LEZIONE 37
PERSISTENZA
Java I/O, Accesso ai dati, Serializzazione, Deserializzazione

Ingegneria del Software e Progettazione Web

Università degli Studi di Tor Vergata - Roma

Guglielmo De Angelis
guglielmo.deangelis@isti.cnr.it

I/O : dimensioni && aspetti

- dispositivi eterogenei
  - files, console, dispositivi in rete, tastiera, controllori, sensori, etc.
- formati di dato
  - testo, binario, audio, video, compressioni, etc.
- modalità di accesso
  - sequenziale, ad accesso casuale, bufferizzato
- modalità di interazione
  - bit, byte, word, linee, blob, etc.

soluzione Java

- Java gestisce la complessità delle dimensioni di I/O attraverso una vasta gamma di librerie e classi incluse nella piattaforma
- ogni classe incapsula le operazioni necessarie per gestire al meglio :
  - un tipo di dato
  - una tipologia di dispositivi
  - una modalità di accesso
- è possibile trattare tutte le dimensioni della gestione dell'I/O componendo gli oggetti afferenti a classi diverse
- è possibile estendere ed aumentare il comportamento delle librerie di piattaforma specializzandone le classi

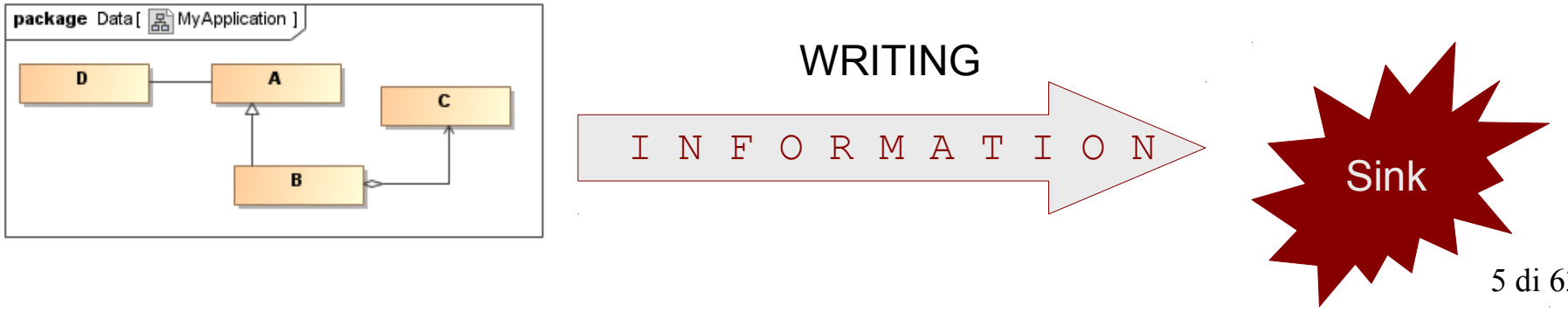
stream – 1

- l'intera gestione dell'I/O in Java è basata sul concetto di stream
- uno stream modella il flusso di informazione con qualsiasi dispositivo capace di produrre o ricevere dati
- uno stream è astratto rispetto ai dettagli con i quali vengono gestiti i dati dagli effettivi dispositivi di I/O
- in generale
  - uno stream è considerato sequenziale
  - esistono dei casi particolari in cui gli stream modellano flussi ad accesso casuale

stream – 2

- per ricevere in ingresso dei dati, un programma apre uno stream su una sorgente di informazioni e ne *legge sequenzialmente* le informazioni

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



stream – 3

- il processo di lettura di informazioni da uno stream segue sempre il seguente schema:

```
1. open(stream)
2. while(more information ?)
3.   read(information)
4. close(stream)
```

- similmente, segue lo schema (generico) per il processo di scrittura da stream

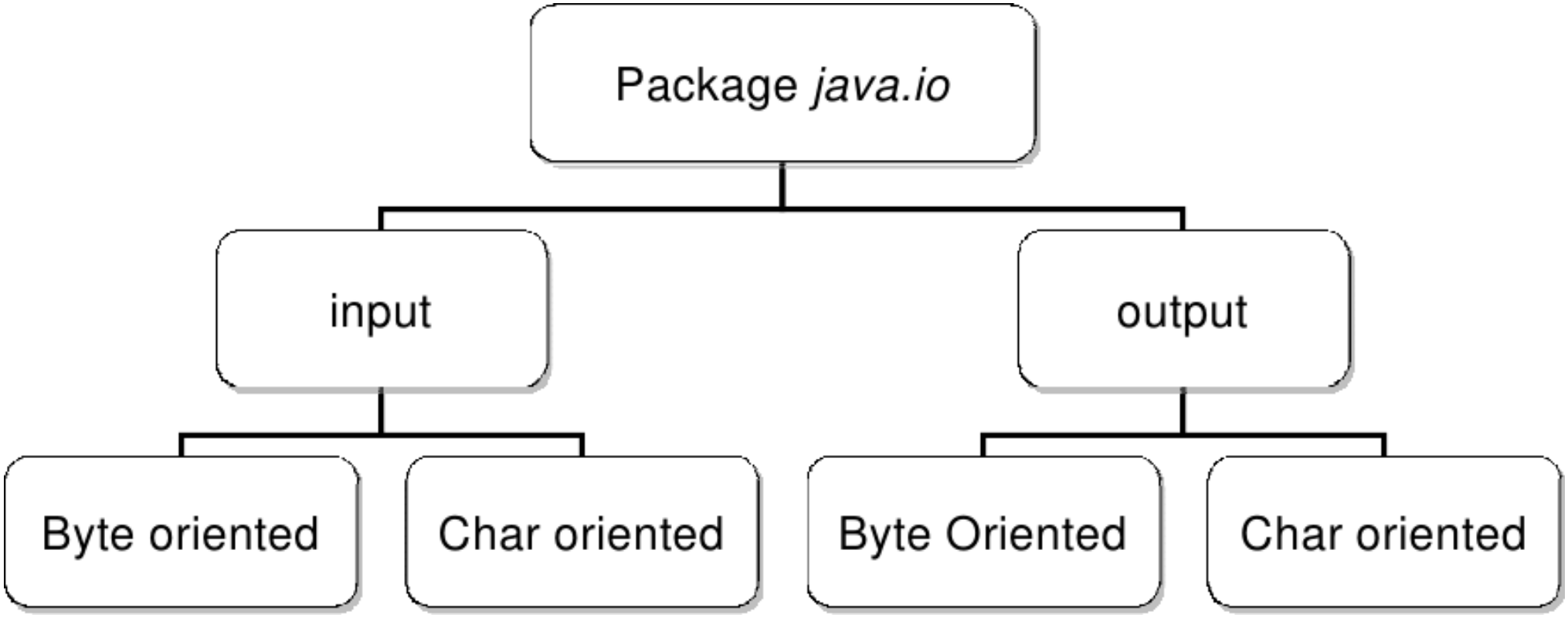
```
1. open(stream)
2. while(more information ?)
3.   write(information)
4. close(stream)
```

java.io – 1

- la libreria standard della piattaforma Java per la gestione degli stream è localizzato nel package `java.io`
- ogni specifico stream è definito sotto forma di classi/interfacce nel package `java.io`
- tipicamente un programma che ha bisogno di utilizzare delle operazioni di I/O avrà nell'intestazione l'import di alcune delle classi presenti in tale package:
  - `import java.io.*;`

`java.io` – 2

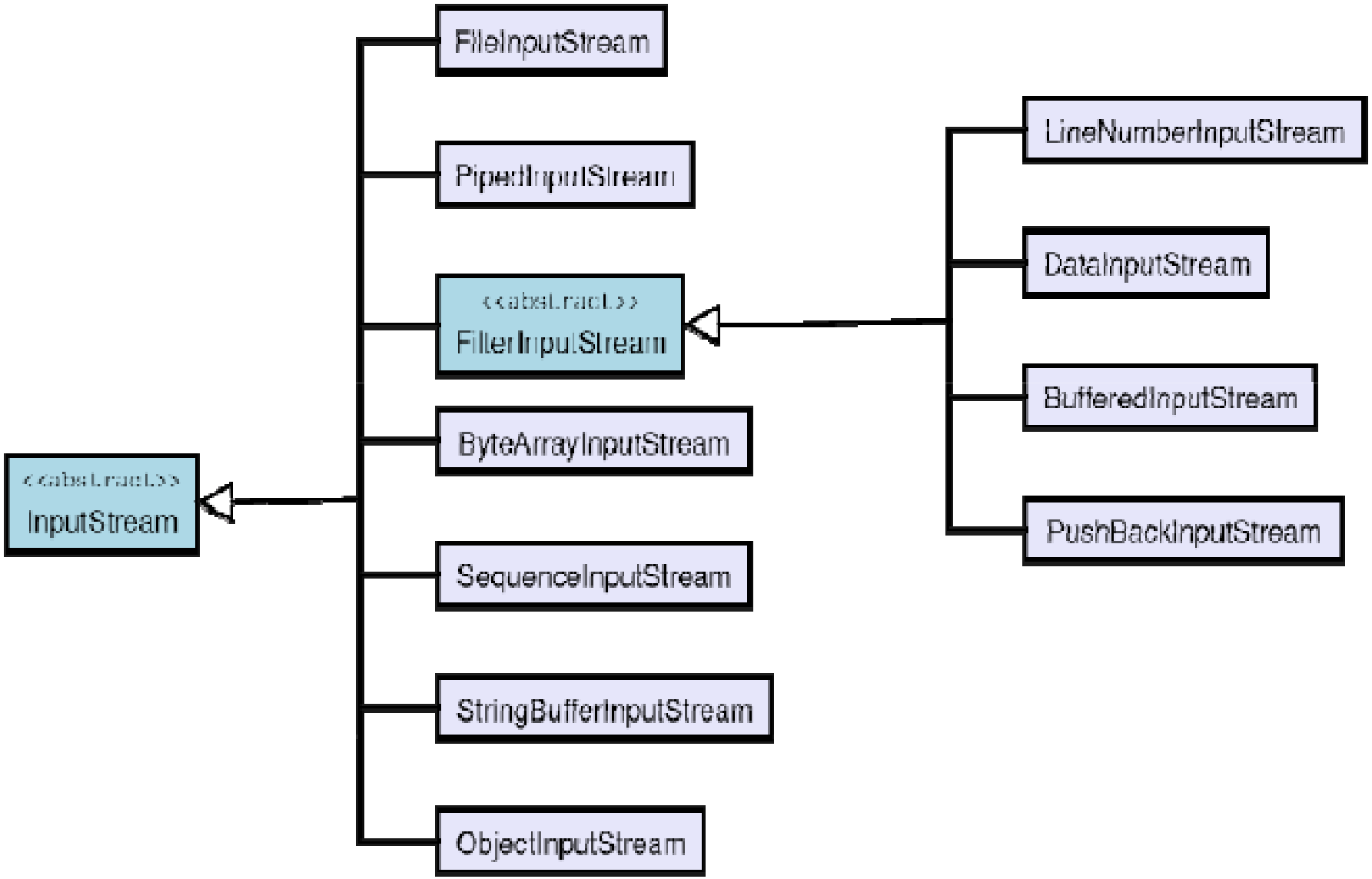
- gli elementi in `java.io` sono logicamente suddivisi in base alla funzionalità che espletano ed a come gestiscono i dati



java.io – 3

- byte oriented stream
  - l'unità atomica di memorizzazione è il byte
  - I/O binario
    - viene usato in generale per i dati (es. i bit di un'immagine digitale o di un segnale sonoro digitalizzato)
  - flussi di byte
    - in ingresso: specializzazioni della classe astratta `InputStream`
    - in uscita: specializzazioni della classe astratta `OutputStream`
- char oriented stream
  - utilizzati per trattare testo
  - I/O testuale
    - usato per scambiare dati rappresentati come sequenza di caratteri (e.g. ASCII)
  - flussi di caratteri
    - lettori: specializzazioni della classe astratta `Reader`
    - scrittori: specializzazioni della classe astratta `Writer`

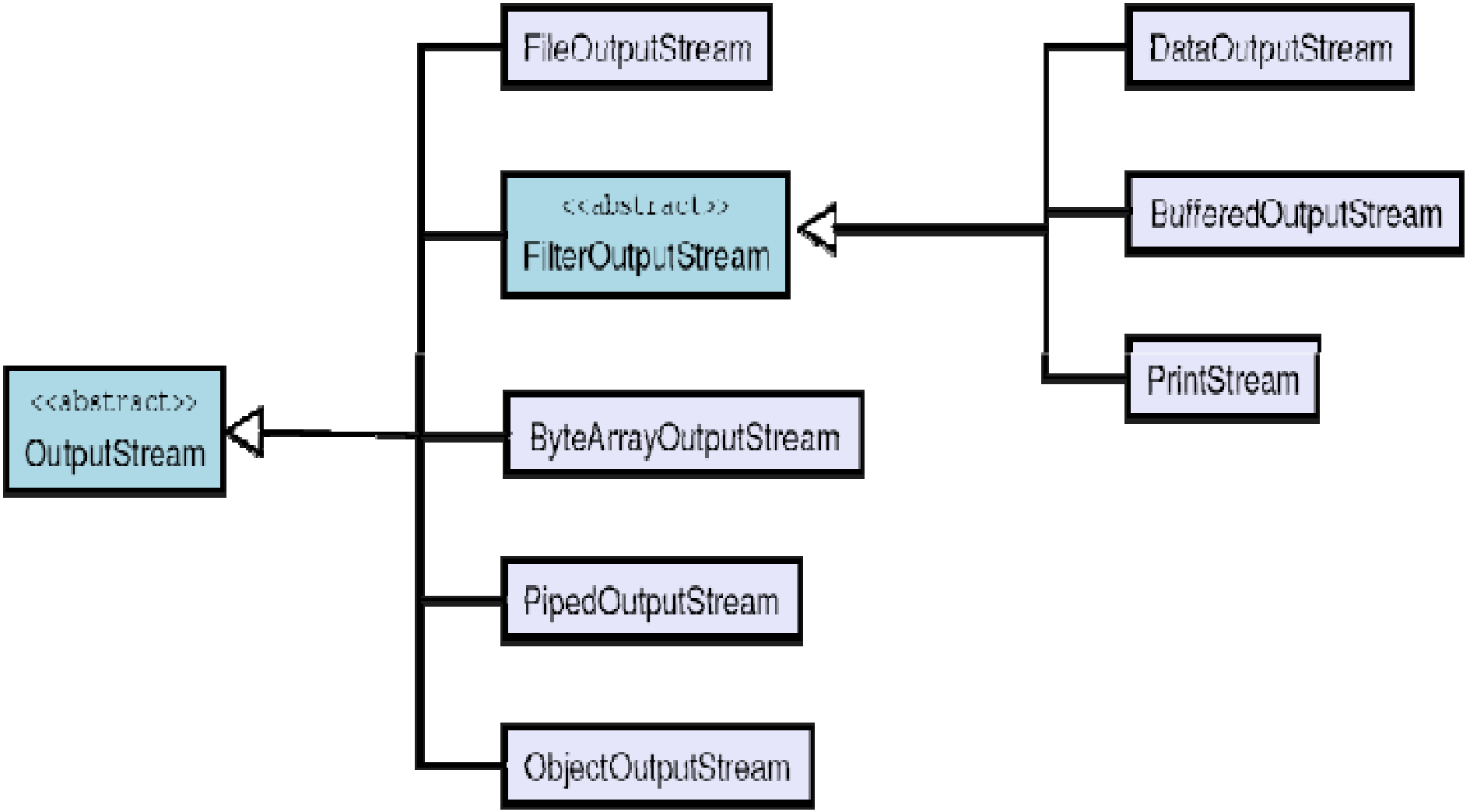
java.io.InputStream



java.io.InputStream

- 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.

java.io.OutputStream



java.io.OutputStream

- 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

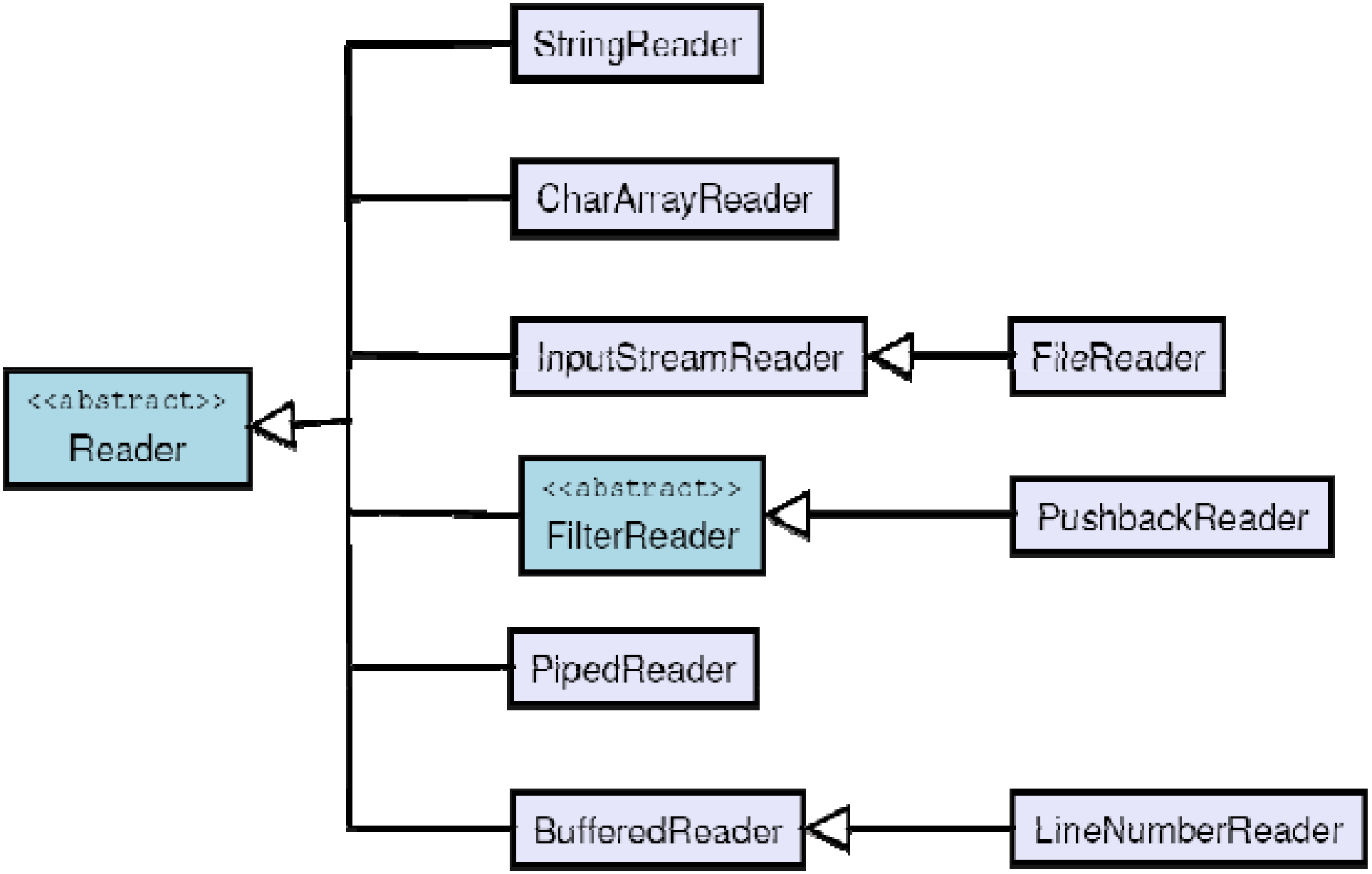
- in aggiunta la classe astratta FilterOutputStream fornisce utili modalità di output per dati particolari come i tipi primitivi, meccanismi di bufferizzazione, ecc.

flussi di byte da/verso file

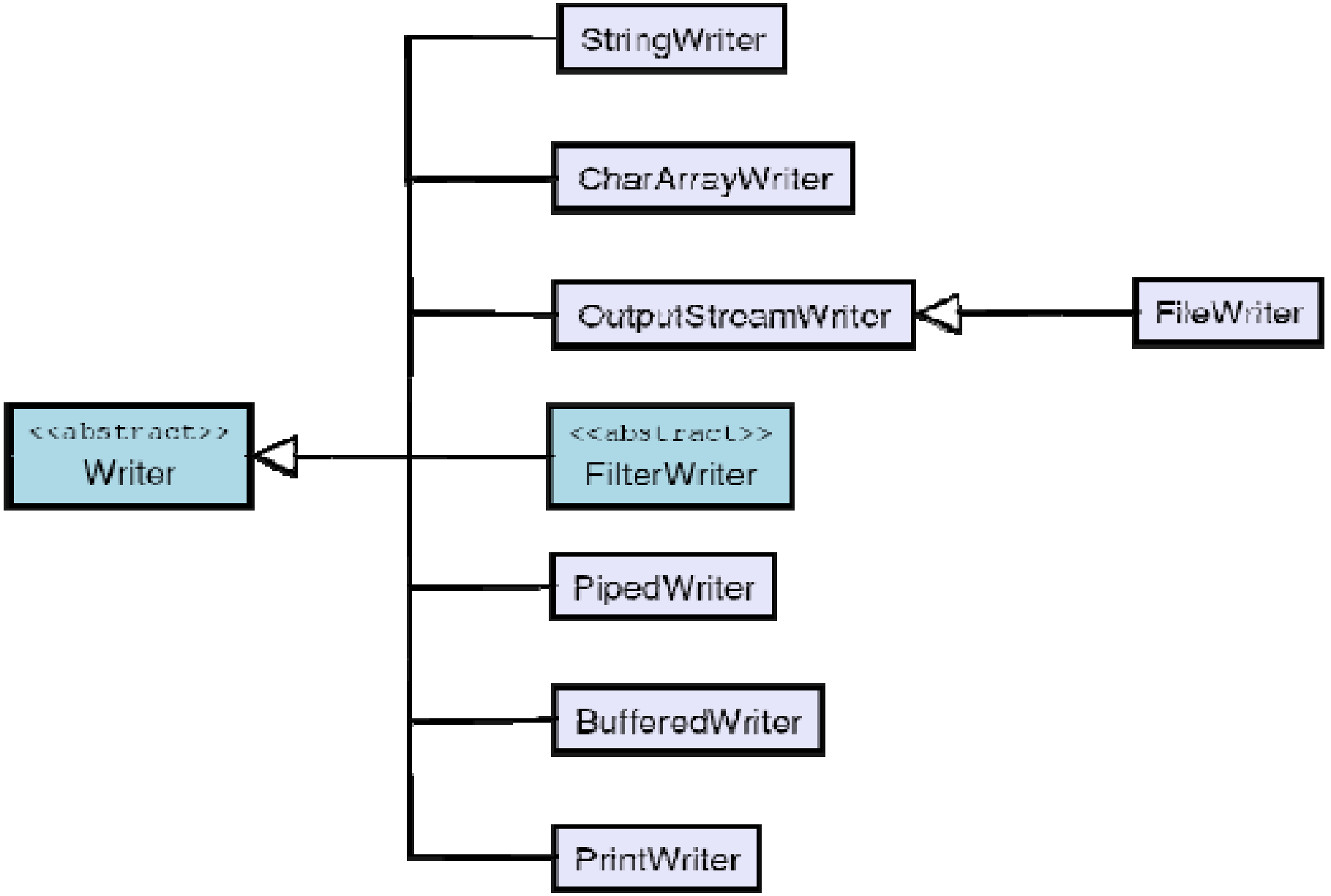
- per scrivere su un file di byte è necessario far cooperare:
  - un oggetto che crea fisicamente un collegamento con il file
  - un oggetto che gestisce lo stream (il canale di comunicazione verso il file) e che è in grado di inviare byte lungo lo stream (sarà un'istanza di una sottoclasse di `OutputStream`)

- la classe **`File`**
  - fornisce una rappresentazione astratta ed indipendente dal sistema dei pathname gerarchici (list, permessi, check esistenza, dimensioni, tipo, crea dir, rinomina, elimina)
    - in generale le interfacce utente e i sistemi operativi utilizzano pathname dipendenti dal sistema per attribuire un nome ai file e alle directory
  - rappresenta solo il nome di un particolare file o il nome di gruppi di file in una directory
  - consente di creare un collegamento con il file fisico

java.io.Reader



java.io.Writer



usi tipici

usi tipici – input bufferizzato da file di caratteri

```
BufferedReader in = new BufferedReader(
   new FileReader("file.txt"));
String s = new String();
String s2 = new String();
while((s = in.readLine())!= null)
   s2 += s + "\n";
in.close();
```

usi tipici – input bufferizzato da file di caratteri

```
BufferedReader in = new BufferedReader(
   new FileReader("file.txt"));
String s = new String();
String s2 = new String();
while((s = in.readLine())!= null)
   s2 += s + "\n";
in.close();
```

apertura file come stream di input a caratteri

usi tipici – input bufferizzato da file di caratteri

```
BufferedReader in = new BufferedReader(
   new FileReader("file.txt"));
String s = new String();
String s2 = new String();
while((s = in.readLine())!= null)
   s2 += s + "\n";
in.close();
```

l'uso lettori bufferizzati migliora le prestazioni di accesso ai dati

usi tipici – input bufferizzato da file di caratteri

```
BufferedReader in = new BufferedReader(
   new FileReader("file.txt"));
String s = new String();
String s2 = new String();
while((s = in.readLine())!= null)
   s2 += s + "\n";
in.close();
```

ogni riga del file è letta dallo stream con il metodo `readLine`
al raggiungimento della fine del file si interrompono le iterazioni

usi tipici – input bufferizzato da file di caratteri

```
BufferedReader in = new BufferedReader(
   new FileReader("file.txt"));
String s = new String();
String s2 = new String();
while((s = in.readLine())!= null)
   s2 += s + "\n";
in.close();
```

lo stream associato al file viene chiuso

usi tipici – input di caratteri da memoria

```
String s2 = "this is foo!!";
StringReader in = new StringReader(s2);

int c;

while((c = in.read())!= -1)

  System.out.print((char)c);
```

usi tipici – input di caratteri da memoria

```
String s2 = "this is foo!!";
StringReader in = new StringReader(s2);

int c;

while((c = in.read()) != -1)

   System.out.print((char)c);
```

viene creato un lettore di flussi di caratteri a partire dalla string `s2`

usi tipici – input di caratteri da memoria

```
String s2 = "this is foo!!";
StringReader in = new StringReader(s2);

int c;

while((c = in.read())!= -1)

   System.out.print((char)c);
```

i caratteri nel flusso vengono letti sequenzialmente con il metodo `read` quando tutti caratteri sono stati letti l'iterazione si interrompe

usi tipici – input di caratteri da memoria

```
String s2 = "this is foo!!";
StringReader in = new StringReader(s2);

int c;

while((c = in.read())!= -1)

   System.out.print((char)c);
```

il carattere letto ha una rappresentazione intera; va convertito

usi tipici – input di byte da memoria

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

usi tipici – input di byte da memoria

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

`getBytes` converte la stringa in una sequenza di byte utilizzando la rappresentazione specifica di Java

usi tipici – input di byte da memoria

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

viene creato un lettore di flussi di byte ...

usi tipici – input di byte da memoria

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

viene creato un lettore di
flussi per array di byte in Java ...

usi tipici – input di byte da memoria

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

… con DataInputStream consente di gestire tutti i tipi di dato primitivi di Java riuscendo ad avere un codice più flessibile

usi tipici – input di byte da memoria

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

viene letto un byte per vota dallo stream

usi tipici – output di caratteri su file

```
String s2 = "this is foo!!";
try {
  BufferedReader in3 = new BufferedReader( new StringReader(s2));
  PrintWriter out1 = new PrinterWriter( new BufferedWriter(
    new FileWriter("file.txt")));
  int lineCount = 0;
  String s;
  while((s =  in3.readLine()) != null ){
   lineCount ++;
   out1.println(lineCount + ": " + s);
  }
  out1.println();
  out1.close();
} catch (EOFException e){
  System.err.print("End of stream");
}
```

usi tipici – output di caratteri su file

```
String s2 = "this is foo!!";
try {
  BufferedReader in3 = new BufferedReader( new StringReader(s2));
  PrintWriter out1 = new PrinterWriter( new BufferedWriter(
    new FileWriter("file.txt")));
  int lineCount = 0;
  String s;
  while((s =  in3.readLine()) != null ){
   lineCount ++;
   out1.println(lineCount + ": " + s);
  }
  out1.println();
  out1.close();
} catch (EOFException e){
  System.err.print("End of stream");
}
```

apertura file come stream di output a caratteri

usi tipici – output di caratteri su file

```
String s2 = "this is foo!!";
try {
  BufferedReader in3 = new BufferedReader( new StringReader(s2));
  PrintWriter out1 = new PrinterWriter( new BufferedWriter(
    new FileWriter("file.txt")));
  int lineCount = 0;
  String s;
  while((s =  in3.readLine()) != null ){
   lineCount ++;
   out1.println(lineCount + ": " + s);
  }
  out1.println();
  out1.close();
} catch (EOFException e){
  System.err.print("End of stream");
}
```

buffering per migliorare
le performance di accesso

usi tipici – output di caratteri su file

```
String s2 = "this is foo!!";
try {
  BufferedReader in3 = new BufferedReader( new StringReader(s2));
  PrintWriter out1 = new PrinterWriter( new BufferedWriter(
    new FileWriter("file.txt")));
  int lineCount = 0;
  String s;
  while((s =  in3.readLine()) != null ){
   lineCount ++;
   out1.println(lineCount + ": " + s);
  }
  out1.println();
  out1.close();
} catch (EOFException e){
  System.err.print("End of stream");
}
```

gestisce lo stream attraverso una classe che supporta la stampa di vari tipi basici formattatandoli come testo

usi tipici – output di caratteri su file

```
String s2 = "this is foo!!";
try {
  BufferedReader in3 = new BufferedReader( new StringReader(s2));
  PrintWriter out1 = new PrinterWriter( new BufferedWriter(
    new FileWriter("file.txt")));
  int lineCount = 0;
  String s;
  while((s =  in3.readLine()) != null ){
   lineCount ++;
   out1.println(lineCount + ": " + s);
  }
  out1.println();
  out1.close();
} catch (EOFException e){
  System.err.print("End of
}
```

lo stream di input è letto una linea alla volta. l'iterazione termia al raggiungimento della fine del flusso

usi tipici – output di caratteri su file

```
String s2 = "this is foo!!";
try {
  BufferedReader in3 = new BufferedReader( new StringReader(s2));
  PrintWriter out1 = new PrinterWriter( new BufferedWriter(
    new FileWriter("file.txt")));
  int lineCount = 0;
  String s;
  while((s =  in3.readLine()) != null ){
   lineCount ++;
   out1.println(lineCount + ": " + s);
  }
  out1.println();
  out1.close();
} catch (EOFException e){
  System.err.print("End of stream");
}
```

gli stream bufferizzati non garantiscono che il loro contenuto venga effettivamente scritto nel file con l'invocazione dei metodi scrittura.
`close()` forza la scrittura dei dati e chiude il file.

usi tipici – output (ed input) di byte su file

```
try {
  DataOutputStream out2 = new DataOutputStream(
    new BufferedOutputStream(
      new FileOutputStream("data.dat")));
  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("data.dat")));
  // Must use DataInputStream for data:
  System.out.println(in5.readDouble());
  // Only readUTF() will properly recover the data
  System.out.println(in5.readUTF());
  // Read the following double and String:
  System.out.println(in5.readDouble());
  System.out.println(in5.readUTF());
} catch( …
...
```

- la rappresentazione dei tipi primitivi di Java su stream è delegata alle classi `DataOutputStream` ed `DataInputStream`
- `DataOutputStream` e `DataInputStream` sono orientate ai byte quindi richiedono gli `InputStream` e `OutputStrem`

usi tipici – output (ed input) di byte su file

```
try {
  DataOutputStream out2 = new DataOutputStream(
    new BufferedOutputStream(
      new FileOutputStream("data.dat")));
  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("data.dat")));
  // Must use DataInputStream for data:
  System.out.println(in5.readDouble());
  // Only readUTF() will properly recover the data
  System.out.println(in5.readUTF());
  // Read the following double and String:
  System.out.println(in5.readDouble());
  System.out.println(in5.readUTF());
} catch( …
...
```

ATTENZIONE !!

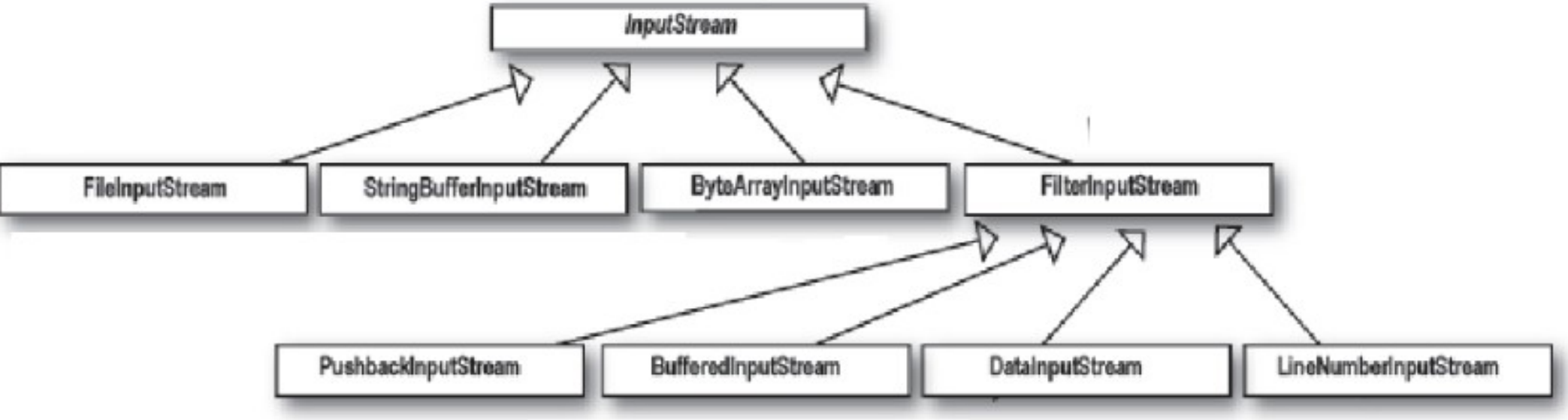
data.dat NON È UN FILE DI TESTO

PROVIAMO AD ESEGUIRE IL CODICE ED AD APRIRLO

- la rappresentazione dei tipi primitivi di Java su stream è delegata alle classi DataOutputStream ed DataInputStream
- DataOutputStream e DataInputStream sono orientate ai byte quindi richiedono gli InputStream e OutputStrem

riflessione – 1

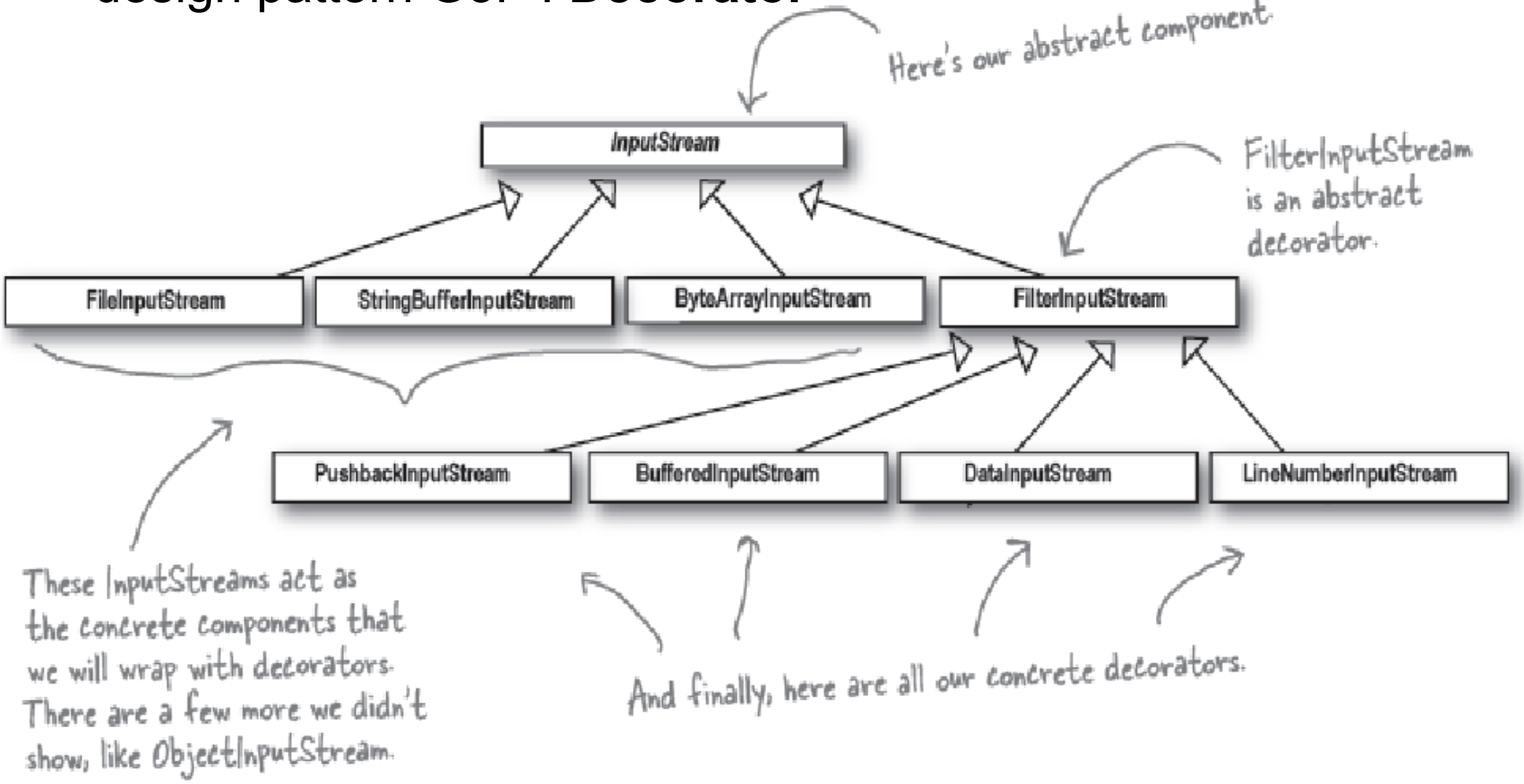
- consideriamo l'organizzazione del package java.io relativamente agli stream; per esempio:
  - riferire la gerarchia introdotta in slide 10 :



  - considerare che dagli esempi abbiamo visto che **stream differenti possono essere composti tra loro**
  - abbiamo già incontrato in altri contesti l'uso di una soluzione simile?

riflessione – 2

- l'organizzazione del package java.io riflette l'applicazione del design pattern GoF : **Decorator**



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 forniscono dei metodi per leggere dati esclusivamente in maniera sequenziale
  - `FileInputStream`, `FileOutputStream`, `FileReader`, `FileWriter`
- 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

java.io.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
- il modo di interazione con la classe `RandomAccessFile` *ricorda* quello definito per `DataInputStream` e `DataOutputStream`
  - le interfacce implementate dalla classe `RandomAccessFile` sono le stesse implementate dalle classi `DataInputStream` e `DataOutputStream`
- differenza delle classi per la gestione degli stream la classe `RandomAccessFile` esporta il metodo **`seek`** che consente di posizionarsi un qualsiasi punto del file

usi tipici – leggere e scrivere file ad accesso casuale

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

usi tipici – leggere e scrivere file ad accesso casuale

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

l'uso di RandomAccessFile è simile all'uso di una combinazione di un `DataOutputStream` e di `DataInputStream`

usi tipici – leggere e scrivere file ad accesso casuale

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

è possibile accedere lo stream in un qualsiasi suo punto senza doverlo scorrere sequenzialmente

persistenza di oggetti – 1

- il ciclo di vita “canonico” di un oggetto
  - inizia
    - l'allocazione di opportuno spazio di memoria (i.e. `new` in Java)
    - l'invocazione di uno dei costruttori definiti dalla classe di afferenza
  - termina
    - la deallocazione esplicita dell'oggetto (i.e. in C++)
    - la deallocazione implicita da parte di un garbage collector (i.e. Java)
    - la terminazione della applicazione che ha creato l'oggetto

persistenza di oggetti – 2

- il ciclo di vita “canonico” di un oggetto ha generalmente senso
- esistono molteplici scenari nei quali è comodo/utile/desiderabile che
  - lo stato di un oggetto continui ad esistere anche dopo la terminazione di una esecuzione dell'applicazione che lo ha creato
  - una applicazione possa configurare il suo contesto di esecuzione attraverso il ripristino dello stato di un insieme di oggetti
  - migrare l'esecuzione di un oggetto tra più nodi di una applicazioni distribuita

persistenza di oggetti – 3

- in generale l'uso di file o DBMS in parte sopperisce a queste necessità, tuttavia:
  - **file**: gestione esplicita delle convenzioni di memorizzazione e del formato dei dati utilizzato
  - **DBMS**: gestione e manutenzione generalmente non adatte a sistemi di dimensioni limitati o non specificatamente “enterprise”. Inoltre c'è da gestire il “mismatch” tra la rappresentazione Object Oriented del dominio e la rappresentazione specifica da adottare nel DB

serialization && deserialization

- Java specifica 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
- entrambe le operazioni vengono gestite riferendo il concetto di stream
- le classi che supportano la gestione di object (de)serialization sono:
  - `ObjectInputStream`
  - `ObjectOutputStream`

come, cosa ?!?!

- la serializzazione degli oggetti prevede che venga salvata l'istanza considerata, ma anche anche tutti i riferimenti in essa contenuti e così via
- Java tratta come serializzabili tutte le variabili afferenti ai tipi di dato primitivi della piattaforma
- per i tipi di dato complessi (i.e. le classi) i meccanismi di serializzazione sono abilitati **se e solo se** una classe realizza l'interfaccia `Serializable`
- `Serializable` è una interfaccia *particolare*:
  - non definisce nessuna operazione (i.e. tagging interface)
  - non richiede la definizione di nessun metodo dalla classe che la realizza

serializzazione && generalizzazione

- dalle javaDoc di `java.io.Serializable` in Java 8
  - "... All subtypes of a serializable class are themselves serializable. ..."
  - "... To allow subtypes of non-serializable classes to be serialized, the subtype may assume responsibility for saving and restoring the state of the supertype's public, protected, and (if accessible) package fields. **The subtype may assume this responsibility only if the class it extends has an accessible no-arg constructor to initialize the class's state**. … "
  - " … During deserialization, the fields of non-serializable classes will be initialized using the public or protected no-arg constructor of the class. A no-arg constructor must be accessible to the subclass that is serializable. The fields of serializable subclasses will be restored from the stream. … "
  - " … Classes that require special handling during the serialization and deserialization process must implement special methods with these exact signatures:
    - `private void writeObject(java.io.ObjectOutputStream out) throws IOException`
    - `private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException;`
    - `private void readObjectNoData() throws ObjectStreamException;`

    ..."

serializzazione && variabili di classe

- serializzatore / deserializzatore in Java lavorano su oggetti allocati nell'heap di memoria
- la JVM non mantiene le variabili con abito di classe in heap
  - sono istanziate sempre e soltanto al momento del caricamento (dinamico) della classe
  - inizializzate sempre con il valore (statico) definito nella classe
- le variabili statiche di una classe Java non sono oggetto di caricamento e salvataggio da parte di serializzatore / deserializzatore
- per serializzare/deserializzare il valore corrente, è necessario prevedere dei meccanismi di espliciti di salvataggio e ripristino in memoria secondo necessità

breve 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

breve schema riassuntivo

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

breve schema riassuntivo

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

esercizi proposti in classe

- **Esercizio 1** : Costruire la classe CpFile che effettui la copia di un file in un altro già esistente (sovrascrivendolo). Se il file destinatario non esiste non fa nulla e stampa a video un messaggio di errore.
- it.uniroma2.dicii.ispw.ioExamples.CpFile.java

esercizi proposti in classe

- **Esercizio 2** : Scrivere un programma (classe Copy.java) che usa FileReader e FileWriter per copiare se stesso in un file di backup (CopyClone.java)

- it.uniroma2.dicii.ispw.ioExamples.CopyFile.java

esercizi proposti in classe

- **Esercizio 3** : Scrivere la classe Echo.java che legge righe di testo da stdin e, dopo aver digitato return, le ristampa su stdout

- it.uniroma2.dicii.ispw.ioExamples.Mirror.java

esercizi proposti : buffered input file (da Thinking in Java)

- **Exercise 7**: (2) Open a text file so that you can read the file one line at a time. Read each line as a String and place that String object into a LinkedList. Print all of the lines in the LinkedList in reverse order.

- **Exercise 8**: (1) Modify Exercise 7 so that the name of the file you read is provided as a command-line argument.

- **Exercise 9**: (1) Modify Exercise 8 to force all the lines in the LinkedList to uppercase and send the results to System.out.

- **Exercise 10**: (2) Modify Exercise 8 to take additional command-line arguments of words to find in the file. Print all lines in which any of the words match.

- **Exercise 11**: (2) In the innerclasses/GreenhouseController.java example, GreenhouseController contains a hard-coded set of events. Change the program so that it reads the events and their relative times from a text file.

esercizi proposti : basic file output

(da Thinking in Java)

- **Exercise 12**: (3) Modify Exercise 8 to also open a text file so you can write text into it. Write the lines in the LinkedList, along with line numbers (do not attempt to use the "LineNumber" classes), out to the file.

- **Exercise 13**: (3) Modify BasicFileOutput.java so that it uses LineNumberReader to keep track of the line count. Note that it's much easier to just keep track programmatically.

- **Exercise 14**: (2) Starting with BasicFileOutput.java, write a program that compares the performance of writing to a file when using buffered and unbuffered I/O.

- **Exercise 15**: (4) Look up DataOutputStream and DataInputStream in the JDK documentation. Starting with StoringAndRecoveringData.java, create a program that stores and then retrieves all the different possible types provided by the DataOutputStream and DataInputStream classes. Verify that the values are stored and retrieved accurately.

esercizi proposti : random access file (da Thinking in Java)

- **Exercise 16**: (2) Look up RandomAccessFile in the JDK documentation. Starting with UsingRandomAccessFile.java, create a program that stores and then retrieves all the different possible types provided by the RandomAccessFile class. Verify that the values are stored and retrieved accurately.