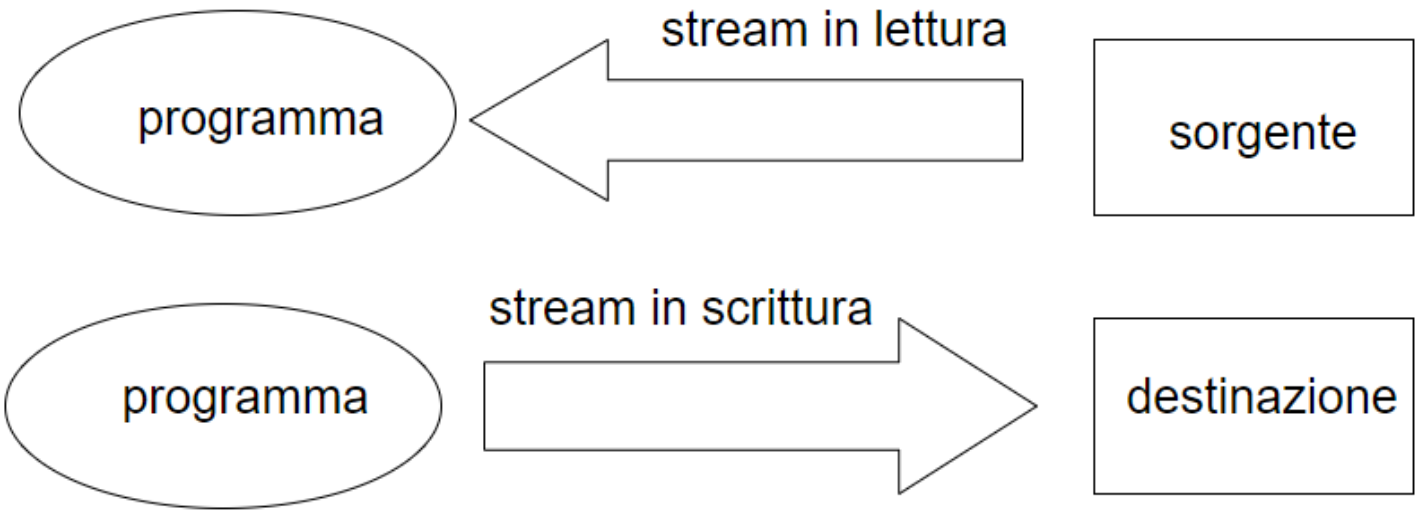
Programmazione III

Prof.ssa Liliana Ardissono
Dipartimento di Informatica
Università di Torino

Input/Output

Le operazioni di I/O avvengono attraverso **stream**

Flussi o streams: sono successioni di informazioni in forma elementare, principalmente sequenze di bytes. I files sono un caso particolare di flussi.



Sorgente e destinazione possono essere di vario genere (file, connessioni di rete, …) ma vengono gestite essenzialmente nello stesso modo.

Java fornisce molte classi per I/O (circa 60).

Tutti i file sono memorizzati come sequenze di bit (0/1). Tuttavia in alcune situazioni si interpreta il contenuto di un file come sequenza di caratteri (file di testo), in altre come sequenza di byte (file binari).

Divise in due gerarchie:

- basate su caratteri Unicode (2 byte) (derivate dalle classi astratte **Reader** e **Writer**)
- basate su byte (derivate dalle classi astratte **InputStream** e **OutputStream**)

Tutti i file sono memorizzati come sequenze di bit (0/1). Nei file di testo la sequenza di bit viene vista come sequenza di caratteri UNICODE, mentre in quelli binari come sequenza di byte. Per es. consideriamo il carattere 5 e il numero 5

Dato il valore (numero) 5:

- Rappresentazione del valore 5 in un file di testo:
  - 00000000 00110101 (corrisponde al 5 UNICODE)
  - Esempio: Laura ha 5 anni
- Rappresentazione del valore 5 in un file binario:
  - 101
  - Esempio: 5+5=10

Le classi di I/O possono essere classificate in base allo scopo:

- tipo di sorgente/destinazione
- tipo di elaborazione sui dati

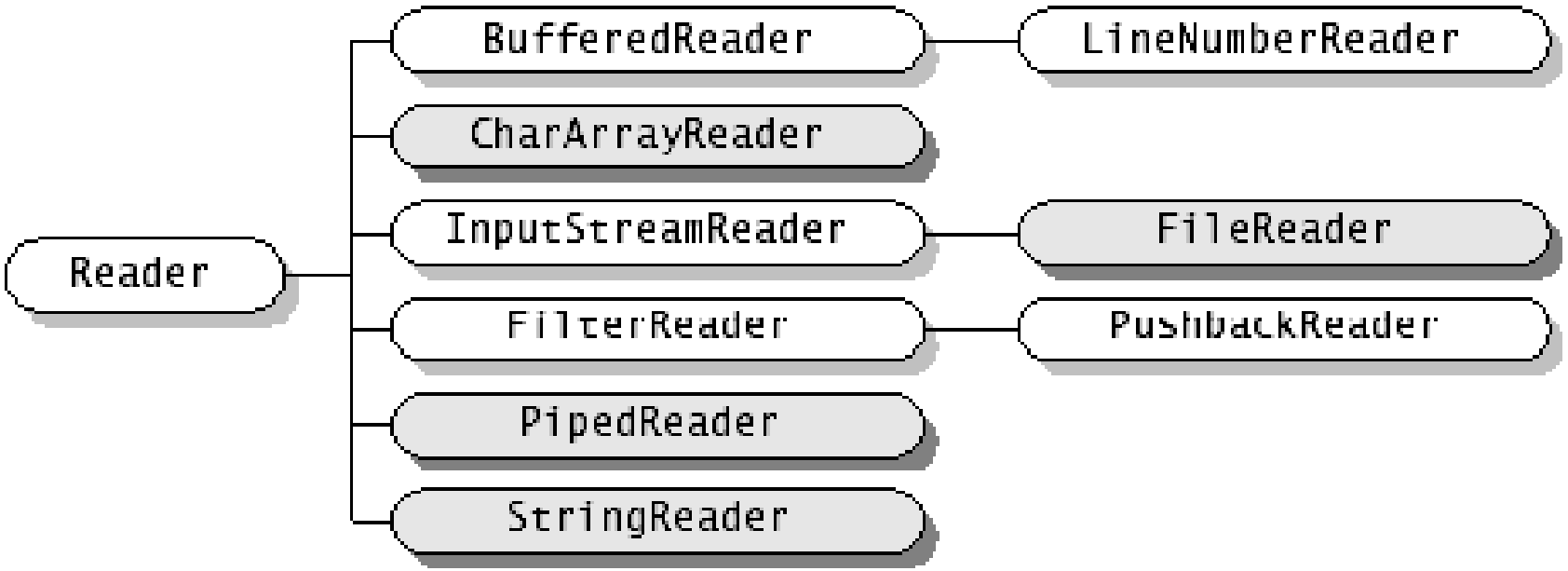
Sorgente o destinazione possono essere di vario tipo

- memoria (array o stringhe),
- *file*
- *pipe* (per collegare due programmi - thread),
- connessioni via socket o internet.

Input a caratteri: **Reader** è una classe astratta.

La classe **Reader** ha metodi per leggere un carattere per volta:

abstract int read()



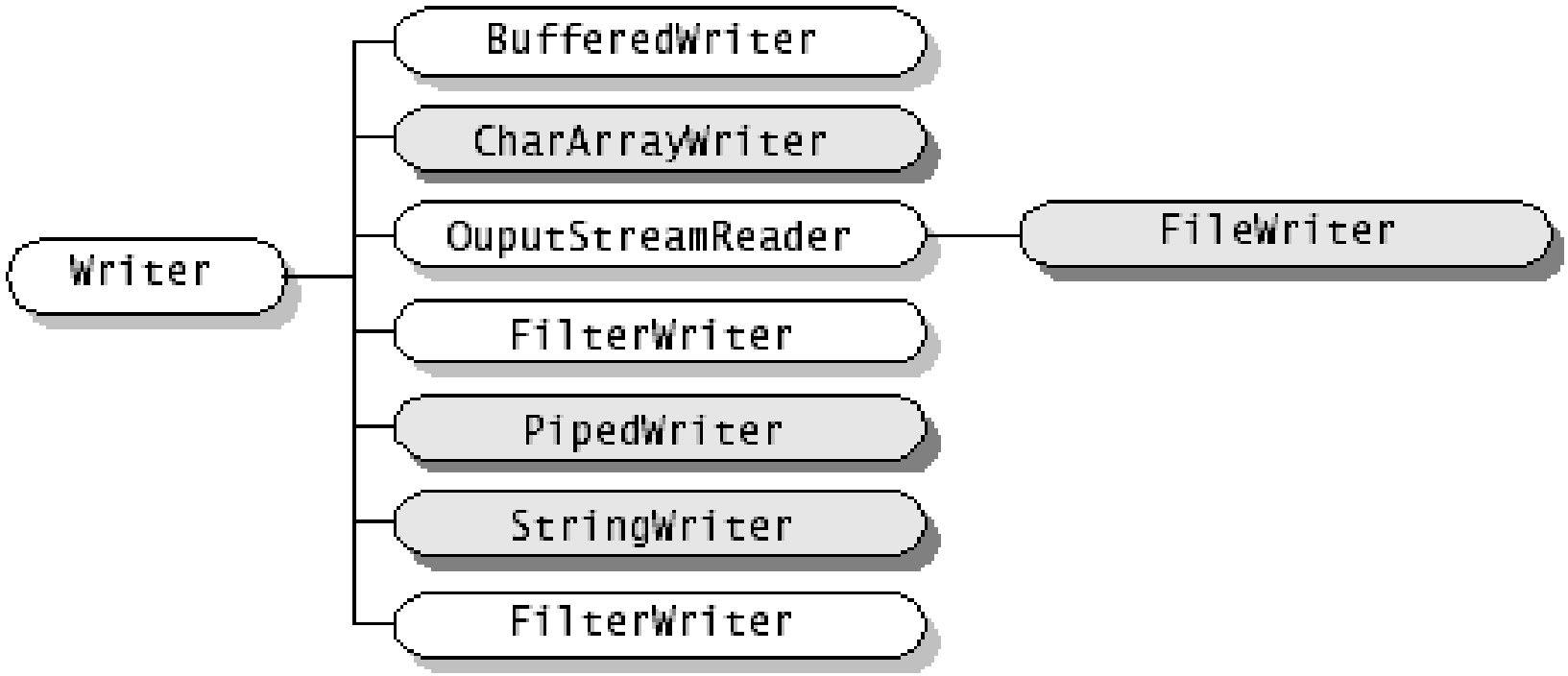
In grigio gli *stream* che leggono da una sorgente.

In bianco quelli che eseguono qualche tipo di elaborazione.

Output a caratteri: **Writer** è una classe astratta.

Writer può scrivere un carattere per volta:

abstract void write(int c)



Uso dei buffer per leggere e scrivere dati su file

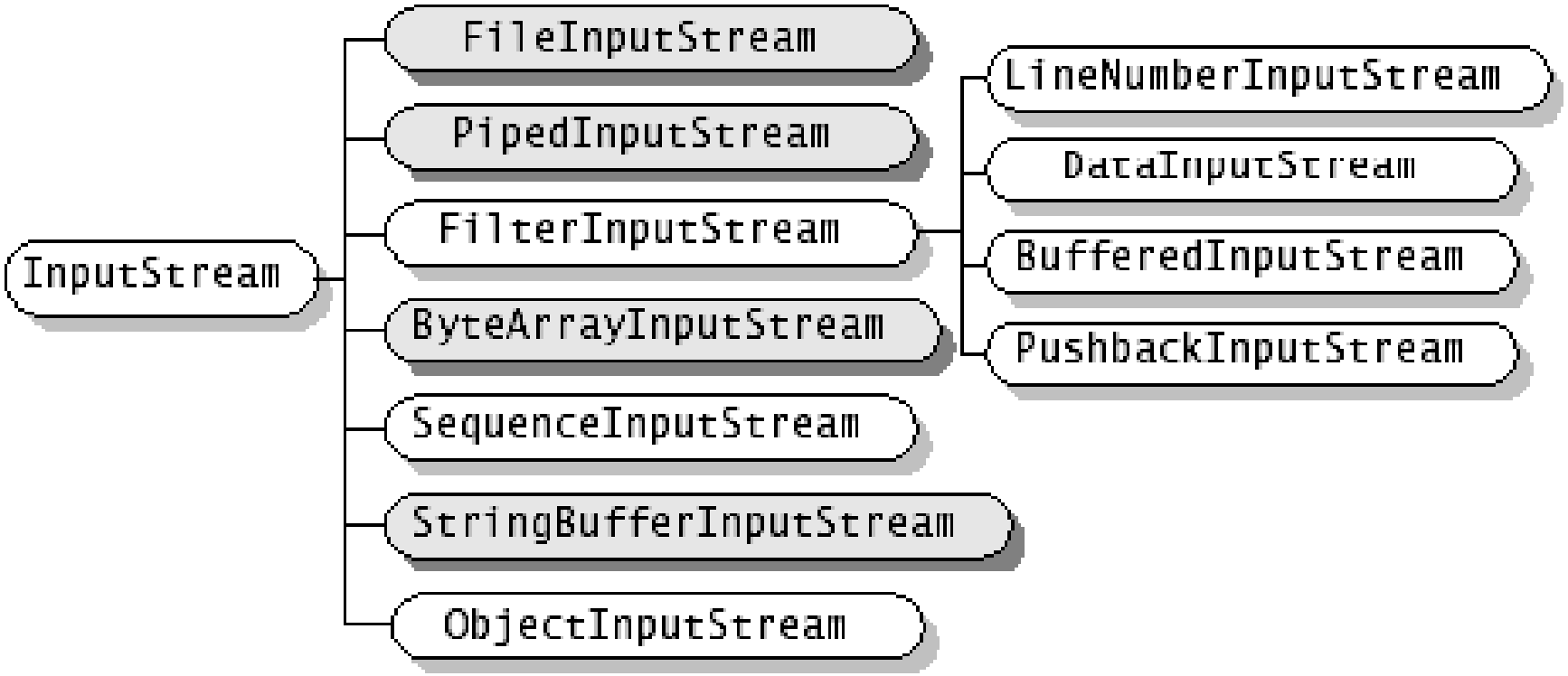
Per ottimizzare i tempi di accesso ad una risorsa è opportuno leggere e scrivere molti caratteri in una sola volta e memorizzarli in un buffer, in modo che la prossima lettura/scrittura prelevi/scriva il carattere dal buffer anziché richiedere un nuovo accesso alla risorsa

➔ introdotte le classi **BufferedReader** e **BufferedWriter** per usare buffer in lettura e scrittura (e analogamente esistono classi di buffering per la lettura/scrittura di byte)

Input a byte: **InputStream** è una classe astratta.

InputStream può leggere un byte per volta:

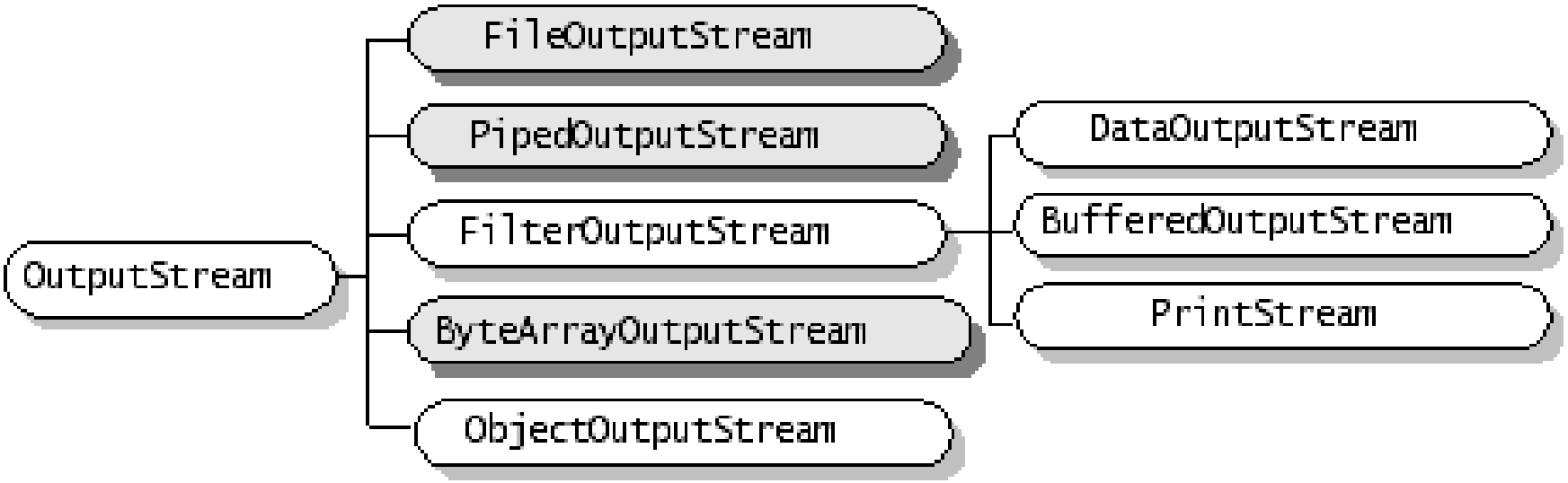
abstract void read(int c)



Output a byte: **OutputStream** è una classe astratta.

OutputStream può scrivere un byte per volta:

abstract void write(int c)



Flussi standard

I flussi standard (tastiera e video) per motivi storici sono flussi di byte:

- **System.in** ha tipo InputStream
- **System.out** ha tipo PrintStream, che è sottotipo di OutputStream
- **System.err** ha tipo PrintStream

Input e output da file (a caratteri)

La classe **FileReader** può essere usata per leggere da file, specificando il nome del file nel costruttore:

```
FileReader in = new FileReader("esempio.txt");
```

Analogamente per scrivere:

```
FileWriter out = new FileWriter("copia.txt");
```

FileReader e **FileWriter** sono sottoclassi di **Reader** e **Writer**. **FileReader** converte da uno stream di byte contenuti nel file in uno stream di caratteri, e viceversa per **FileWriter**.

```
int c = in.read();
out.write(c);
```

Lettura e scrittura di stream di dati

Il processo di lettura di dati da uno stream può essere sintetizzato come segue:

```
apri lo stream
while (ci sono dati da leggere nello stream)
    leggi un dato dallo stream
chiudi lo stream
```

Il processo di scrittura è schematizzato nel modo seguente:

```
apri lo stream
while (ci sono dati da scrivere)
    scrivi il dato nello stream
chiudi lo stream
```

Esempio: programma che copia un file in un altro file leggendo e scrivendo carattere per carattere.

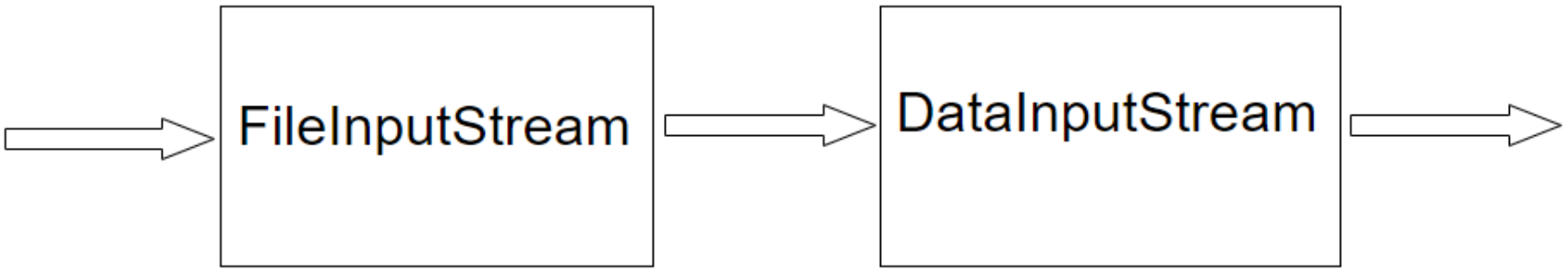
```
public class Copy {
   public static void main(String[] args) throws IOException {

      FileReader in = new FileReader("esempio.txt");
      FileWriter out = new FileWriter("copia.txt");
      int c;

      while ((c = in.read()) != -1)   // -1 è l'EOF
         out.write(c);

      in.close();
      out.close();
   }
}
```

Gli *stream* che specificano il tipo di sorgente, come ad es. **FileInputStream**, non hanno metodi per leggere tipi primitivi (numeri, ...) - leggono solo byte o caratteri.

Altri *stream*, derivati da **FilterInputStream**, possono assemblare i byte in tipi di dati. Es. DataInputStream (vd. poi)

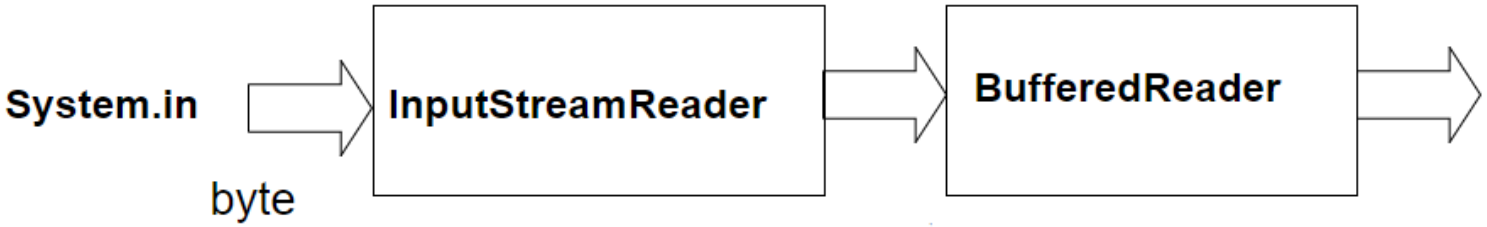
Analogamente per l'output.

Gli *stream* possono essere composti per manipolare i dati:



Composizione di stream:

Lo «standard input» (System.in) ha tipo InputStream ➔ si legge usando un InputStreamReader.




```
BufferedReader in = new BufferedReader(
                    new InputStreamReader(System.in));
```

Composizione di stream

```
BufferedReader in = new BufferedReader(
                    new InputStreamReader(System.in));
```

La classe **InputStreamReader** trasforma uno stream di input che contiene byte in un *reader* che emette caratteri Unicode.

Componendo **BufferedReader** con **InputStreamReader** si ottiene uno stream di input che usa un buffer:

```
DataInputStream in = new DataInputStream(
        new BufferedInputStream(
                new FileInputStream("impiegati.dat")))
```

Esempio:

```
public class CopiaFileCaratteriConBuffer {

    public static void main(String[] args) throws IOException {
        BufferedReader in =
                new BufferedReader(new FileReader("esempio.txt"));
        BufferedWriter out =
                    new BufferedWriter(new FileWriter("copia.txt"));
        int c;
        while ((c = in.read()) != -1)
            out.write(c);
        out.flush();
        in.close();
        out.close();
    }
}
```

I/O di tipi di dati primitivi

Le classi **DataInputStream** e **DataOutputStream** forniscono metodi per leggere e scrivere dati primitivi

I dati vengono codificati in formato binario e non sono leggibili come testi.

```
DataOutputStream dout = new DataOutputStream(
        new FileOutputStream("prova.dat"));
dout.writeInt(250);
dout.writeDouble(3.14);
dout.writeChar('a');
dout.close();
DataInputStream din = new DataInputStream(
        new FileInputStream("prova.dat"));
System.out.println(din.readInt());
System.out.println(din.readDouble());
System.out.println(din.readChar());
```

I/O di oggetti

Con **ObjectInputStream** e **ObjectOutputStream** è possibile leggere o scrivere qualunque **oggetto** purché la sua classe implementi l'interfaccia **Serializable.**

```
ObjectOutputStream out = new ObjectOutputStream(
        new FileOutputStream("impiegati.dat"));
Impiegato rossi = new Impiegato(.....);
out.writeObject(rossi);
ObjectInputStream in = new ObjectInputStream(
        new FileInputStream("impiegati.dat"));
Impiegato imp = (Impiegato)in.readObject();
```

```
class Impiegato implements Serializable {...}
```

Interface Serializable

L'interfaccia **Serializable, predefinita,** non contiene nessun metodo.

Serve da marcatore di classi di oggetti che possono essere trasmessi negli stream. ObjectOutputStream non emette alcun oggetto che non sia Serializable. Gli oggetti Serializable vengono salvati con tanto di informazioni sulla definizione della loro classe, per permetterne la ricostruzione (deserializzazione).

Un oggetto Serializable non può contenere variabili di istanza che non lo siano, oppure le deve marcare come ***transient*** per indicare che non sono serializzabili e quindi verranno ignorate nel processo di Serializzazione. Tutti i tipi primitivi sono serializable per default.

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

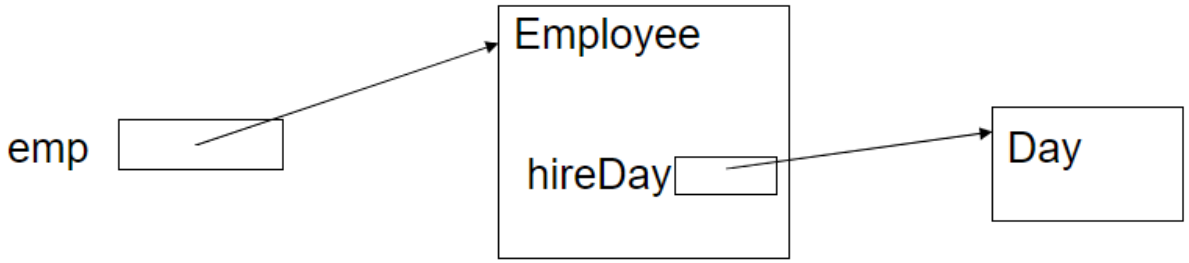
private void readObject(java.io.ObjectInputStream in) throws IOException,

ClassNotFoundException;

private void readObjectNoData() throws ObjectStreamException;

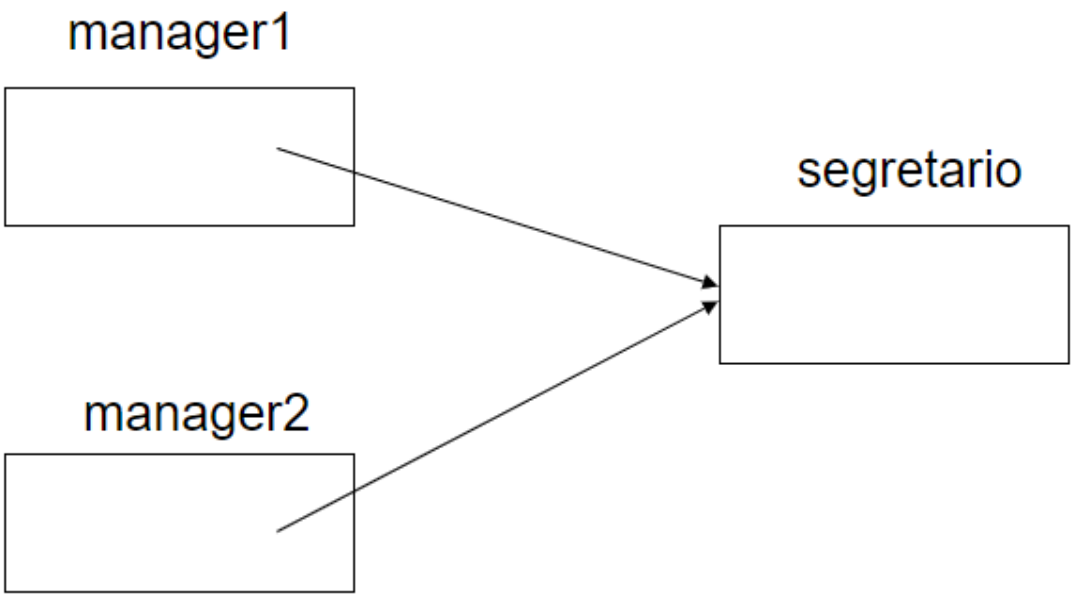
Cosa succede se si scrive un oggetto che contiene riferimenti ad altri oggetti?

Vengono scritti anche tutti gli altri oggetti raggiungibili.



Se si esegue **out.writeObject(emp)** vengono scritti in **out** sia l'oggetto **Employee** che l'oggetto **Day**.

Se un oggetto è condiviso, ne viene scritta solo una copia.



File

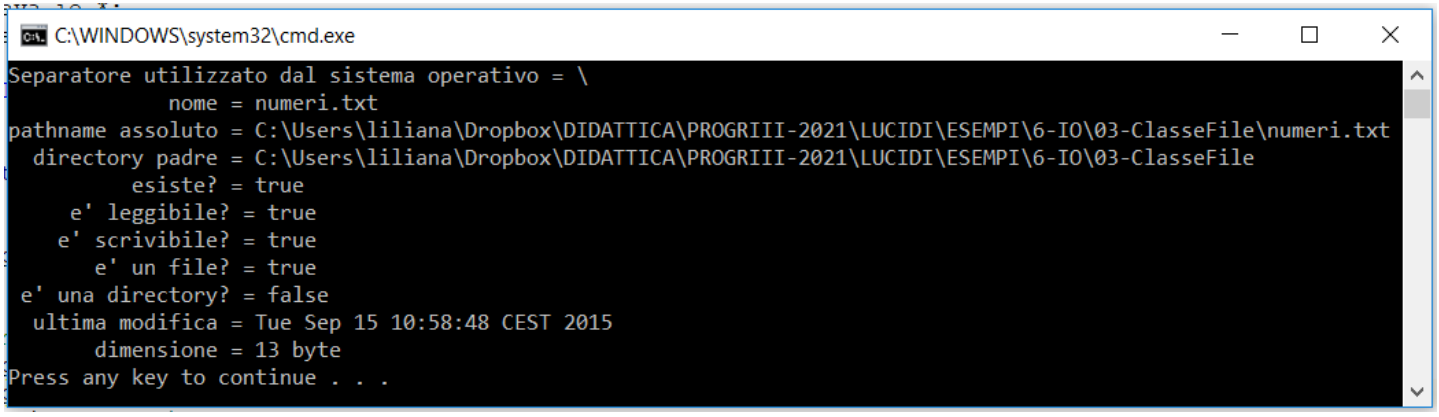
File rappresenta i file come oggetti (nascondendo dettagli del file system)**.** Si può passare un file come parametro al costruttore di **FileReader** per creare un lettore che lavora sul file corrispondente:

```
File inputFile = new File("esempio.txt");
FileReader in = new FileReader(inputFile);
```

La classe **File** può anche essere usata per gestire directory.

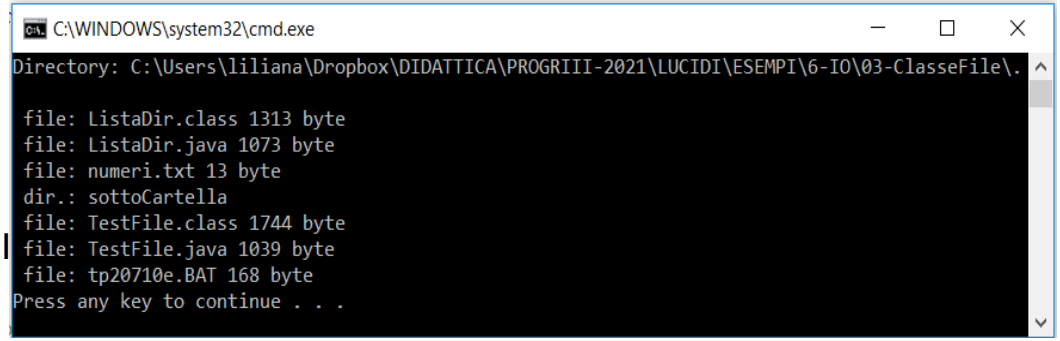
File: gestione di file

```
public class TestFile {

  public static void main(String[] args) throws IOException {
    PrintStream out = System.out;
    out.println("Separatore utilizzato dal sistema operativo = " + File.separator);
    File file = new File(args[0]);
    out.println("             nome = " + file.getName());
    out.println("pathname assoluto = " + file.getAbsolutePath());
    out.println("  directory padre = " +
                       (new File(file.getAbsolutePath()).getParent()));
    out.println("          esiste? = " + file.exists());
    out.println("     e' leggibile? = " + file.canRead());
    out.println("    e' scrivibile? = " + file.canWrite());
    out.println("       e' un file? = " + file.isFile());
    out.println(" e' una directory? = " + file.isDirectory());
    out.println("  ultima modifica = " +
                       (new Date(file.lastModified())).toString());
    out.println("       dimensione = " + file.length() + " byte");
  }
}
```



File: gestione di directory

```
public class ListaDir {
  public static void main(String[] args) throws IOException {
    PrintStream out = System.out; //predisposizione del canale di output
    File f = null;
    if (args.length == 0) //se non ci sono argomenti consideriamo la directory corrente
        f = new File(".");
    else
        f = new File(args[0]);

    if (f.isFile())              //se il nome specificato e'...
                                 //...quello di un file allora ne stampa nome e dimensione
      out.println("File: " + f.getAbsolutePath() + ", " + f.length() + " byte");
    else {                       //...quello di una directory allora...
      out.println("Directory: " + f.getAbsolutePath());
      String[] lista = f.list();                        //...preleva la lista dei file...
      for(int i=0; i < lista.length; i++){              //...stampa i dati di ognuno
        File tmp = new File(f.getAbsolutePath(), lista[i]);
        if (tmp.isFile())
          out.println(" file: " + tmp.getName() + " " + tmp.length() + " byte");
        else
          out.println(" dir.: " + tmp.getName());
} } } }
```



Classi Java per I/O da file – da JDK 1.5

- Per file testuali
  - **classi Scanner e File per leggere da file**
  - classe **PrintWriter** per scrivere su file
- Per file binari
  - classe FileInputStream per leggere da file
  - classe FileOutputStream per scrivere su file

Lettura da file testuale - I

Per aprire un file da applicazione java e leggerne i dati

- **Scanner scf = new Scanner(new File(path+“/info.txt"));**
- Scanner s = new Scanner(new File(“C:\\documenti\\info.txt”));

Lo Scanner mi permette di specificare file con path assoluto (notare i doppi backslash)

Lettura da file testuale - II

- **Scanner scf = new Scanner(new File(path+“/info.txt"));**
  - Oggetto File permette di aprire un file senza scrivere operazioni a livello di sistema operativo
  - Oggetto Scanner opera sul file passato come parametro e permette di leggere contenuto (come da standard input)
- File è nel package java.io
- Scanner è nel package java.util

→ importare i package se si usano queste classi

Input/Output da file - eccezioni

Leggere dati da file richiede la gestione di eccezioni

- **FileNotFoundException**: il file specificato non esiste (nome di file sbagliato, path errato, file cancellato per errore, …)
- **InputMismatchException**: dati malformattati - non corrispondono a ciò che applicazione si aspetta (es: si aspetta token int e trova boolean)

$\Rightarrow$ operazioni da eseguire in **try** e **catch** di queste eccezioni

Oppure si catturi IOException: eccezione + generale che si può catturare quando si manipolano file (consigliata in quanto più generica delle altre).

Esempio: visualizzo linee di file - I

```
public class TestScannerPrintFile {
    public static void main(String[] args) {
      String path = «….path….»;
      try {
          Scanner scf = new Scanner(new File(path+“/info.txt"));
          while (scf.hasNextLine()) { //Verifico che ci sia una
                    // linea di file da leggere con l’oggetto Scanner

                    String riga = scf.nextLine(); // leggo la linea

                    System.out.println("RIGA: "+riga); }
          scf.close();  // chiudo l’oggetto Scanner (meglio in un blocco finally)
          } catch (IOException e) {…gestione eccezione …}
    }
}
```

Classe Scanner

L'oggetto Scanner impostato su un file permette di analizzare il contenuto di una linea del file se si sa quali tipi di dati contiene

- Es: se il file contiene una sequenza di numeri interi posso estrarre tali numeri (token) a 1 a 1 con scf.nextInt()
- L'estrazione di token si basa su definizione di **token** e di **simbolo delimitatore** (ciò che separa 1 token dal successivo). Il delimitatore di default è lo spazio vuoto.

Quando si chiude lo scanner (scf.close()) si rilascia la risorsa su cui lavorava (file, String, etc.)

Token e delimitatori - I

- Il delimitatore di base è lo spazio vuoto (uno o + spazi consecutivi sono un solo delimitatore)
  - Es: sequenza di numeri interi (5 token):
    123 24 76 4 534
  - Es: sequenza di stringhe (3 token):
    ciao a tutti
- Per leggere i token presenti in un file bisogna leggere il file, linea per linea, ed estrarre i token a 1 a 1 da ciascuna linea

Token e delimitatori - esempio

```
public class TestScanToken {
    public static void main (String[] args) {
      try {
          Scanner scf = new Scanner(new File("dati1.txt"));

          while (scf.hasNext())
                    System.out.print(scf.nextInt() + " ");
          scf.close();
          } catch (IOException e) {e.printStackTrace();}
    }
}
```

```
3425 342 7777 2 34 12 897 44
222 2222 333 4444 55555
File dati1.txt
```

Token e delimitatori - II

- Si può configurare un oggetto Scanner affinché usi un delimitatore diverso da quello base specificandolo attraverso espressioni regolari

- Es: **1 fish 2 fish red fish blue fish**, voglio trattare **fish** come delimitatore

```
String tmp = “1 fish 2 fish red fish blue fish”;
Scanner s = new Scanner(tmp).useDelimiter(“\\s*fish\\s*“);
System.out.println(s.nextInt());
System.out.println(s.nextInt());
System.out.println(s.next());
System.out.println(s.next());
s.close();
```

Nelle espressioni regolari, \\s rappresenta lo spazio vuoto
→ \\s* significa 0 o più spazi vuoti

Scrittura su file testuale - I

- **PrintWriter p = new PrintWriter(new File(*nomeFile*));**
- **Es: PrintWriter p = new PrintWriter(new File(“risultati.txt”));**
- Se nella cartella corrente esiste già un file di nome *nomeFile*, viene aperto tale file. Altrimenti, viene creato un nuovo file con il nome specificato.
- L’oggetto PrintWriter **p** è ora pronto per scrivere sul file specificato

Scrittura su file testuale - II

- **PrintWriter p = new PrintWriter(new File(“risultati.txt”));**
- PrintWriter offre metodi **print(String s)** e **println(String s)** per stampare stringhe passate come parametro

  p.println(23); p.println(29); …
- NB: la scrittura avviene su buffer di output, per ragioni di efficienza ⇒ per scrivere effettivamente sul file bisogna svuotare il buffer: **p.flush()**

23
29
File risultati.txt

NB: i numeri sono stati convertiti in String …

Scrittura su file testuale - esempio

```
public static void main (String[] args) {
    PrintWriter p = null;
    try {
            p = new PrintWriter(new File("numeri.txt"));
            for (int i=0; i<5; i++) {
                    int num = Lettura.leggiIntero("digita numero: ");
                    p.println(num);
                    p.flush();  }
    }
    catch(IOException | RuntimeException e) {
                                            System.out.println(e.getMessage()); }
        // NB: metto la chiusura del file in finally per essere certa che
        //  se ci sono problemi venga chiuso il file (che conterrà
        //  quanto inserito fino a prima dell'eccezione).
    finally {if (p!=null)  //NB: se fallisce la new, p è null
                p.close(); }
 }
}
```

NB: chiudere il PrintWriter a termine scrittura.

Ringraziamenti

Grazie al Prof. Emerito Alberto Martelli e al Prof. Matteo Baldoni del Dipartimento di Informatica dell’Università di Torino per aver prodotto parte del material presentato in queste slides.