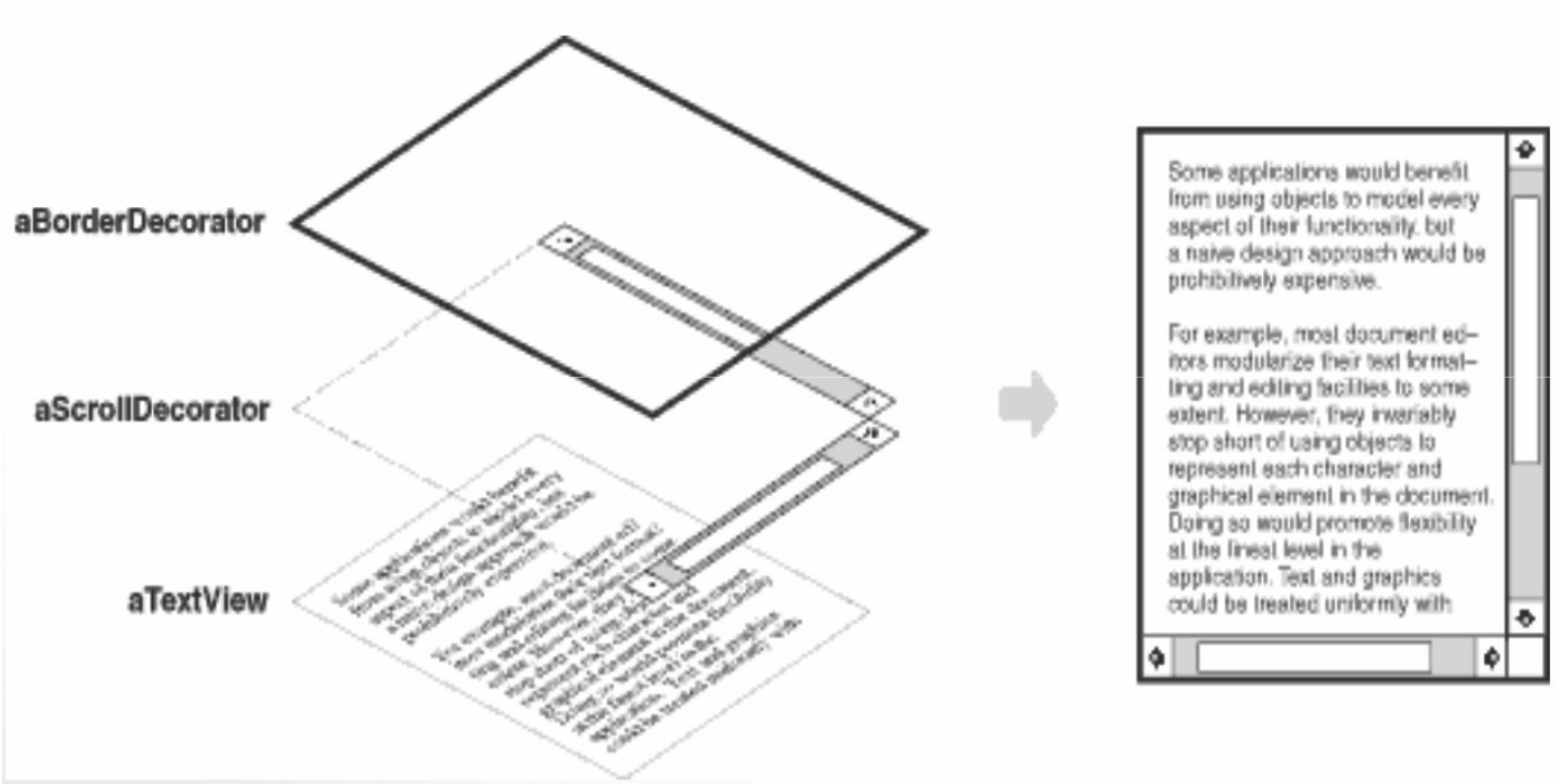
» I filtri consentono di incapsulare progressivamente gli *stream* in oggetti più complessi capaci di manipolare i dati degli *stream* stessi in diversi modi

» Il meccanismo dei filtri segue un famoso *design pattern* chiamato *decorator*. Questo pattern specifica che tutti gli oggetti che vengono combinati hanno tutti la stessa interfaccia. In questo modo si può utilizzarli senza indipendentemente che siano “decorati” o no.

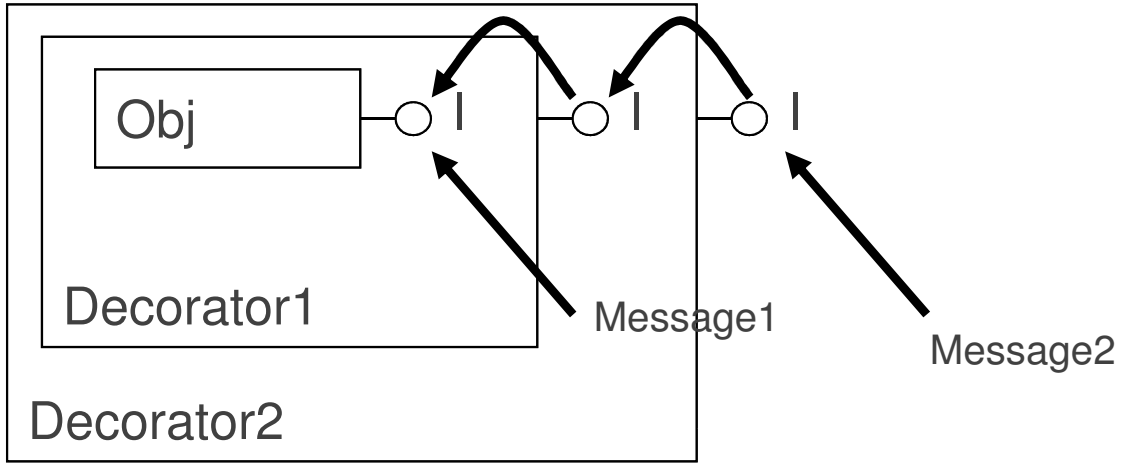
Design pattern *Decorator*

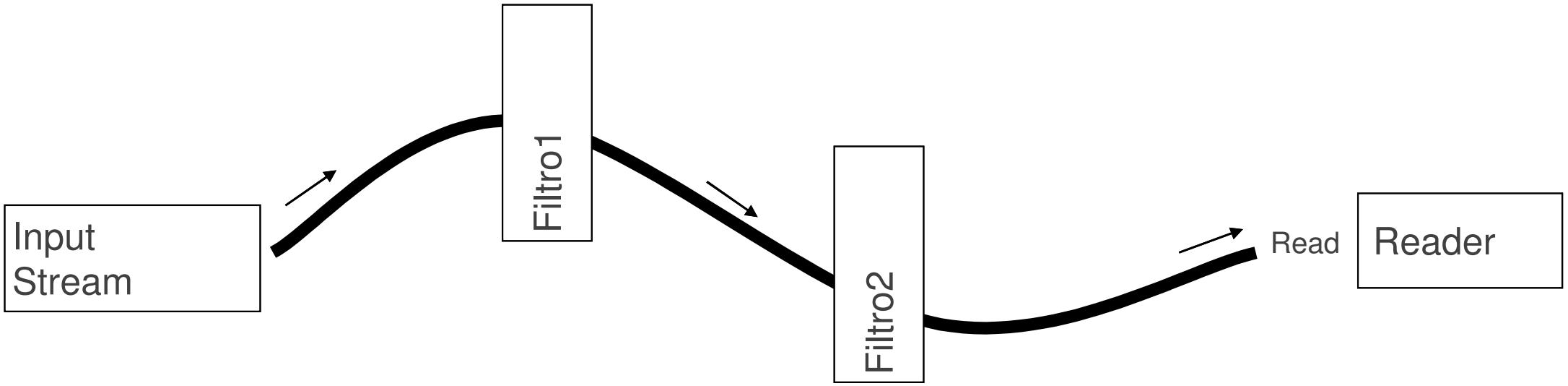
- Consente di aggiungere metodi a classi esistenti durante il run-time, permettendo una maggior flessibilità nell'aggiungere delle funzionalità agli oggetti
- Questo viene realizzato costruendo un nuovo decoratore intorno all'oggetto originale
- Normalmente ciò viene realizzato passando l'oggetto originale come parametro al costruttore del decoratore.

Design pattern *Decorator*



Filtri e design pattern





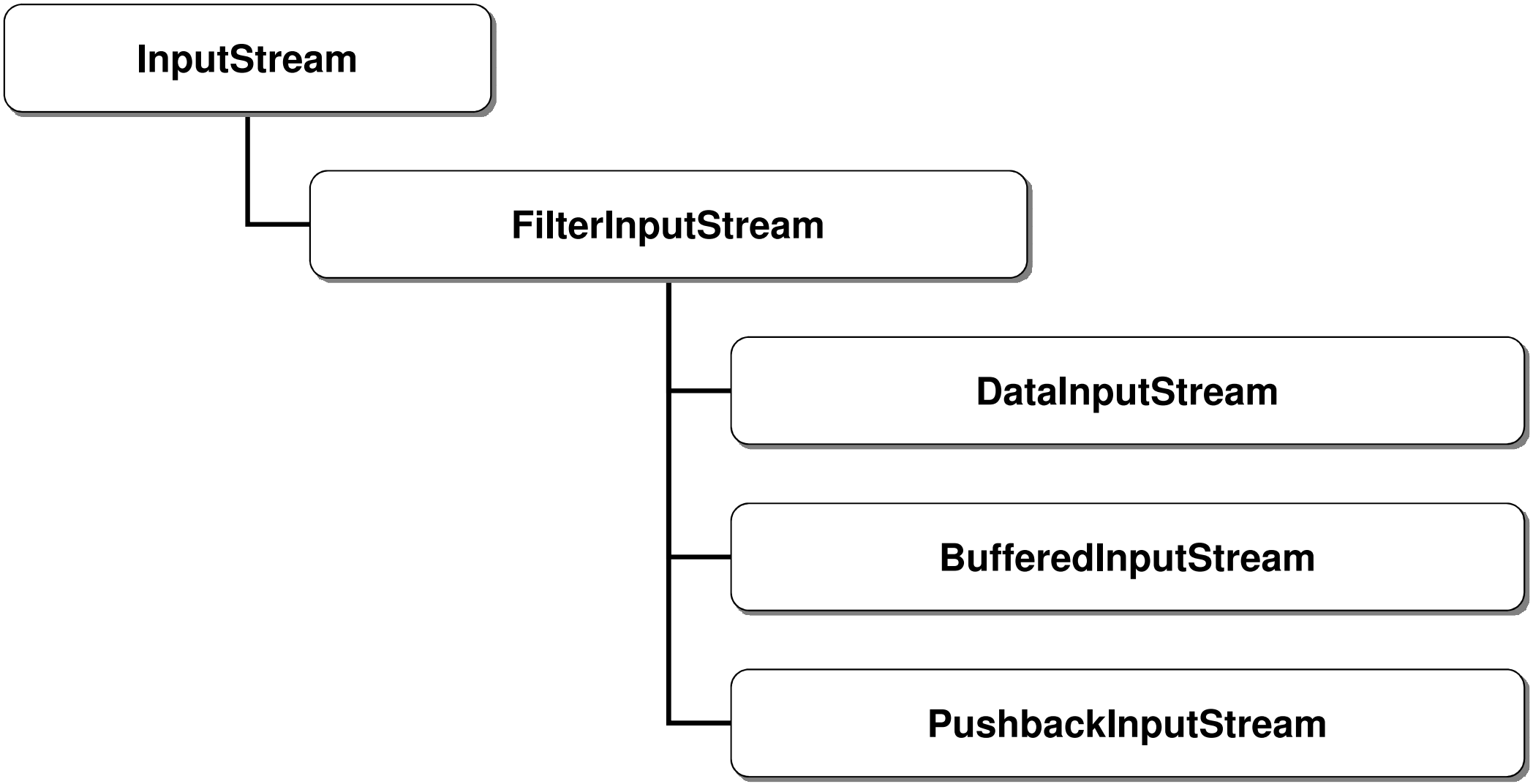
La classe `FilterInputStream`

- La classe **`FilterInputStream`** eredita le proprietà della classe **`InputStream`** e presenta la medesima interfaccia.
- È la classe base della gerarchia di filtri da applicare agli *stream* di input
- È dichiarata come **`abstract`** e non è direttamente istanziabile
- La differenza risiede nel costruttore mediante il quale si specifica lo *stream* di input da filtrare

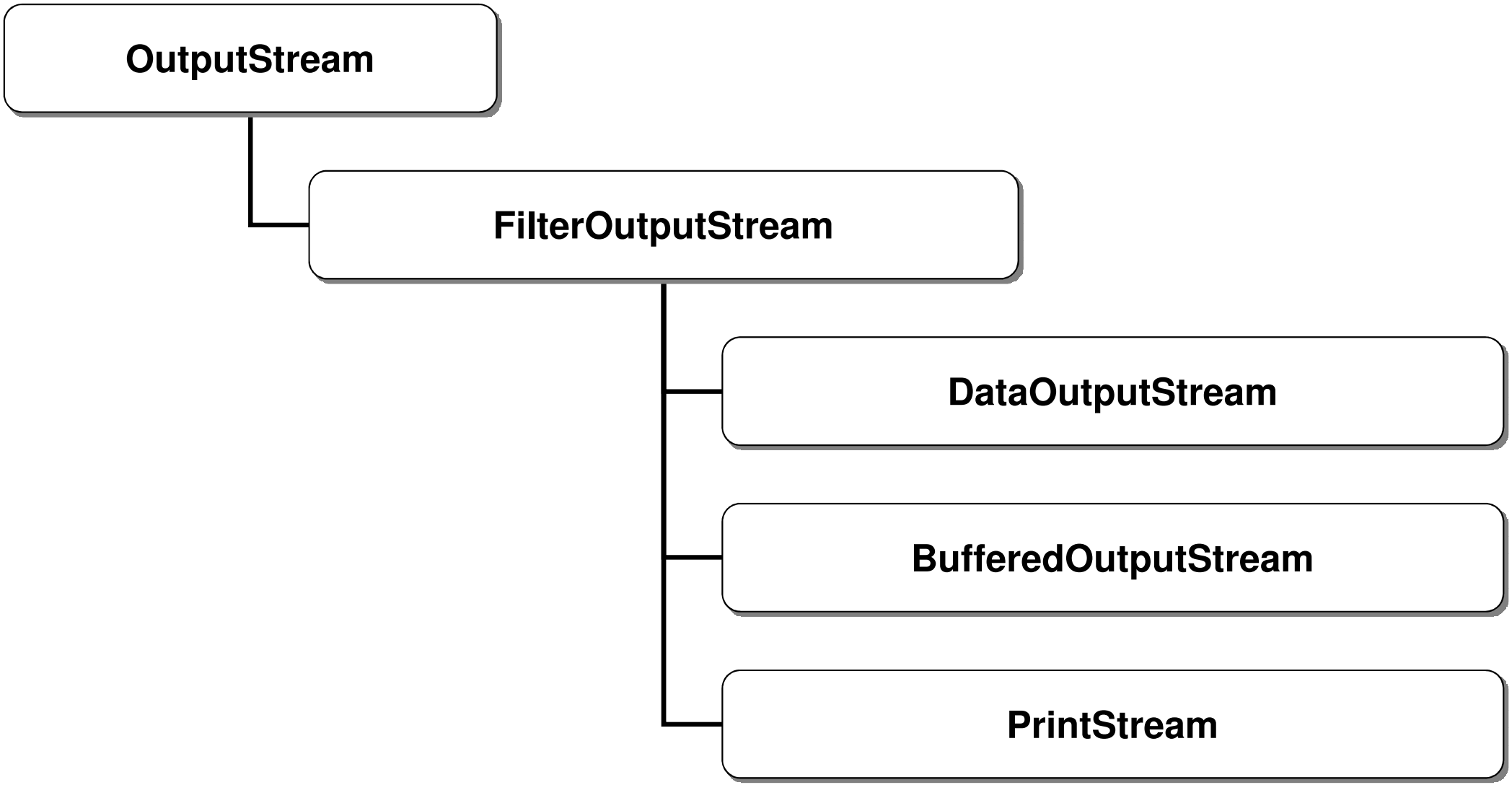
La classe `FilterOutputStream`

- La classe **`FilterOutputStream`** eredita le proprietà della classe **`OutputStream`** e presenta la medesima interfaccia.
- È la classe base della gerarchia di filtri da applicare agli *stream* di output
- È dichiarata come **`abstract`** e non è direttamente istanziabile
- La differenza risiede nel costruttore mediante il quale si specifica lo *stream* di output da filtrare

La gerarchia `FilterInputStream`



La gerarchia `FilterOutputStream`



Sottoclassi: `DataInputStream`

» La classe **`DataInputStream`** fornisce tutti i metodi necessari a leggere da un input *stream* i tipi di dato primitivi in maniera portabile indipendente dal sistema

» Vengono messi a disposizione una serie di metodi **`readType()`** dove **`Type`** è uno dei tipi basici disponibili in *Java* che restituiscono il corrispondente valore letto dallo *stream* di input (es. readByte(), readFloat(), …)

Sottoclassi: `BufferedInputStream`

» La classe **`BufferedInputStream`** fornisce un meccanismo per bufferizzare trasparentemente uno *stream* di input rendendone più efficiente l'accesso

» È consigliabile utilizzare sempre questo filtro con gli *stream* di input

Sottoclassi: `PushbackInputStream`

» La classe **`PushbackInputStream`** fornisce un meccanismo reinserire nello *stream* gli ultimi byte letti in modo che possano essere di nuovo letti successivamente

» L'utilità di questa classe risiede soprattutto nell'implementazione dei compilatori e probabilmente non è spesso utilizzata nella programmazione general purpose

Sottoclassi: *OutputStream

» La classe **`DataOutputStream`** offre le funzionalità complementari a quelle presenti nella classe **`DataInputStream`** per scrivere in maniera portabile i tipi di dato primitivi di *Java* su uno *stream* di output

» La classe **`BufferedOutputStream`** fornisce un meccanismo trasparente per aumentare l'efficienza della gestione degli *stream* di output

- La scrittura fisica di un dato non avviene ad ogni invio del dato
- E' possibile usare il metodo **`flush()`** per svuotare il buffer e forzare la scrittura

Sottoclassi: `PrintStream`

» Per scrivere dati primitivi in un file di testo avremo bisogno di una classe che consenta di convertire tali dati in sequenze di byte

» La classe **`PrintStream`**, che è un'estensione **`FilterOutputStream`**, fornisce tutte le funzionalità necessarie per scrivere su uno *stream* di output tutti i tipi di dato presenti in *Java*

– **`public PrintStream(OutputStream out)`**
Crea un print stream verso l'output stream specificato

– I metodi che mette a dispozione sono i tipici metodi di scrittura:

- **`void print(boolean b), void print(char c),`**
- **`void print(char[] s), void print(double d),`**
- **`void print(float f), void print(int i), void`**
- **`print(long l), void print(Object obj), void`**
- **`print(String s)`**
- La corrispondente versione con **`println`**

Sottoclassi: `PrintStream`

- A differenza dalle funzionalità offerte dalla classe **`DataOutputStream`**, i dati scritti sullo *stream* di output sono in formato leggibile per gli uomini
- In particolare, **`DataOutputStream`** gestisce la *memorizzazione* dei dati, **`PrintStream`** la *visualizzazione*
- Scrivere il float 3.14 mediante un **`DataOutputStream`** non significa scrivere i caratteri '3', '.', '1' e '4' (cosa che accadrebbe utilizzando **`PrintStream`**), bensì significa scrivere sullo *stream* di output tutti i byte della codifica del numero in virgola mobile secondo lo standard utilizzato da *Java*

(vedere DataOutputStreamDemo.java)

Sottoclassi e metodi *deprecated*

- La classe **`LineNumberInputStream`**,pur comparendo nel package **`java.io`**, non deve essere utilizzata poiché fa assunzioni errate sul formato dei dati proveniente dallo *stream* di input.

- Il metodo **`readLine`** della classe **`DataInputStream`** non deve essere utilizzato per lo stesso motivo

- Quando si utilizzano degli *stream* da cui provengono dati testuali bisogna utilizzare le funzionalità messe a disposizione dalle classi per la gestione di *stream char oriented*

Classi per la gestione degli *stream* char oriented

Caratteri UNICODE

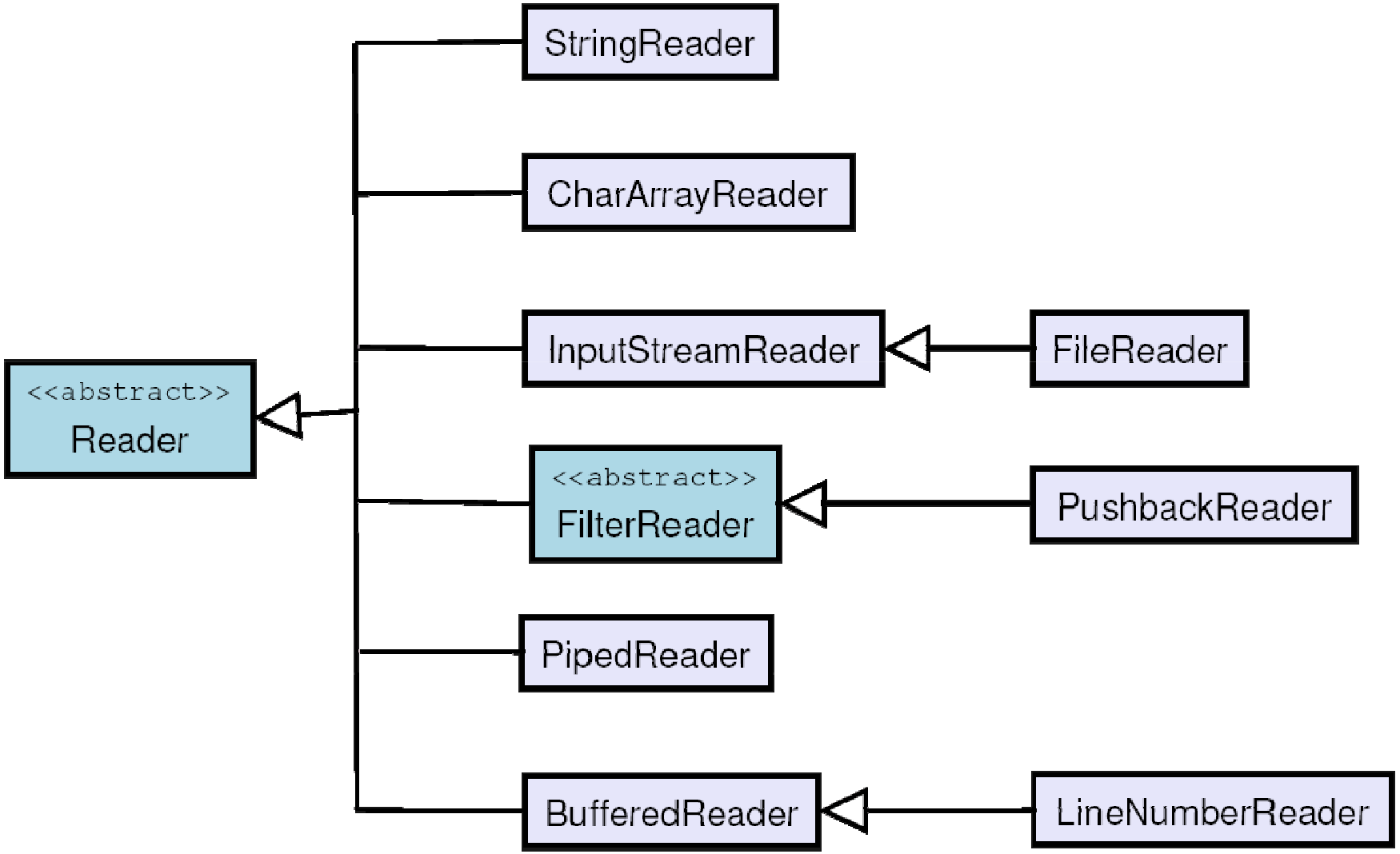
» Unicode è uno standard nato per la internazionalizzazione dei programmi

» Dalla versione 1.3 la piattaforma *Java* gestisce trasparentemente l'internazionalizzazione ed il tipo primitivo **`char`** è compatibile con i caratteri UNICODE

» I caratteri UNICODE hanno una codifica a 16bit secondo uno schema standard (www.unicode.org)

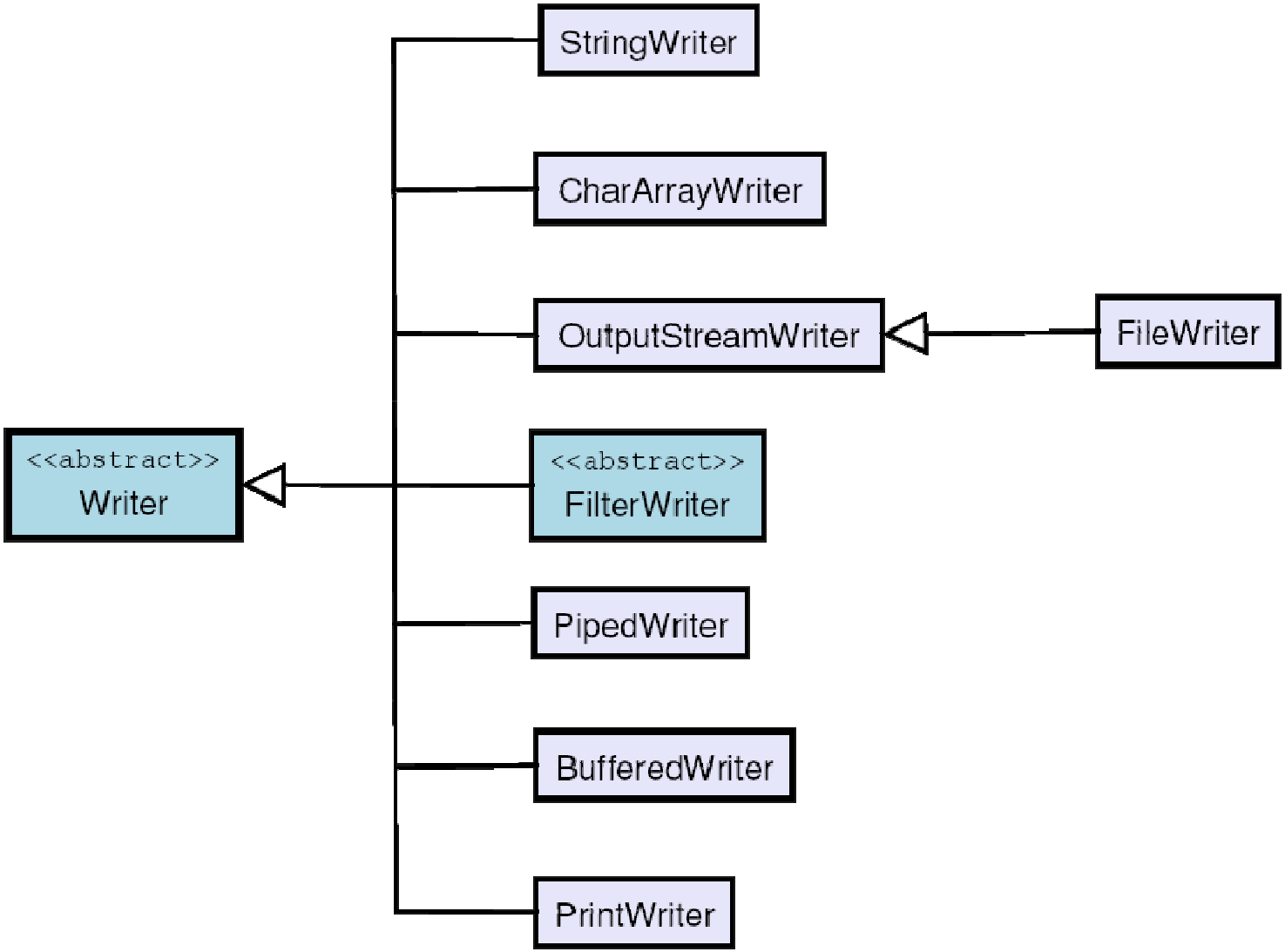
Classi per *stream* char oriented

» La gerarchia **`Reader`**



Classi per *stream* char oriented

» La gerarchia **`Writer`**



Classi per *stream* char oriented

» La gerarchia **`Reader`** contiene tutte le classi che gestiscono l'input da *stream* char oriented

» La gerarchia **`Writer`** contiene tutte le classi che gestiscono l'output su *stream* char oriented

La classe `Reader`

» La classe **`Reader`** è la classe base della gerarchia. I metodi dichiarati in questa classe sono disponibili nelle varie sottoclassi

```
public class Reader {

  ...

  int read();

  int read(char[] b);

  int read(char[] b, int off, int len);

  void close();

}
```

La classe `Reader`

» Fornisce metodi analoghi a quelli della classe InputStream

» **`int read()`**

– Legge un singolo carattere dallo stream e restituisce -1 se si è arrivati alla fine dello stream

» **`int read(char[] b)`**

– Legge dei caratteri dallo stream e li memorizza nell'array b. restituisce -1 se si è arrivati alla fine dello stream

» **`int read(char[] b, int off, int len)`**

– Legge len caratteri dallo stream e li memorizza nell'array b a partire dalla posizione off. Restituisce -1 se si è arrivati alla fine dello stream

» **`void close()`**

– Rilascia tutte le risorse associate con questo stream

La classe `Writer`

» La classe **`Writer`** è la classe base della gerarchia. I metodi dichiarati in questa classe sono disponibili nelle varie sottoclassi

```
public class Writer {

  ...

  void write();

  void write(char[] b);

  void write(char[] b, int off, int len);

  void close();

}
```

La classe `Writer`

- Fornisce un'astrazione per scrivere i caratteri in una certa destinazione
- **`void write()`**
  - Scrive un singolo carattere sullo stream
- **`void write(char[] b)`**
  - Scrive tutti caratteri memorizzati nell'array b sullo stream
- **`void write(char[] b, int off, int len)`**
  - Scrive i len caratteri memorizzati nell'array b a partire dalla posizione off sullo stream
- **`void close()`**
  - Rilascia tutte le risorse associate con questo stream

Uso di `Reader` e `Writer`

- Le classi **`Reader`** e **`Writer`** sono dichiarate come **`abstract`** e, di conseguenza, non sono direttamente istanziabili

- Raramente i metodi **`read`** e **`write`** dichiarati dalle classi **`Reader`** e **`Writer`** sono utilizzati direttamente ma servono soprattutto alle sottoclassi per recuperare i dati dagli *stream* soggiacenti

Anomalie nelle gerarchie `Reader` e `Writer`

» Molte classi delle gerarchie **`Reader`** e **`Writer`** utilizzano il design pattern “decorator” per filtrare i dati provenienti dagli *stream* soggiacenti. Tuttavia nella gerarchia, tali filtri non sono sottoclassi di **`FilterReader`** o **`FilterWriter`** ma discendono direttamente da **`Reader`** o **`Writer`**

» Esempio: **`BufferedReader`**, **`BufferedWriter`**

Sottoclassi: `BufferedReader`

- La classe **`BufferedReader`** è una classe filtro e fornisce un meccanismo per bufferizzare trasparentemente uno *stream* di input rendendone più efficiente l'accesso

- È consigliabile utilizzare sempre questo filtro con gli *stream* di input

- La classe **`BufferedReader`** ha un metodo molto utile che consente di leggere una linea di testo dallo *stream* soggiacente

Sottoclassi: `LineNumberReader`

» La classe **`LineNumberReader`** è una classe filtro e fornisce un meccanismo per tenere traccia del numero di linee lette da uno *stream* di input

» La classe **`LineNumberReader`** discende direttamente da **`BufferedReader`** ed è una delle anomalie della gerarchia **`Reader`**

Sottoclassi: `CharArrayReader`

» La classe **`CharArrayReader`** consente di utilizzare un buffer in memoria come sorgente per uno *stream* di input

» L'interfaccia offerta dalla classe **`CharArrayReader`** è la stessa della classe **`Reader`** a differenza del costruttore che accetta come parametro il buffer da cui recuperare i dati

Sottoclassi: `FilterReader`

» La classe **`FilterReader`** è la classe base per la gerarchia di filtri per *stream* di input orientati ai caratteri

» La classe **`FilterReader`** non è direttamente istanziabile

» Per alcune scelte di design alcune classi che si comportano effettivamente come filtri non discendono direttamente da **`FilterReader`**

Sottoclassi: `PushBackReader`

» La classe **`PushBackReader`** fornisce un meccanismo reinserire nello *stream* gli ultimi caratteri letti in modo che possano essere di nuovo letti successivamente

» L'utilità di questa classe risiede soprattutto nell'implementazione dei compilatori e probabilmente non è spesso utilizzata nella programmazione general purpose

Sottoclassi: `InputStreamReader`

- La classe **`InputStreamReader`** rappresenta il ponte tra gli *stream* di input orientati ai byte e quelli orientati ai caratteri

- Mediante questa classe è possibile utilizzare uno *stream* di byte come se fosse uno di caratteri. Vengono, infatti, letti i byte e convertiti in caratteri secondo la localizzazione attuale

- **`InputStreamReader`** è una sottoclasse di **`Reader`** quindi mette a disposizione i metodi per leggere caratteri

Sottoclassi: `FileReader`

» La classe **`FileReader`** rappresenta il ponte tra gli *stream* di input orientati ai byte e quelli orientati ai caratteri

» La classe **`FileReader`** discende da **`InputStreamReader`** ed espone la medesima interfaccia

» Costruttori:

- `public FileReader(File file)`
- `public FileReader(String nomefile)`

Sottoclassi: `StringReader`

» La classe **`StringReader`** consente di utilizzare una stringa come sorgente per uno *stream* di input di caratteri

Sottoclassi: `*Writer`

» Le classi descritte fino ad ora hanno i relativi corrispondenti per gestire gli *stream* di output:

- `BufferedWriter`
- `CharArrayWriter`
- `FilterWriter`
- `OutputStreamWriter`
- `PipedWriter`
- `StringWriter`

» Il comportamento implementato da queste classi è complementare a quello descritto per le rispettive classi di gestione degli *stream* di input

Sottoclassi: `PrintWriter`

» La classe **`PrintWriter`** fornisce tutte le funzionalità necessarie scrivere su uno *stream* di output tutti i tipi di dato presenti in *Java*.

» I dati scritti sullo *stream* di output sono in formato leggibile per gli uomini.

(vedere IOStreamDemo.java)

Esempio

» Scrivere un programma (classe Copy.java) che usa FileReader e FileWriter per copiare se stesso in un file di backup (CopyBkup.java)

(vedere Copy.java)

Reinderizzare l'I/O standard

» La classe System di Java consente di reindirizzare i flussi di I/O standard di input, output ed essore utilizzando delle chiamate a metodi statici:

- setIn(InputStream)
- setOut(PrintStream)
- setErr(PrintStream)

(vedere Redirecting.java)

Esempio

» Leggere linee da un file di testo

```
//collegamento al file
File file = new File(nomefile);

//stream di caratteri
FileReader fr = new FileReader(file);

//lettore bufferizzato
BufferedReader br = new BufferedReader(fr);
String riga;
while ((riga = br.readLine()) != null )
   System.out.println(riga);
```

Esempio

» Leggere linee dallo standard input

Avevamo osservato che System.in è di tipo InputStream. Allora un possibile modo di leggere linee di testo dallo standard input è il seguente

```
//per convertire stream di byte in strem di car.
InputStreamReader isr =
new InputStreamReader(System.in);

//lettore bufferizzato
BufferedReader br = new BufferedReader(isr);
String riga;
while ((riga = br.readLine()) != null )
   System.out.println(riga);
```

Esempio

» Scrivere la classe Echo.java che legge righe di testo da stdin e, dopo aver digitato return, le ristampa su stdout

(vedere Echo.java)

File ad accesso casuale

File ad accesso casuale

- Tutti gli *stream* visti fino ad ora consentivano un accesso alle risorse in maniera sequenziale
- Le classi che consentivano di operare su *stream* provenienti da file (**`FileInputStream`**, **`FileOutputStream`**, **`FileReader`**, **`FileWriter`**) forniscono dei metodi per leggere dati esclusivamente in maniera sequenziale
- Con l'accesso casuale i dati possono essere letti e riletti da qualsiasi posizione nella struttura soggiacente lo *stream*
- L'accesso casuale viene consentito esclusivamente mediante *stream* associati a file (tipicamente presenti nel filesystem locale)

La classe `File`

» La classe File rappresenta in astratto il concetto di file o directory

» Mediante la classe File è possibile rappresentare un oggetto tipicamente presente sul filesystem

» L'interfaccia esposta da questa classe consente di eseguire le tipiche operazioni possibili sui file o directory:

- Lettura e modifice attributi
- Cancellazione
- Creazione
- Ridenominazione
- Enumerazione dei file presenti nella directory

La classe `RandomAccessFile`

- La classe **`RandomAccessFile`** non fa parte di nessuna delle gerarchie di classi per la gestione dell'input/output ma è una classe a se stante

- Le interfacce implementate dalla classe **`RandomAccessFile`** sono le stesse implementate dalle classi **`DataInputStream`** e **`DataOutputStream`**

- A differenza delle classi per la gestione degli stream la classe **`RandomAccessFile`** esporta il metodo **`seek`** che consente di posizionarsi un qualsiasi punto del file

Classi per la gestione di stream speciali

Object serialization

- La piattaforma *Java* prevede un modo semplice ed efficiente per garantire la persistenza degli oggetti creati a runtime

- Il salvataggio dello stato interno di un oggetto viene detta *serializzazione*

- Il ripristino dello stato precedentemente salvato è detta *deserializzazione*

Object serialization

- La *serializzazione* e la *deserializzazione* vengono gestite mediante degli *stream* particolari detti *object stream*
- Le classi che si occupano di gestire la *object serialization* sono:
  - **`ObjectInputStream`**
  - **`ObjectOutputStream`**
- Poiché non tutte le variabili possono essere *serializzate*, vengono *serializzate* sono quelle variabili che sono dichiarate con i seguenti tipi di dato:
  - Tipi di dato primitivi
  - Tipi di dato che implementano l'interfaccia **`Serializable`**
- La serializzazione degli oggetti salva anche tutti i riferimenti contenuti nell'oggetti, i riferimenti contenuti in ciascun di tali oggetti e così via

La classe `ObjectInputStream`

- La classe **`ObjectInputStream`** consente di effettuare la *deserializzazione* di un oggetto ovvero di recuperare la struttura di un oggetto da uno *stream*

- L'interfaccia offerta da questa classe contiene tutti i metodi per leggere i tipi di dato primitivi ed esporta il metodo **`readObject`** che effettua la *deserializzazione* vera e propria restituendo una istanza dell'oggetto deserializzato

La classe `ObjectOutputStream`

- La classe **`ObjectOutputStream`** consente di effettuare la *serializzazione* di un oggetto ovvero di memorizzare la struttura di un oggetto attraverso uno *stream*

- L'interfaccia offerta da questa classe contiene tutti i metodi per scrivere i tipi di dato primitivi ed esporta il metodo **`writeObject`** che effettua la *serializzazione* vera e propria

Object serialization

- Gli oggetti non sono automaticamente serializzabili
- Per poter scrivere (o leggere) gli oggetti di una classe questa deve dichiarare esplicitamente di ammettere la serializzazione implementando l'interfaccia marker Serializable

Object serialization: `Serializable`

- Si tratta di un'interfaccia vuota
- Non contiene nessun metodo, per questo motivo viene anche detta classe marcatrice (marker)
- Per rendere una classe serializzabile è sufficiente dichiarare che implementa l'interfaccia Serializable
- Non occorre implementare alcun metodo

(vedere Worm.java)

Object serialization

» Per poter recuperare la struttura dell'oggetto una volta letto necessario è che la JVM possa accedere a tutte le classi che sono state coinvolte nel processo di serializzazione

(vedere FreezeAlien.java, Alien.java, ThawAlien.java)

Stream checked

» Gli *stream* checked sono degli *stream* che mantengono un *checksum* sui dati che vi vengono scritti utilizzando una particolare funzione di *checksumming*

» Il meccanismo del calcolo del *checksum* dei dati scritti o letti sullo/dallo *stream* viene implementato mediante i filtri da applicare agli *stream* veri e propri

» Le classi che realizzano il *checksumming* sono:

- **`CheckedInputStream`**
- **`CheckedOutputStream`**

La classe `CheckedInputStream`

- La classe **`CheckedInputStream`** è una classe filtro che implicitamente mantiene traccia del checksum dei dati letti dallo *stream* soggiacente

- costruttore consente di specificare che funzione di *checksum* utilizzare. Nella libreria standard di Java sono presenti due funzioni di *checksum:*
  - CRC32
  - Adler32

- Il metodo **`getChecksum`** consente di recuperare il valore del *checksum* dei dati letti dallo *stream* soggiacente

La classe `CheckedOutputStream`

- La classe **`CheckedOutputStream`** è una classe filtro che implicitamente mantiene traccia del *checksum* dei dati scritti sullo *stream* soggiacente

- Il costruttore consente di specificare che funzione di *checksum* utilizzare. Nella libreria standard di Java sono presenti due funzioni di *checksum:*
  - CRC32
  - Adler32

- Il metodo **`getChecksum`** consente di recuperare il valore del *checksum* dei dati scritto sullo *stream* soggiacente

Stream compressi

» I dati che vengono scritti o letti dagli *stream* possono essere compressi secondo alcuni algoritmi

» I filtri **`GZIPInputStream`** e **`ZipInputStream`** leggono i dati da uno *stream* e ne restituiscono la versione non compressa

» I filtri **`GZIPOutputStream`** e **`ZipOutputStream`** scrivono i dati su uno *stream* comprimendoli trasparentemente

La classe `GZIPInputStream`

- La classe **`GZIPInputStream`** è una classe filtro che implicitamente decomprime i dati che vengono letti dallo *stream* soggiacente

- Si suppone che i dati letti dallo *stream* soggiacente siano compressi utilizzando il medesimo algoritmo usato dal programma GZip

La classe `GZIPOutputStream`

- La classe **`GZIPOutputStream`** è una classe filtro che implicitamente comprime i dati che vengono scritti sullo *stream* soggiacente

- I dati scritto sullo *stream* soggiacente sono compressi utilizzando il medesimo algoritmo utilizzato dal programma Gzip

(Vedere GZIPcompress.java)

Le classi `ZipInputStream` e `ZipOutputStream`

- Le classi **`ZipInputStream`** e **`ZipOutputStream`** funzionano allo stesso modo delle rispettive classi **`GZIPInputStream`** e **`GZIPOutputStream`** ma vengono utilizzati in combinazione con le classi **`ZipFile`** e **`ZipEntry`**
- La classe **`ZipFile`**, similmente alla classe File quando usata per gestire le directory, incapsula la struttura di un file .zip permettendo di recuperarne il contenuto
- La classe **`ZipEntry`** fornisce una rappresentazione di un singolo elemento contenuto all'interno di un file .zip
- Mediante gli stream **`ZipInputStream`** e **`ZipOutputStream`** è possibile leggere e scrivere degli elementi (**`ZipEntry`**) all'interno di file .zip (**`ZipFile`**)

(Vedere ZipCompress.java)

Eccezioni

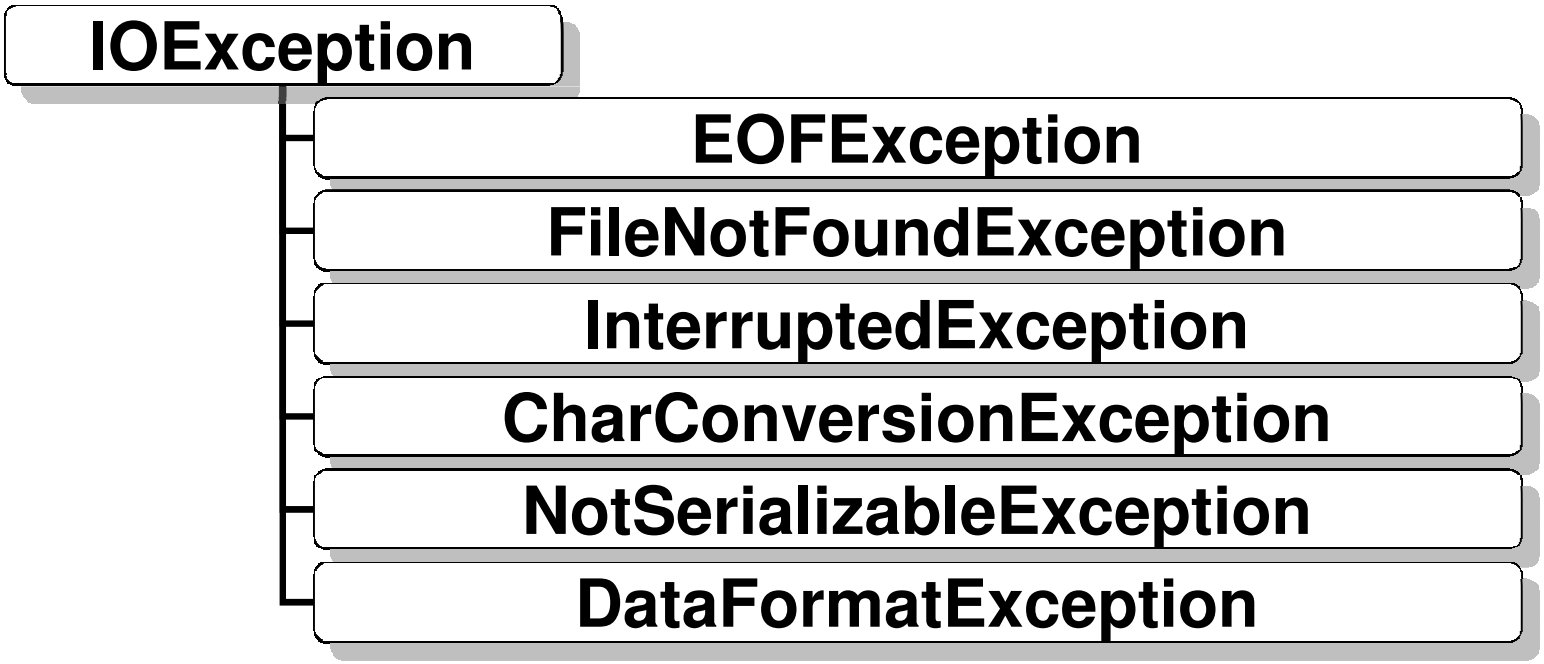
Eccezioni e Input/Output

» Tutti gli errori che si possono verificare nella gestione dell'input/output sono gestiti mediante le eccezioni

» Tipicamente i metodi che vengono chiamati per leggere, scrivere o manipolare i dati sugli *stream* possono sollevare delle eccezioni che vanno opportunamente catturate

Eccezioni e Input/Output

» Tutte le eccezioni derivano dalla classe **`IOException`** che rappresenta un generico errore nella gestione dell'input output



Schema riassuntivo

Tipo di I/O	Classi (Char Oriented/Byte Oriented)	Descrizione

Memoria	`CharArrayReader`
	`CharArrayWriter`
	
	`ByteArrayInputStream`
	`ByteArrayOutputStream`
		Questi stream sono utilizzati per leggere o scrivere su degli array già presenti in memoria

Memoria	`StringReader`
	`StringWriter`
	
	`StringBufferInputStream`
		Stream per leggere o scrivere su delle stringhe utilizzando delle classi di tipo StringBuffer

Pipe	`PipedReader`
	`PipedWriter`
	
	`PipedInputStream`
	`PipedOutputStream`
		Le pipe vengono usate per convogliare l'output di un processo nell'input di un altro

File	`FileReader`
	`FileWriter`
	
	`FileInputStream`
	`FileOutputStream`
		Accesso sequenziale a file presenti nel filesystem

Tipo di I/O	Classi (Char Oriented/Byte Oriented)	Descrizione

Concatenazione	`N/A`
	
	`SequenceInputStream`
		Concatena più input stream come se fossero uno solo

Object	`N/A`
	
	`ObjectInputStream`
	`ObjectOutputStream`
		Consente di scrivere o leggere le rappresentazioni degli oggetti

Conversione dati	`N/A`
	
	`DataInputStream`
	`DataOutputStream`
		Leggono o scrivono i tipi di dato primitivi in un formato indipendente dalla macchina

Stampa	`PrintWriter`
	
	`PrintStream`
		Forniscono dei metodi convenienti per stampare le informazioni

Conteggio linee	`LineNumberReader`
	
	`LineNumberInputStream`
		Tengono traccia del numero di linee di testo lette

Tipo di I/O	Classi (Char Oriented/Byte Oriented)	Descrizione

Buffering	`BufferedReader`
	`BufferedWriter`
	
	`BufferedInputStream`
	`BufferedOutputStream`
		Provvedono a fornire un buffer agli stream per rendere più efficienti le operazioni di input/output

Filtri	`FilterReader`
	`FilterWriter`
	
	`FilterInputStream`
	`FilterOutputStream`
		Consentono di applicare dei filtri anche definiti dall'utente agli stream per processare automaticamente i dati letti o scritti

Conversione tra byte e caratteri	`N/A`
	
	`InputStreamReader`
	`OutputStreamWriter`
		Convertono degli stream orientati ai byte in stream orientati a caratteri