Università degli Studi di Modena e Reggio Emilia

Facoltà di Ingegneria

CORSO DI

RETI DI CALCOLATORI
Linguaggio Java: Il Package di Input Output

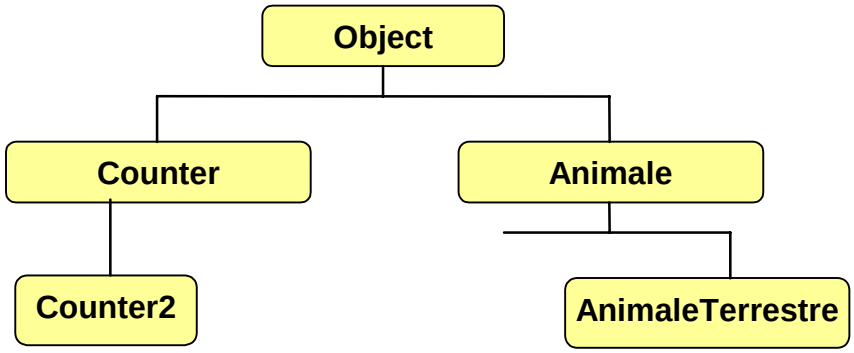
Prof. Franco Zambonelli

Lucidi realizzati in collaborazione con
Ing. Enrico Denti - Univ. Bologna

JAVA: L'ARCHITETTURA

Java non è solo un linguaggio: è una *architettura, indipendente dalla piattaforma e già pronta per l'uso,* che fornisce package, classi e funzionalità per realizzare applicazioni.

Tutte le classi Java appartengono a un'unica tassonomia, alla cui radice c'è `Object`.



Quando si dichiara una classe "apparentemente stand-alone" in Java, la frase `extends Object` è sottintesa.

Questo consente di progettare software generico anche in assenza dei *template* stile C++.

Package principali:

- Package grafico (indipendente dalla piattaforma!): AWT
- Programmazione a eventi (molto evoluta!)
- Supporto di rete: URL, connessioni via Socket, …
- Supporto per il multi-threading
- Concetto di APPLET = piccola (?) applicazione da eseguirsi dentro un browser Internet
- Supporto per sicurezza
- Modello di sicurezza "sandbox" per le applet
- Cifratura, crittografia, chiavi pubbliche/private…
- Connessione e interfacciamento con database

IL PACKAGE `java.io`

Questo package definisce fondamentalmente 4 classi base astratte:

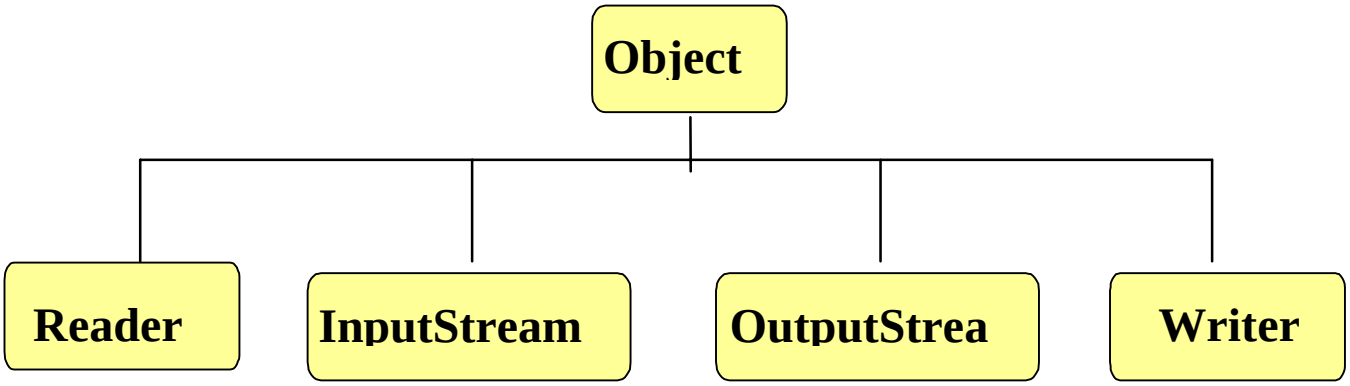
- `InputStream` e `OutputStream` per leggere/scrivere ***stream di byte*** (come i *file binari* del C). I byte non coincidono con i caratteri
- `Reader` e `Writer` per leggere/scrivere *stream di caratteri* (come i *file di testo* del C - solo da Java 1.1.x in poi). Servono classi specializzate perché i caratteri Java sono a due byte!!!

Ognuna di queste classi astratte dà origine a una varietà di classi "concrete", che leggono/scrivono tipi specifici di stream:

Da notare che è necessario trattare separatamente i byte dai caratteri, in quanto i caratteri Java non sono byte.

Esempio: se si legge un file di testo come sequenza di byte, Java non riesce a interpretare i byte letti come caratteri.

In seguito, tratteremo separatamente stream di byte e di caratteri.



CONCETTI BASE DELLA GERARCHIA DI IO IN JAVA

Le 4 classi base astratte forniscono solo metodi generici per leggere e scrivere "flussi" non strutturati di dati dall'input e verso l'output. Le funzionalità fornite sono in qualche modo simili, allo stesso livello, delle primitive di accesso ai file in UNIX.

Le classi derivate si dividono in due categorie, specializzate in due sensi:

- da un lato ci sono classi (dette **sorgenti**) che, senza aggiungere funzionalità, **specializzano le classi astratte rispetto alla sorgente/destinazione destinazione** dei flussi.
  - Per l'input, il flusso può diventare un file o un buffer;
  - Per l'output, il flusso può diventare un file o un buffer;

- dall'altro lato, ci sono classi (dette di **filtraggio**) che, di nuovo non preoccupandosi della sorgente/destinazione dei flussi, **specializzano e aumentano le funzionalità delle classi astratte** per fare in modo di poter leggere/scrivere non più soltanto stram di byte o di caratteri, ma dati strutturati, quali:
  - i tipi primitivi di Java;
  - interi oggetti;

  inoltre, ci sono classi che specilizzano le funzionalità permettendo forme complesse di filtraggio ed elaborazione dei dati.

METODO DELL'INCAPSULAMENTO

Per usare l'IO, si tratta di "comporre" in qualche modo tutte le classi concrete che ci interessano in modo da avere un oggetto che presenti tutte le funzionalità richieste. Si utilizza il **metodo dell'incapsulamento**:

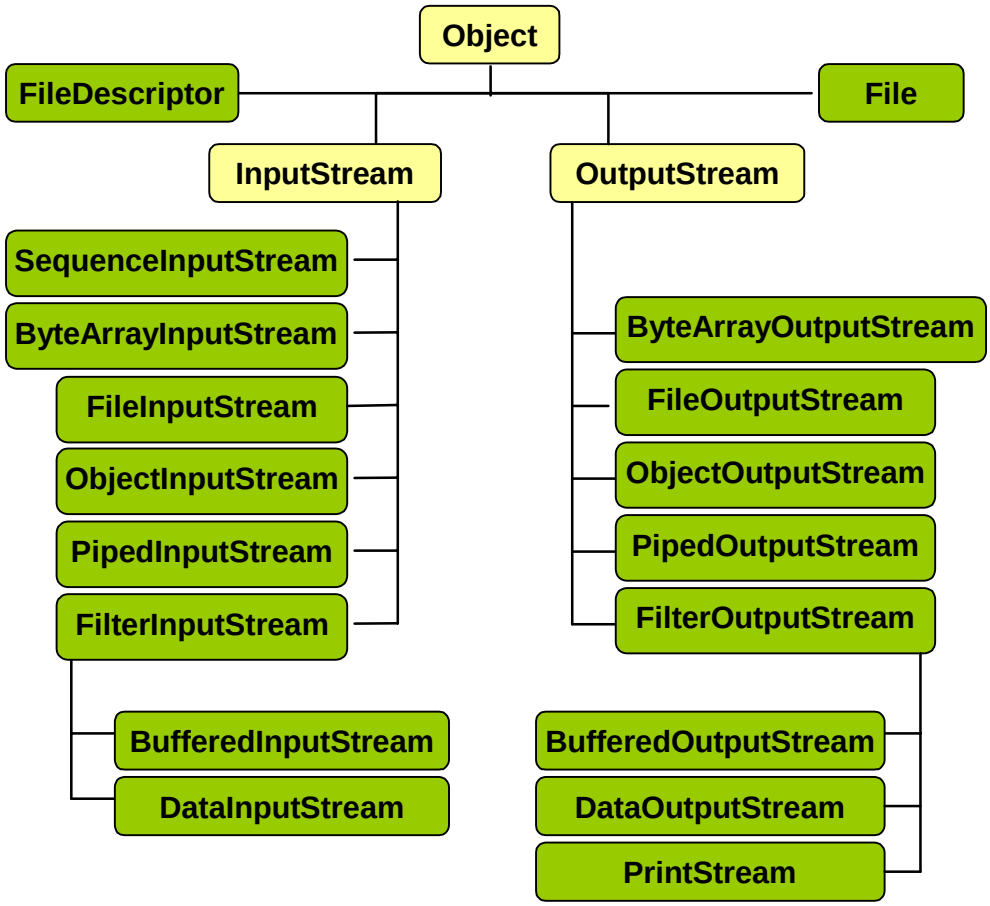
- si crea un oggetto da una classe sorgente, per definire il flusso specifico dei dati;
- si crea un oggetto da una classe di filtraggio concreta del secondo tipo, e sui passa al costruttore l'oggetto stream prima creato (come per incapsularlo dentro)
- si possono poi eseguire ulteriori incapsulamenti della classe di filtraggio in un'altra classe di filtraggio al fine di ottenere, tramite successivi incapsulamenti, tutte le funzionalità previste.

Esempio: (verrà dettagliato meglio in seguito)

```
import java.io.*;
…
    FileOutputStream f = null;
    f = new FileOutputStream("Prova.dat");
//creiamo 1 stream di output associato a un file
// tramite l'apposita classe "sorgente"
    DataOutputStream os = new DataOutputStream(f);
// lo incapsuliamo in una classe di "filtraggio"
// che da' le funzionalità necessarie per
// scrivere i tipi primitivi di Java

      os.writeFloat(f1);
      os.writeBoolean(b1);
      os.writeDouble(d1);
      os.writeChar(c1);
```

LA GERARCHIA DEGLI STREAM DI BYTE



Le classi per l'I/O da stream di byte costituiscono il supporto più generale per l'I/O.

In generale, `InputStream` definisce metodi per:

- leggere uno o più byte (`read()`)
- sapere quanti byte sono già disponibili in input (`available()`)
- saltare N byte in input (`skip()`)
- ricavare la posizione corrente sullo stream (`mark()`) e tornarci (`reset()`), sempreché `markSupported()` sia vero.

`OutputStream` definisce invece metodi per:

- scrivere uno o più byte (`write()`)
- svuotare il buffer di scrittura (`flush()`)
- chiudere lo stream (`close()`)

STREAM DI BYTE - INPUT - CLASSI SORGENTI

Rispetto alla classe-base (astratta) `InputStream` , ci sono due ***classi sorgente*** notevoli derivate da essa:

- `ByteArrayInputStream` ne implementa i metodi nel caso particolare in cui l'input è un buffer (array) di byte, passato all'atto della costruzione del `ByteArrayInputStream`;
- `FileInputStream` ne implementa i metodi nel caso particolare in cui l'input è un file, il cui nome è passato al costruttore di `FileInputStream`; in alternativa si può passare al costruttore un oggetto `File` (o anche un `FileDescriptor`).

Esempio d'uso di FileInputStream:

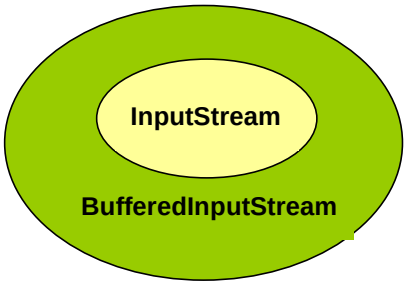
```
FileInputStream f = null;
f = new FileInputStream("Prova.dat");
```

Oppure si può fare anche:

```
File ff = new File("pippo.dat");
f = new FileInputStream(f);
```

STREAM DI BYTE - INPUT - CLASSI SORGENTI

Le altre classi di filtraggio derivano direttamente o indirettamente da `InputStream` e provvedono a incapsulare un `InputStream` in un tipo di stream più evoluto, onde aggiungere nuove funzionalità.



Due classi derivano direttamente da InputStream::

- `ObjectInputStream` serve per leggere oggetti serializzati salvati su stream e offre anche metodi per i tipi primitivi e le classi standard (`Integer`, etc.) di Java.
- un `FilterInputStream` modifica il metodo `read()` (ed eventualmente gli altri) in accordo al criterio di filtraggio richiesto (per default, il filtro è trasparente e non filtra niente). *In pratica, quindi, si usano sue sottoclassi notevoli di FilterOutputStream*:
  - `BufferedInputStream` modifica il metodo read() in modo tale da avere un input bufferizzato tramite un buffer che aggiunge egli stesso.
  - `DataInputStream` definisce metodi per leggere i tipi primitivi di Java scritti su ub file binario, come da esempio per esempio `readInteger`(), `readFloat`().

IL CASO DI System.In

In Java, il dispositivo di input standard è la variabile (`static`) `System.in`, di classe `InputStream`.

InputStream è una classe astratta, e `System.in` è risulta esserne una istanza "anomala" predefinita.

Poiché `InputStream` fornisce solo un metodo `read()` che legge singoli byte, si usa incapsulare `System.in` in un oggetto dotato di maggiori funzionalità, come ad esempio un `BufferedReader`, che fornisce anche un metodo `readLine()`:

```
import java.io.*;
class EsempioIn {
  public static void main(String args[]){
    int a = 0, b = 0;
    BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));
    try {
      System.out.print("Primo valore: ");
      a = Integer.parseInt(in.readLine());
      System.out.print("Secondo valore:");
      b = Integer.parseInt(in.readLine());
    }
    catch (IOException e) {
      System.out.println("Errore in input");
    }
    System.out.println("La somma vale " + (a+b));
  }
}
```

STREAM DI BYTE - OUTPUT

Rispetto alla classe-base (astratta) `OutputStream` , ci sono due **classi sorgente** notevoli:

- `ByteArrayOutputStream` implementa questi metodi nel caso particolare in cui l'output è un buffer (array) di byte *interno*, dinamicamente espandibile, recuperabile con`toByteArray()` o `toString()`, secondo i casi.
- `FileOutputStream` implementa questi metodi nel caso particolare in cui l'output è un file, il cui nome è passato al costruttore di `FileOutputStream`; in alternativa si può passare al costruttore un oggetto `File` (o anche un `FileDescriptor`)

Per quanto riguarda le **classi di filtraggio**, sono notevoli le seguenti:

- `ObjectOutputStream` serve per scrivere oggetti serializzati su stream: in particolare offre metodi per scrivere i tipi primitivi e le classi standard (`Integer`, etc.) di Java.
- un `FilterOutputStream` modifica il metodo `write()` (ed eventualmente gli altri) in accordo al criterio di filtraggio richiesto (per default, il filtro è trasparente): *in pratica si usano sue sottoclassi notevoli*, come `BufferedOutputStream` o `DataOutputStream`.
- `PrintStream` definisce metodi per stampare sotto forma di `String` i tipi primitivi e le classi standard (`Integer`, etc.) di Java (medinte un metodo `toString()`)
- `DataOutputStream` definisce metodi per scrivere in forma binaria i tipi primitivi e le classi standard di Java (`writeInteger(), writeFloat(),` etc.)
- `BufferedOutputStream` modifica il metodo `write()` in modo dfa scrivere tramite un buffer (che aggiunge egli stesso).

ESEMPIO 1 - Scrittura e lettura da file binario

Per scrivere su un file binario occorre un `FileOutputStream`, che però consente solo di scrivere un *byte* o un *array di byte*.

Volendo scrivere dei `float`, `int`, `double`, `boolean`, … è molto più pratico un `DataOutputStream`, che fornisce metodi idonei.

Si incapsula il `FileOutputStream` in un `DataOutputStream`.

```
import java.io.*;
public class Es1 {
  public static void main(String args[]){
    FileOutputStream f = null;
    try {
        f = new FileOutputStream("Prova.dat");
    }
    catch(IOException e){
      System.out.println("Apertura file fallita");
      System.exit(1);
    }
    DataOutputStream os = new DataOutputStream(f);
    System.out.print("Scrittura su file...");
    float f1 = 3.1415F;   char c1 = 'E';
    boolean b1 = true;    double d1 = 1.4142;
    try {
      os.writeFloat(f1);
      os.writeBoolean(b1);
      os.writeDouble(d1);
      os.writeChar(c1);
      os.writeInt(12);
      os.close();
    }
    catch (IOException e){
      System.out.println("Scrittura fallita");
      System.exit(2);
    }
    System.out.println(" done.");
```

ESEMPIO 1 (segue)

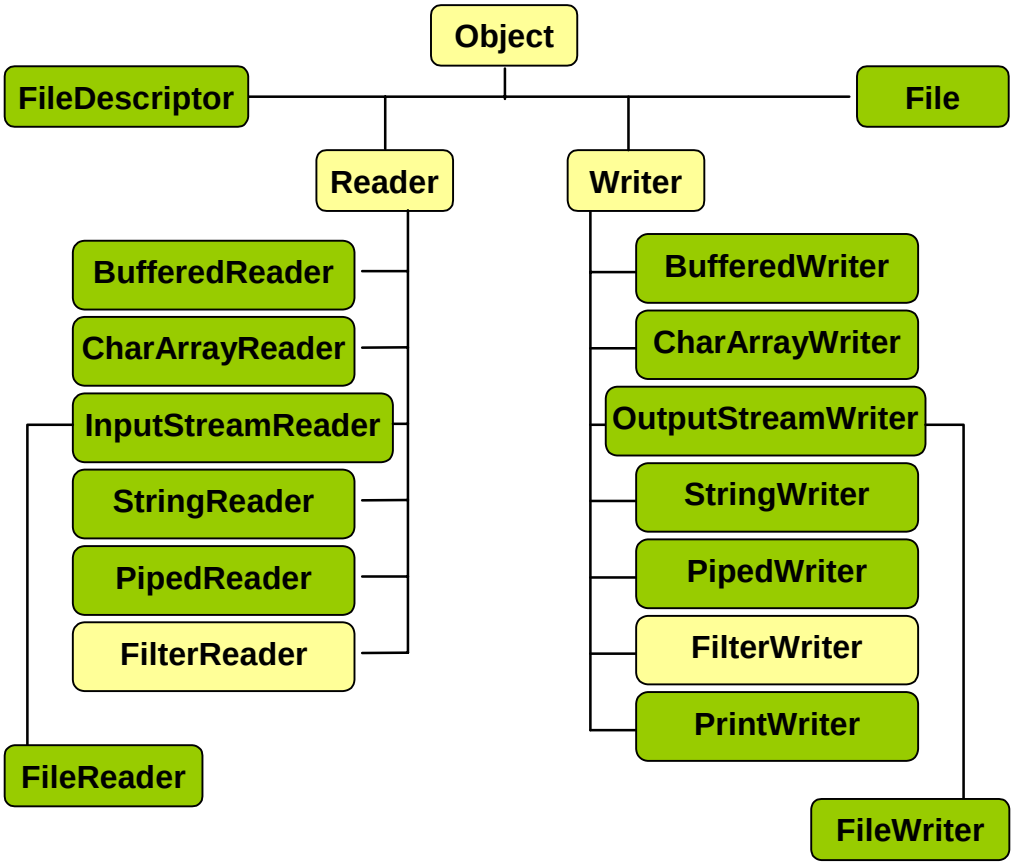
Analogamente, per leggere da un file binario occorre un `FileInputStream`, che incapsuliamo in un `DataInputStream` per poter leggere i tipi primitivi di Java in modo semplice:

```
    System.out.println("Rilettura da file:");
    FileInputStream  fin = null;
    try {
        fin = new FileInputStream("Prova.dat");
    }
    catch(FileNotFoundException e){
      System.out.println("File non trovato");
      System.exit(3);
    }
    DataInputStream is = new DataInputStream(fin);
    try {
      float f2;      char  c2;
      boolean b2;    double  d2;
      int i2;
      f2 = is.readFloat();
      b2 = is.readBoolean();
      d2 = is.readDouble();
      c2 = is.readChar();
      i2 = is.readInt();
      is.close();
      System.out.println(f2 + ", " + b2 + ", "
                  + d2 + ", " + c2 + ", " + i2);
    }
    catch (IOException e){
      System.out.println("Errore di input");
      System.exit(4);
    }
  }

}
```

LA GERARCHIA DEGLI STREAM DI CARATTERI

Le classi per l'I/O da stream di caratteri sono più efficienti di quelle per l'I/O generico a byte, e gestiscono la *conversione fra la rappresentazione UNICODE adottata da Java e quella specifica della piattaforma in uso (tipicamente ASCII)* e della lingua adottata (cosa essenziale per il supporto dell'internazionalizzazione).



Anche qui vale il principio dell'"incapsulamento" di classi pozzo in classi di filtraggio di un `Reader` o di un `Writer` da parte di classi più evolute.

I metodi forniti sono del tutto analoghi a quelli degli stream.

STREAM DI CARATTERI - INPUT

Rispetto alla classe-base (astratta) `Reader`, una classe concreta deve implementare `read()` e `close()`, anche se può ovviamente ridefinire altri metodi. `read()` restituisce un carattere UNICODE.

Classi sorgenti notevoli:

- `CharArrayReader` è il corrispondente di `ByteArrayInput-Stream` nel caso di `Reader`: prende l'input da array di caratteri.
- `StringReader` è analogo al precedente, ma usa come sorgente un oggetto `String` invece di un array di caratteri.
- `InputStreamReader` **è un caso particolarissimo di reader!** Reinterpreta un `InputStream` (stream di byte) come stream di caratteri, traslando l'input da byte a caratteri UNICODE secondo la lingua locale prescelta. L'uso tipico di questa classe è con lo stream `System.In`. (VEDI PAGINE SEGUENTI!)
  - La sua sottoclasse `FileReader` crea tale `InputStream` a partire da un nome di file o da un oggetto `File` o da un `FileDescriptor`. In pratica, costruisce un `FileIntputStream` e lo incapsula in un `InputStreamReader`.

Classi di filtraggio notevoli:

- `BufferedReader` aggiunge il buffering a un `Reader` di altro tipo (solitamente un `FileReader` o un `InputStreamReader`), definendo anche il metodo `readLine()`, che legge una riga fino al fine linea (platform-dependent)
- `FilterReader`, a differenza degli stream è qui una classe astratta, sebbene non abbia metodi astratti: istanziando una sottoclasse si ha un filtro "trasparente", a meno che non si ridefiniscano opportunamente i due metodi `read()`.

Non ci sono classi di filtraggio che "*interpretano*" i caratteri letti (p.e., non c'è un `DataInputReader`).

STREAM DI CARATTERI - OUTPUT

Rispetto alla classe-base (astratta) `Writer`, una classe concreta deve implementare solo `write()`, `flush()` e `close()`, anche se può ovviamente ridefinire altri metodi.

`write()` restituisce un carattere UNICODE.

Classi sorgenti notevoli:

- `CharArrayOutputStream` è il corrispondente di `ByteArrayOutputStream` nel caso di `Writer`.
- `OutputStreamWriter` è **un caso particolarissimo di writer** e reinterpreta un `OutputStream` (stream di byte) come stream di caratteri, traslando l'output da caratteri UNICODE a byte secondo la lingua locale prescelta. L'uso tipico è con `System.Out`.
  - La sua sottoclasse `FileWriter` crea tale `OutputStream` a partire da un nome di file o da un oggetto `File` o da un `FileDescriptor`. In pratica, costruisce un FileOutputStream e lo incapsula in un OutputStreamReader.

Classi di Filtraggio Notevoli:

- `BufferedWriter` aggiunge il buffering a un `Writer` di altro tipo (solitamente un `FileWriter` o un `OutputStreamWriter`), definendo anche il metodo `newLine()` per emettere il fine linea (platform-dependent)
- `FilterWriter` a differenza degli stream, è qui una classe astratta, sebbene non abbia metodi astratti: istanziando una sottoclasse si ha un filtro "trasparente", a meno che non si ridefiniscano opportunamente i due metodi `write()`.

Non ci sono classi di filtraggio che "*interpretano*" i dati da scrivere (p.e., non c'è un `DataOutputReader`).

IL CASO DI System.In e System.Out

Lo standard input (`System.in`) e lo standard output (`System.out`) sono in effetti *stream di caratteri*

- Dovrebbero essere definiti come `Reader` e `Writer`, rispettivamente
- Invece, sono definiti come `InputStream` e `OutputStream` *per motivi di compatibilità* con le versioni precedenti di Java, perché in Java 1.0 Reader e Writer non c'erano e c'erano solo gli stream!

Quindi, bisogna fare attenzione:

- anche se definiti come InputStream e OutputStream *per motivi di compatibilità*, rimangono *stream di caratteri*
- Non ha senso scrivere su essi *dati in forma binaria*
  - in output, si vedrebbero quadratini e faccine
  - in input, si leggerebbero valori casuali
- Si possono "trasformare" in Reader e Writer con alcune classe apposite: `InputStreamReader e OutputStreamReader`.

- **A tutti gli effetti, le classi** `InputStreamReader` e `OutputStreamReader` sono state definite solo per motivi di compatibilità con le versioni precedenti di Java.

ESEMPIO 2 - Scrittura e lettura da file di testo

Per scrivere su un file di testo occorre un `FileWriter`, il cui metodo `write()` consente di scrivere o un singolo carattere, o una stringa.
Volendo scrivere su un file di testo `float`, `int`, ... occorre *convertirli in stringhe a priori*, usando il metodo `toString()` definito nella classe wrapper Java corrispondente. Non c'è qualcosa di corrispondente al `DataOutputStream`.

```
import java.io.*;
public class Es2 {
  public static void main(String args[]){
    FileWriter fout = null;
    try { fout = new FileWriter("Prova.txt"); }
    catch(IOException e){
      System.out.println("Apertura file fallita");
      System.exit(1);
    }
    float f1 = 3.1415F;   char c1 = 'E';
    boolean b1 = true;    double d1 = 1.4142;
    try {
      String buffer = null;
      buffer = Float.toString(f1);
      fout.write(buffer,0,buffer.length());
      buffer = new Boolean(b1).toString(); // *
      fout.write(buffer,0,buffer.length());
      buffer = Double.toString(d1);
      fout.write(buffer,0,buffer.length());
      fout.write(c1); // singolo carattere
      buffer = Integer.toString(12);
      fout.write(buffer,0,buffer.length());
      fout.close();
    }
    catch (IOException e){
      System.out.println("Errore di scrittura");
      System.exit(2);  }
```

ESEMPIO 2 (segue)

Analogamente, per leggere da un file di testo occorre un `File-Reader`, il cui metodo `read()` consente di leggere o un singolo carattere, o un array di caratteri.

Per leggere fino alla fine del file, basta fare un ciclo controllato dal metodo `ready()`, che restituisce true finché ci sono altri caratteri da leggere sullo stream:

```
    System.out.println("Rilettura da file:");
    FileReader fin = null;
    try {
        fin = new FileReader("Prova.txt");
    }
    catch(FileNotFoundException e){
      System.out.println("File non trovato");
      System.exit(3);
    }
    // ----- ciclo di lettura ----
    try {
      while(fin.ready()){
        char ch = (char)fin.read();
        System.out.print(ch);
      }
      System.out.println("");
    }
    catch (IOException e){
      System.out.println("Errore di input");
      System.exit(4);
    }
  } // NB: stampa 3.1415true1.4142E12
}
```

NB: `read()` restituisce il carattere letto come `int`, in modo da poter indicare con un valore negativo (-1) la fine dello stream. L'intero va quindi convertito esplicitamente in `char` con un cast.

ESEMPIO 3 - Una variante

Nell'esercizio precedente, i valori sono scritti sul file uno di seguito all'altro, e vengono quindi riletti nello stesso modo:

```
3.1415true1.4142E12
```

Questo è decisamente poco pratico nelle applicazioni.
Sarebbe meglio che i vari campi fossero separati fra loro almeno da uno spazio: ciò consentirebbe forme più sofisticate di lettura.

Perciò in questo esempio:

- in fase di scrittura, si aggiunge uno spazio fra un elemento e l'altro
- in fase di lettura, si definisce un apposito metodo `readField()` che legge tutti i caratteri fino al primo spazio, restituendo una stringa.

```
import java.io.*;
public class Es3 {
  public static void main(String args[]){
    FileWriter fout = null;
    …
    try {
      String buffer = null;
      buffer = Float.toString(f1)+ " ";
      fout.write(buffer,0,buffer.length());
      buffer = new Boolean(b1).toString() + " ";
      fout.write(buffer,0,buffer.length());
       …
      fout.write(c1); // singolo carattere
      fout.write(' '); // AGGIUNGO LO SPAZIO
      buffer = Integer.toString(12);
      fout.write(buffer,0,buffer.length());
      fout.close();
    }
    …
```

ESEMPIO 3 - Una variante (segue)

Ora i valori sono scritti sul file separati da uno spazio:

```
3.1415 true 1.4142 E 12
```

Il metodo di classe `readField()` può quindi facilmente leggere tutti i caratteri fino al primo spazio, restituendo infine una stringa:

```
static public String readField(Reader in){
   StringBuffer buf = new StringBuffer();
   boolean nospace = true;
   try {
     while(in.ready() && nospace){
       char ch =(char)in.read();
       nospace = (ch!=' ');
       if (nospace) buf.append(ch);
     }
   }
   catch (IOException e){
     System.out.println("Errore di input");
     System.exit(4);
   }
   return buf.toString();
}
```

Il ciclo di lettura risulta quindi riformulato come segue:

```
   FileReader fin = null;
   …
   try {
     while(fin.ready()){
       String s = readField(fin);
       System.out.println(s);
     }
   }
```

Questo pone le premesse per poter interpretare le stringhe lette come valori `int`, `float`, `double`, etc.

ESEMPIO 4 - Lettura <u>di valori complessi</u> da file di testo

Poiché nessuna delle classi `Reader` standard offre la capacità di leggere direttamente da un file di testo valori `int`, `float`, ..., *decidiamo di* ***definire una nuova classe DataReader*** che la fornisca..

Il punto è: **<u>dove collocare questa classe nella gerarchia?</u>**

<u>Riflessioni</u>:

1. se facciamo `DataReader` come sottoclasse di `FileReader`, si può usarla solo con i file → troppo limitativo
2. fare `DataReader` come sottoclasse di `InputStreamReader` non è praticamente possibile, perché il costruttore di `InputStreamReader` richiede un `InputStream` (non un `Reader`!) → poi non si riuscirebbe a usarla con i `Reader`, e in particolare con i `FileReader` → inutile
3. volendo creare una classe "wrapper" che manipoli i file, occorre che il costruttore di `DataReader` accetti come argomento un `FileReader`, o, *meglio*, *un `InputStreamReader`*, che è più generale

Non resta quindi che

far derivare `DataReader` direttamente da `Reader`.

e

costruire `DataReader` a partire da un `InputStreamReader`

```
class DataReader extends Reader {
    ...
  public DataReader(InputStreamReader in) {...}
    ...
}
```

ESEMPIO 4 (II) - La classe `DataReader` (1)

Problema: `Reader` è una classe astratta
→ occorre implementare i due metodi astratti

```
public void close()  throws IOException;
public int read(char buf[], int off, int len)
                      throws IOException;
```

Come implementarli ?

Riflessione:

Poiché `DataReader` è costruita a partire da un `Input-StreamReader`, *dovrà contenere al suo interno un riferimento a un oggetto `InputStreamReader`*

↓

possiamo **delegare le due operazioni a tale oggetto**:

```
class DataReader extends Reader {
  InputStreamReader in;
  public DataReader(InputStreamReader in) {
        this.in=in;
  }
  public void close() throws IOException {
       in.close();   // DELEGAZIONE
  }
  public int read(char b[], int off, int len)
    throws IOException {
       return in.read(b,off,len); // DELEGAZIONE
  }
...
}
```

A questo punto, la classe `DataReader` può essere facilmente completata come segue:

ESEMPIO 4 (III) - La classe `DataReader` (2)

Il metodo `readString()` non è altri che il `readField()` dell'esercizio precedente, con alcune modifiche:

- verifica dell'end-of-file controllando se `read()` restituisce -1 *anziché chiamando `ready()`*, per gestire correttamente anche il caso di input da console (da console, `ready()` dà sempre falso)
- ampliamento dei caratteri considerati "di separazione" (in particolare, `'\r'` corrisponde alla pressione del tasto INVIO)

```
class DataReader extends Reader {
  InputStreamReader in;
    ...
  public String readString() throws IOException {
    StringBuffer buf = new StringBuffer();
    boolean go = true;
    do {
        int x = in.read(); // DELEGAZIONE
        if (x!=-1) { // end-of-file not reached
            char ch = (char)x;
            go = (ch!=' ' && ch!='\n' &&
                  ch!='\t' && ch!='\r');
            if (go) buf.append(ch);
        } else go=false;
    } while(go);
    return buf.toString();
  }
    ...
```

A partire da `readString()` è poi molto semplice implementare gli altri metodi:

ESEMPIO 4 (IV) - La classe `DataReader` (3)

Gli altri metodi si creano a partire da `readString()` sfruttando i metodi `ParseInt`, `ParseFloat`, etc delle rispettive classi:

```
class DataReader extends Reader {
 InputStreamReader in;
    ...
 public int readInt() throws IOException {
    return Integer.parseInt(readString());
 }
 public float readFloat() throws IOException {
    return Float.parseFloat(readString());
 }
 public double readDouble() throws IOException {
    return Double.parseDouble(readString());
 }
 public boolean readBoolean() throws IOException {
    return readString().equals("true");
 }
 public char readChar() throws IOException {
    return readString().charAt(0);
 }
}
```

ESEMPIO 4 (V) - Il main

Il main a questo punto assume la forma seguente:

```
import java.io.*;
public class Es4 {
  public static void main(String args[]){
    FileReader fin = null;
    …
    DataReader dr = new DataReader(fin);
   try {
      float f2 = dr.readFloat();
      boolean b2 = dr.readBoolean();
      double d2 = dr.readDouble();
      char c2 = dr.readChar();
      int i2 = dr.readInt();
      dr.close();
      System.out.println(f2 + ", " + b2 + ", "
                  + d2 + ", " + c2 + ", " + i2);
    }
    catch (IOException e){
      System.out.println("Errore di input");
      System.exit(4);
    }

    // ------- da console -------
    DataReader consoleInput = new DataReader(
                new InputStreamReader(System.in));
    System.out.println("Inserire nell'ordine un”
        + " float, un boolean, un double, un char”
        + " e un int, separati da UNO spazio: ");
    try {
      float f2 = consoleInput.readFloat();
      ...
    }
    catch (IOException e){...}
  }
}
```

ESEMPIO 5 - Uso di uno `StreamTokenizer`

Il package `java.io` fornisce una classe `StreamTokenizer` che consente di estrarre parole o numerio ("token") da uno stream di testo, in modo configurabile. In particolare:

- il metodo `whitespaceChars(int lo, int hi)` registra tutti i caratteri da `lo` a `hi` come separatori
- il metodo `nextToken()` estrae dallo stream il "token" successivo: il risultato è un intero che rappresenta *il tipo* del token letto (`TT_WORD` → `String`, `TT_NUMBER` → `double`)
- il token letto è disponibile o nel campo pubblico `sval` o nel campo pubblico `nval`, a seconda se è un numero o una stringa.

```
import java.io.*;
public class Es5 {
  public static void main(String args[]){
    FileReader f = null;
    ...
    StreamTokenizer t = new StreamTokenizer(f);
    tkinput.whitespaceChars(0, (int)' ');
    int res = -1;
    do {
      try {  res = tkinput.nextToken(); }
      catch (IOException e){...}
      if (res==StreamTokenizer.TT_WORD) {
        String s = new String(tkinput.sval);
        System.out.println("stringa: " + s);
      } else
      if (res==StreamTokenizer.TT_NUMBER) {
        double d = tkinput.nval;
        System.out.println("double: " + d);
      }
    } while(res!=StreamTokenizer.TT_EOL &&
            res!=StreamTokenizer.TT_EOF);
  }}
```

SERIALIZZAZIONE DI OGGETTI

Serializzare un oggetto significa trasformarlo in uno stream di byte, salvabile su file; analogamente, **deserializzare** un oggetto significa ricostruirlo a partire dalla sua rappresentazione su file.

Una classe serializzabile deve implementare l'interfaccia `Serializable` (è vuota, funge da marcatore).

- `ObjectOutputStream` è la sottoclasse di `OutputStream` usata per serializzare un oggetto
- `ObjectInputStream` è la sottoclasse di `InputStream` usata per deserializzare un oggetto

Queste due classi funzionano in larga misura come `DataInput-Stream` e `DataOutputStream`, con una differenza: **aggiungono la capacità di scrivere/leggere *oggetti non primitivi* su/da uno stram di byte.**

Un oggetto viene

- serializzato invocando il metodo `writeObject()` di `ObjectOutputStream`;
- deserializzato invocando il metodo `readObject()` di `ObjectInputStream`.

Questi metodi scrivono / leggono *tutti* i dati, inclusi i campi privati e quelli ereditati dalla superclasse.

I campi dati che sono riferimenti a oggetti sono serializzati invocando ricorsivamente `writeObject()` sull'oggetto referenziato: serializzare un oggetto può quindi comportare la serializzazione di *un intero grafo* di oggetti. L'opposto accade quando si deserializza un oggetto.

NB: se uno stesso oggetto è referenziato più volte nel grafo, viene serializzato una sola volta, onde evitare che `writeObject()` cada in una ricorsione infinita.

ESEMPIO 1 - Serializzazione e deserializzazione di un oggetto

Una classe Lista di Oggetti…

```
class List implements Serializable {
    Object e;  List next;
    public List() {e=null; next=null; }
    public Object head() { return e; }
    public List tail() { return next; }
    public List cons(Object e) {
        List nl = new List();
        nl.e = e; nl.next = this; return nl;
    }
}
```

… e un main che la crea , la salva …

```
public class Prova1 {
  static List makeList() {
    List l = new List();
    l = l.cons(new Integer(5));
    l = l.cons(new String("ciao"));
    return l;
  }
  public static void main(String args[]){
    FileOutputStream f = null;
    try { f = new FileOutputStream("Prova.dat"); }
    catch(IOException e){ System.exit(1); }
    ObjectOutputStream os = null;
    List l = makeList();
    System.out.print("Scrittura su file...");
    try {  os = new ObjectOutputStream(f);
           os.writeObject(l); os.flush();
           os.close();
    }
    catch (IOException e){ System.exit(2); }
    System.out.println(" ok.");
    // segue...
```

ESEMPIO 1 (segue)

… e la ricostruisce:

```
    System.out.println("Rilettura da file:");
    FileInputStream  fin = null;
    ObjectInputStream is = null;
    try {
        fin = new FileInputStream("Prova.dat");
        is = new ObjectInputStream(fin);
    }
    catch(IOException e){ System.exit(3); }

    List x=null;
    try { x = (List)(is.readObject());
          is.close();
    }
    catch (IOException e){ System.exit(4); }
    catch (ClassNotFoundException e){
        System.exit(5);
    }

    List xx=x;
    while (xx!=null){
      System.out.println(xx.head());
      xx = xx.tail();
    }
  }
}
```

CAMPI TRANSIENTI

A volte, **non tutti i campi dati di un oggetto devono essere serializzati.**

Alcuni dati, infatti, possono non essere significativi (ad esempio, la posizione di un cursore grafico) in quanto devono essere ricalcolati o resettati all'atto del ricaricamento dell'oggetto (ad esempio, il cursore può dover essere riposizionato al centro dell'area).

È possibile distinguere i campi da non serializzare etichettandoli `transient`.

PERSONALIZZAZIONE della serializzazione

A volte, può essere necessario *personalizzare* il modo in cui una classe viene serializzata e deserializzata.
Per fare questo, occorre *definire nella classe i due metodi (privati!)* `writeObject()` *e* `readObject()`.

GESTIONE DELE VERSIONI

La versione serializzata di un oggetto contiene **un numero di versione,** che è importante per verificare se la classe è in grado di deserializzare l'oggetto.
Infatti, una nuova versione di una classe potrebbe non essere in grado di deserializzare un oggetto di una versione precedente della stessa classe.