

# Lettura da e scrittura su file

- Java fornisce operazioni di input/output tramite le classi del package `java.io`.
  - ▣ La struttura è indipendente dalla piattaforma.
  - ▣ Le operazioni si basano sul concetto di **flusso**.
- Un flusso (stream) è una sequenza ordinata di dati che ha una sorgente e una destinazione.
  - ▣ L'ordine della sequenza è importante: possiamo pensare a un nastro che viene inciso o riprodotto in un ordine prefissato, un dato dopo l'altro.

# Lettura da e scrittura su file

- Per prelevare dati da una sorgente bisogna collegarla a uno stream, dove leggere le informazioni in modo ordinato.  
Analogamente, per scrivere dati su una destinazione, bisogna collegarla a uno stream in cui inserire i dati in modo sequenziale.
  - ▣ Già visto per lettura da input / scrittura a video.
- All'inizio di queste operazioni occorre **aprire** lo stream e alla fine occorre **chiuderlo**.

# Lettura da e scrittura su file

- L'uso degli stream maschera la specifica natura fisica di una sorgente o una destinazione.
  - ▣ Si possono trattare oggetti differenti allo stesso modo, ma anche oggetti simili in modi differenti.
- In particolare, esistono due modi principali:
  - ▣ **modalità testo** (per esempio, per file di testo o per l'output a console video): immediato per l'utente
  - ▣ **modalità binaria** (per dati elaborati): si leggono e scrivono byte, risultati non immediati per l'utente

# Lettura da e scrittura su file

- In **modalità testo** i dati manipolati sono in forme simili al tipo `char` di Java.
  - ▣ Le classi coinvolte terminano in `-Reader`, `-Writer`
- In **modalità binaria** i dati manipolati sono byte.
  - ▣ Le tipiche classi coinvolte si chiamano **gestori di flussi** e hanno il suffisso `-(Input/Output)Stream`
- Per entrambi i casi abbiamo già visto le applicazioni al caso di lettura e scrittura a video (con conversioni da byte a stringhe)

# Il file system

- Il tipo **File** può contenere un riferimento a un file fisico del sistema. Il riferimento è creato da un costruttore con un parametro stringa.
  - ▣ in realtà può anche essere una directory
- **File x = new File("temp.tmp");** associa il file `temp.tmp` all'oggetto **File** di nome **x**.
- La classe dispone di utili metodi boolean:
  - ▣ **exists()**, **canRead()**, **canWrite()**, **isFile()**, **isDirectory()**

# Il file system

- La classe **File** ha poi altri metodi utili, come:
  - ▣ **length()**: ritorna il valore in byte
  - ▣ **list()**: se invocato su una cartella, ritorna i nomi dei file in un array di stringhe
  - ▣ **setReadable()**, **setWritable()**, **setReadOnly()**: imposta diritti di lettura/scrittura
  - ▣ **createNewFile()**, **mkdir()**: crea file/cartella
  - ▣ **delete()**: cancella file/cartella
  - ▣ **getParent()**: ritorna la cartella madre (./..)
- E alcuni campi statici, tra cui **File.separator**: il separatore di path (in Linux è `'/'`).

# Il file system

- Stampa i nomi dei file di **x** e delle sottocartelle:

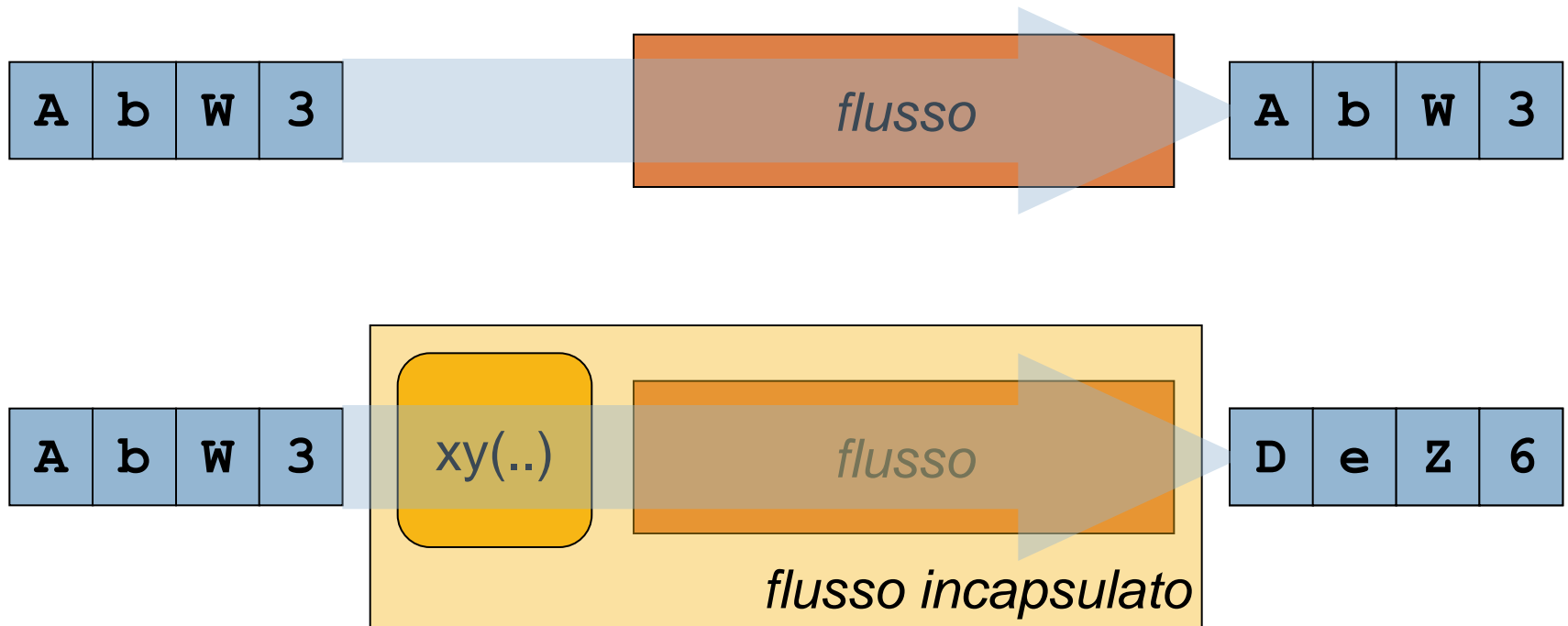
```
public static void stampaRicors (File x)
{ if ( x == null || !x.exists() )
  { return; }
  else if ( x.isDirectory() )
  { String[] elenco = x.list();
    String percorso = x.getAbsolutePath();
    for (String nome: elenco)
    { nome = percorso + File.separator + nome;
      stampaRicors(new File(nome));
    }
  } else if ( x.isFile() )
  { System.out.println(x.getName()); }
}
```

# Incapsulamento dei flussi

- In molti casi, i dati che leggiamo o scriviamo vanno opportunamente processati.
  - ▣ Cifratura, compressione, conversione, ...
- Questa operazione dovrebbe essere trasparente, per garantire portabilità: si parla in tal caso di **incapsulamento dei flussi**.
- Il package `java.io` mette a disposizione le classi `FilterReader`, `FilterWriter`, `FilterInputStream`, `FilterOutputStream` per incapsulare flussi.



# Incapsulamento dei flussi



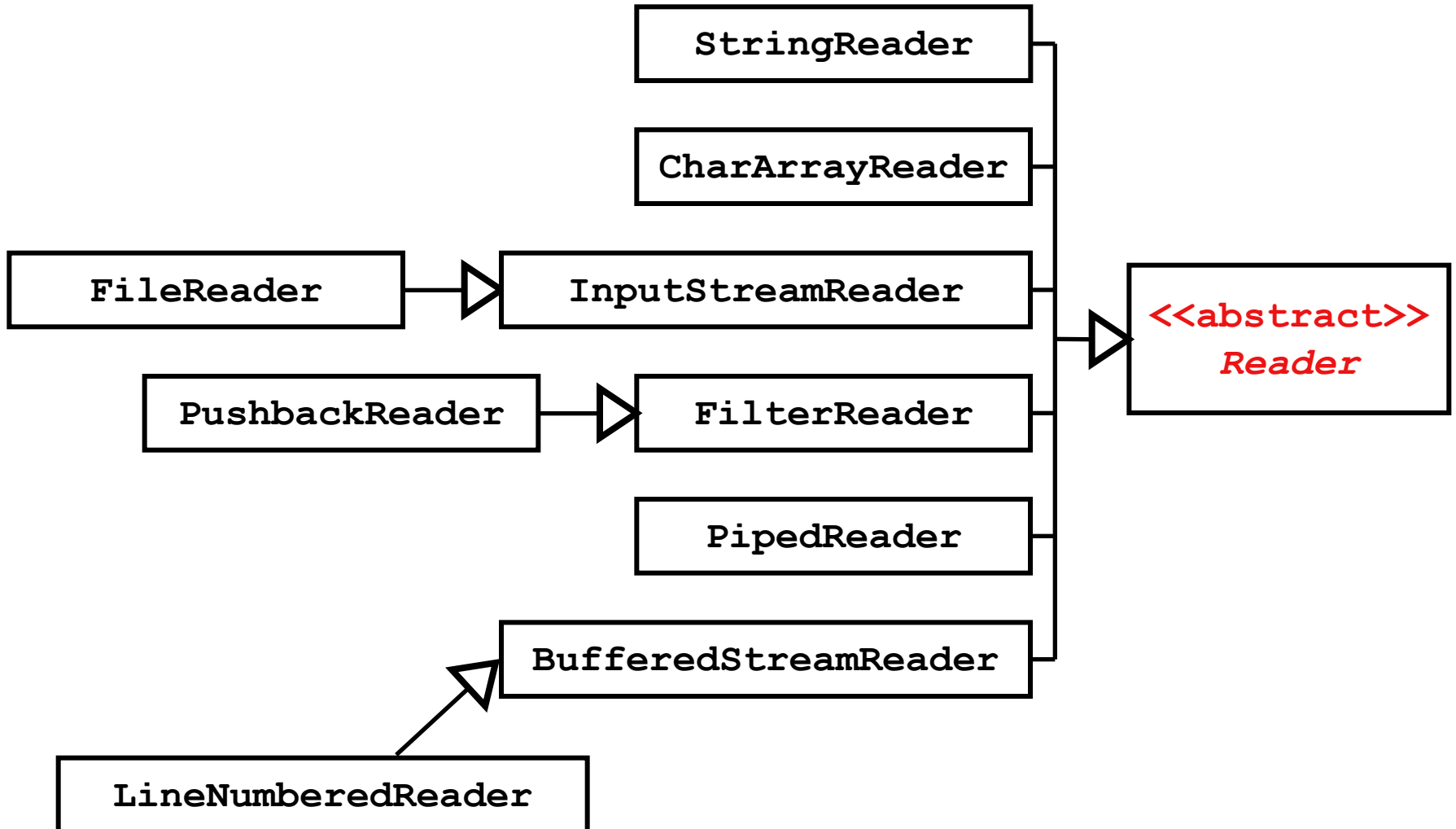
# Incapsulamento dei flussi

- Le classi **Filter**- non fanno niente!
  - ▣ Per default prendono uno stream (passato come parametro al loro costruttore) e gli girano i dati senza fare modifiche.
- Vanno estese a sottoclassi che fanno qualcosa.
  - ▣ Ad esempio **ZipInputStream** e **ZipOutputStream** (del package `java.util.zip`) estendono **Filter (Input/Output) Stream** e leggono file ZIP
  - ▣ **PushbackReader** estende **FilterReader** e reimmette i caratteri letti nel flusso (pushback).
- Casi particolari di incapsulamento sono poi dati dalle classi: **PrintWriter**, **BufferedReader**

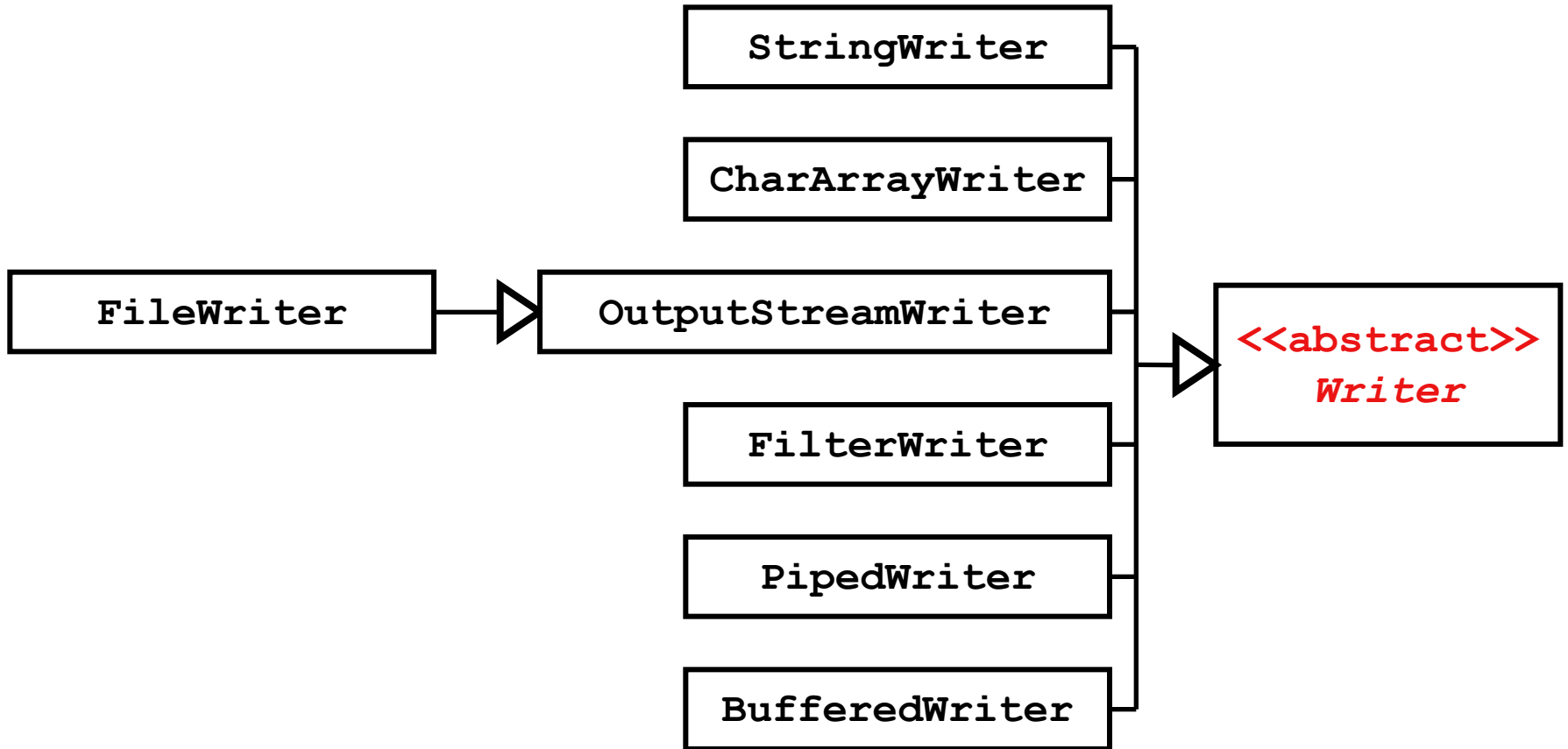
# Lettura/Scrittura verso file di testo

- Per gestire testi, Java fornisce due gerarchie, in lettura (**Reader**) e in scrittura (**Writer**).
- **Reader** e **Writer** sono due classi astratte che servono a definire i metodi di base per lettura e scrittura da file. Sono compresi:
  - ▣ `flush()` / `close()` : scarica/scarica+chiude flusso
  - ▣ `read()` / `write()` : leggi/scrivi pacchetti di **char**
- L'implementazione specifica di questi metodi dipende dall'estensione che si usa.

# Gerarchia di Reader





# Gerarchia di `Writer`



# Scrittura su file di testo

- Per aprire un file di testo, tipicamente si crea un oggetto di tipo **FileWriter**.
- La classe **FileWriter** ha due costruttori con un parametro: un oggetto di tipo **File** o anche una stringa (il nome del file su cui scrivere).
- A sua volta, **FileWriter** è incapsulato in un oggetto di tipo **PrintWriter**. Quindi:

```
PrintWriter w =  =  si scrive su questo  
new PrintWriter( new FileWriter("a.dat") );
```

# Scrittura su file di testo

- La classe **PrintWriter** è contenuta nel package **java.io** e serve a fornire gli stessi metodi della classe **PrintStream** (visti ad esempio per la sua istanza **System.out**).
  - ▣ Solo, invece di stampare testo a video, lo scrivono sul file associato.
- Quindi si possono utilizzare i metodi **print()** e **println()** di **PrintWriter** per scrivere caratteri in un file aperto con successo.

# Il metodo `printf`

- `print` e `println` sono metodi versatili: sono in grado di stampare tipi diversi di dato.
- `PrintWriter` (e `PrintStream`) hanno anche un metodo di stampa formattata: `printf`.
  - ▣ Numero variabile di argomenti: sempre almeno uno, di tipo stringa, e i successivi opzionali per sostituire le parti % della stringa.
- Notare che i metodi di `PrintWriter` non sollevano eccezioni. Esiste un metodo d'istanza `checkError()`.



# Esempio di scrittura file di testo

```
public static void stampaCostanti()
{ f = new File("costanti.txt");
  if ( f.exists() )
  { System.out.println("costanti.txt esiste!");
    return;
  }
  String[] nomi = { "Pi greco", "Nepero" };
  double[] valori = { Math.PI, Math.E };
  FileWriter fw = new FileWriter(f);
  PrintWriter pw = new PrintWriter(fw);
  for ( int i = 0; i<nomi.length; i++)
  { pw.printf("%s è:%10.6f",nomi[i],valori[i]);
    }
  close(f);
}
```

# Lettura da file di testo

- Analogamente alla scrittura, per leggere da file si creerà invece un oggetto **FileReader**.
- Anche in questo caso il costruttore può ricevere un parametro **File** o stringa.
- Tuttavia, un **FileReader** dovrebbe leggere un carattere alla volta, quindi di solito viene incapsulato in un oggetto **BufferedReader**:

```
BufferedReader r =  si legge da questo  
new BufferedReader(new FileReader("a.dat"));
```

# Lettura da file di testo

- Da notare che **FileReader** è una sottoclasse di **InputStreamReader**: per incapsularlo dentro un **BufferedReader** la procedura è identica a quanto visto per il **System.in**.
- Se il file è in formato testo, ma si vogliono leggere in modo più efficiente i dati contenuti, si può anche pensare di usare uno **Scanner**.
  - ▣ Lo **Scanner** si può creare da qualunque oggetto che implementa l'interfaccia **Readable**, quindi anche direttamente da un **File**.

# Esempio di lettura file di testo

```
public static void stampaIlFile()  
{ f = new File("a.txt");  
  if ( !f.exists() )  
  { System.out.println("a.txt non esiste!");  
    return;  
  }  
  FileReader fr = new FileReader(f);  
  BufferedReader re = new BufferedReader(fr);  
  String linea = re.readLine();  
  while (linea != null)  
  { System.out.println(linea);  
    linea = re.readLine();  
  }  
  close(f);  
}
```

# Eccezioni

- La lettura/scrittura su file deve prevedere la possibilità che l'azione non vada a buon fine.
  - ▣ File errato, non trovato, disco pieno, con errori...
- Sono pertanto previste opportune **eccezioni**
- Esiste un albero di ereditarietà che parte da **`IOException`**, sottoclasse diretta di **`Exception`**: quindi è un'**eccezione controllata**.
- Per errori molto gravi, esiste anche **`IOException`** (estensione di **`Throwable`**, non controllata).

# Eccezioni

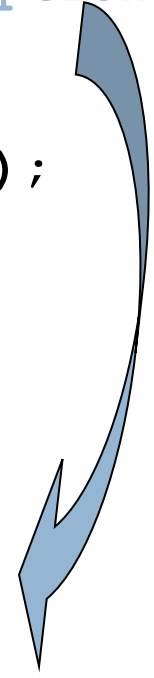
- Esempi di sottoclassi di **`IOException`** :
  - ▣ **`FileNotFoundException`** (non c'è il file)
  - ▣ **`EOFException`** (si legge dopo la fine file) ...
- Controllate: vanno annunciate con **`throws`**.
- Oppure ancora meglio ogni operazione di (lettura da – scrittura verso) file può essere racchiusa dentro opportune **`try . . catch`**.
- Vediamo un esempio: leggiamo numeri interi da **`valori.txt`** e ne stampiamo la somma su **`somma.txt`**: varie cose possono andare male.

# Esempio di filtro file di testo

```
import java.io.*;
public class Prova
{ public static void main(String[] args) throws IOException
  { try
    { BufferedReader in =
      new BufferedReader(new FileReader("valori.txt"));
      String linea = in.readLine(); int somma = 0;
      while (linea != null)
      { somma += Integer.parseInt(linea);
        linea = in.readLine();
      }
      in.close();
    } catch (NumberFormatException e) {
      System.err.println("Errore formato, linea "+linea);
    }
    System.out.println("Il file è valido.");
  }
}
```

# Esempio di filtro file di testo

```
import java.io.*;
public class Prova
{ public static void main(String[] args) throws IOException
  { try
    { BufferedReader in =
      new BufferedReader(new FileReader("valori.txt"));
      String linea = in.readLine(); int somma = 0;
      while (linea != null) { ... }
      in.close();
    } catch (NumberFormatException e) { ...
    } catch (FileNotFoundException e) {
      System.err.println("Manca il file valori.txt");
    } catch (IOException e) {
      System.err.println(e); throws( new IOError() );
    }
    System.out.println("Il file è valido.");
  }
}
```





# Esempio di filtro file di testo


```
import java.io.*;
public class Prova
{ public static void main(String[] args)
  { try
    { BufferedReader in = ...
      in.close();
    } catch (NumberFormatException e) {...}
      catch (FileNotFoundException e) {...}
      catch (IOException e) {...}
    try
    { PrintWriter out =
      new PrintWriter(new FileWriter("somma.txt"));
      out.println("La somma è "+somma);
      out.close();
    } catch (IOException e) {
      System.err.println("Errore in scrittura: "+e);
    }
  }
}
```

# Chiudere il flusso

- In realtà la soluzione prospettata nell'esempio ancora non va del tutto bene.
- Il flusso va sempre chiuso: lasciarlo aperto provoca errori di sincronismo sul file system, e potenzialmente mancata scrittura/lettura di dati
- Però la chiusura dei flussi non va messa dentro la try, o in caso di eccezioni il flusso rimarrà aperto.
- Soluzione perfetta: usare la clausola **finally**

# Esempio di filtro file di testo

```
import java.io.*;
public class Prova
{ public static void main(String[] args)
  { try
    { BufferedReader in = ...
      in.close();
    } catch(NumberFormatException e) {...}
      catch(FileNotFoundException e) {...}
      catch(IOException e) {...}
      finally { in.close(); }
    try
    { PrintWriter out = ...
      out.close();
    } catch(IOException e) { ... }
      finally { in.close(); }
    }
  }
}
```



# Scrittura file in modalità append

- Di default il contenuto di un file viene sovrascritto. Se invece volessimo inserire nuovi caratteri in fondo a un file preesistente esistono ulteriori costruttori:

**`FileWriter(File, boolean)`**

**`FileWriter(String, boolean)`**

che chiedono se vogliamo fare un append (in tal caso la variabile `boolean` = `true`).

- Attenzione a controllare le ulteriori eccezioni!

# Lettura/Scrittura verso file binari

- Supponiamo di voler salvare dati per accederli in seguito (**persistenza**): l'unica soluzione fin qua disponibile è convertire in testo.
  - ▣ Scomodo, e in certi casi impossibile per i tipi riferimento che contengono riferimenti incrociati.
- Java in realtà mette a disposizione anche scrittura/lettura in formato binario.
  - ▣ Analogo al formato testo, problemi compresi!
  - ▣ Vale per qualunque oggetto **serializzabile**.

# Lettura/Scrittura verso file binari

- Come per **Reader/Writer** ci sono gerarchie di ereditarietà analoghe per il formato binario.
  - ▣ Grossomodo uguali con “**Reader**” e “**Writer**” sostituiti da “**InputStream**” e “**OutputStream**”
- Alla base: **InputStream** e **OutputStream**, due classi astratte che danno i metodi di base per lettura e scrittura da file.
  - ▣ Ci sono sempre **flush()** / **close()** e anche **read()** / **write()**; questi ultimi però leggono e scrivono byte (il meno significativo di un **int**)

# Gerarchia di `-Stream`

- In realtà le sottoclassi di `InputStream` e `OutputStream` implementano in modo specifico questi metodi astratti
  - ▣ In particolare, si possono associare flussi a file usando `File (Input/Output) Stream`; hanno i soliti costruttori (una stringa o un `File`, e si può mettere un boolean per dire se si fa append)
- I file ottenuti in questo modo non saranno direttamente leggibili con un editor di testo!

# Salvare dati primitivi

- Una volta creato il file, si possono salvare ad esempio dati di tipi primitivi se incapsuliamo il file dentro un **DataOutputStream**.
  - ▣ Questa classe estende **FilterOutputStream** (che è sottoclasse di **OutputStream**) perciò è un vero e proprio incapsulamento.
  - ▣ Accanto a **write()** che scrive byte ci sono **writeInt()**, **writeDouble()** ... (ogni tipo primitivo)
  - ▣ Per salvare numeri reali **writeDouble()** dà maggiore precisione di una conversione a testo.



# Caricare dati primitivi

- Allo stesso modo si possono leggere dati usando un **DataInputStream**, sottoclasse di **FilterInputStream**, che incapsula il flusso.
- Possiamo usare `readInt()`, `readDouble()`...
- Il problema però è che i dati sono stati salvati come byte: è quindi importante ricordarsi l'ordine in cui li leggiamo!
- Non c'è alternativa migliore: leggere i dati in ordine sbagliato è lecito (anche se errato).

# Serializzazione

- Possiamo però salvare oggetti?
  - ▣ Se per i tipi primitivi la conversione a testo è possibile (anche se può essere scomoda), per gli oggetti (tipo riferimento) è in generale impossibile per via dei riferimenti incrociati.
- La risposta è affermativa se gli oggetti rispettano la proprietà di **serializzazione**.
- Letteralmente: possibilità di conversione in sequenze di byte (requisito indispensabile per essere scritti in formato binario)

# Serializzazione

- In effetti in Java imporre che un oggetto rispetti la serializzazione è abbastanza semplice: basta imporre di implementare l'interfaccia **Serializable**.
- È un'**interfaccia marker**: non ha metodi.
  - ▣ Quindi qualunque classe la può implementare. In realtà ci sono dei problemi ma solo per le classi che estendono classi non serializzabili: in tal caso bisogna che queste classi abbiano un costruttore accessibile e senza parametri.

# Serializzazione

- Implementare **Serializable** ha dei rischi: per esempio rende “pubblici” i membri privati.
- Se lo si fa, è poi bene inserire un identificatore universale **serialVersionUID** di tipo **long**.

```
private static final long serialVersionUID =  
    7526472295622776147L;
```

- ▣ Se non lo si fa, si riceve un warning.
- Il numero serve a garantire che la classe coincide quando si deserializza. Cambiandolo, si rende incompatibile la nuova versione.

# Serializzazione

- Implementando la serializzazione non ci saranno problemi di dipendenze cicliche.
  - ▣ Gli oggetti salvati verranno scritti una volta sola sul flusso. Inoltre spesso salvando un oggetto di una certa classe anche molto di ciò che la classe “usa” viene salvato anch'esso.
- Nel caso si scrivano oggetti (quindi di diverse possibili classi) resta ancora però il problema di doverli “riescare” dal flusso nello stesso ordine in cui li si era scritti.

# File ad accesso casuale

- Un'alternativa all'accesso come flusso di byte in sequenza è dato dall'**accesso casuale**.
- In questa versione l'accesso è scandito da un cursore e può andare in entrambe le direzioni.
  - ▣ Inoltre gestisce anche l'accesso read/write.
- La classe **RandomAccessFile** consente di vedere un file del file system in questo modo.
- Il costruttore ha due parametri: stringa/file e un'ulteriore stringa (che può essere **r**, **w**, **rw** )

# File ad accesso casuale

- I metodi della classe sono come per la classe **Data (Input/Output) Stream**:
  - ▣ **readInt**, **readDouble**, ... per leggere
  - ▣ **writeInt**, **writeDouble**, ... per scrivere  
ma li ha entrambi!
- L'accesso casuale è realizzato con il metodo **getFilePointer** che torna un **long** con la posizione del byte corrente e il metodo **seek (pos)** che sposta il cursore di **pos**