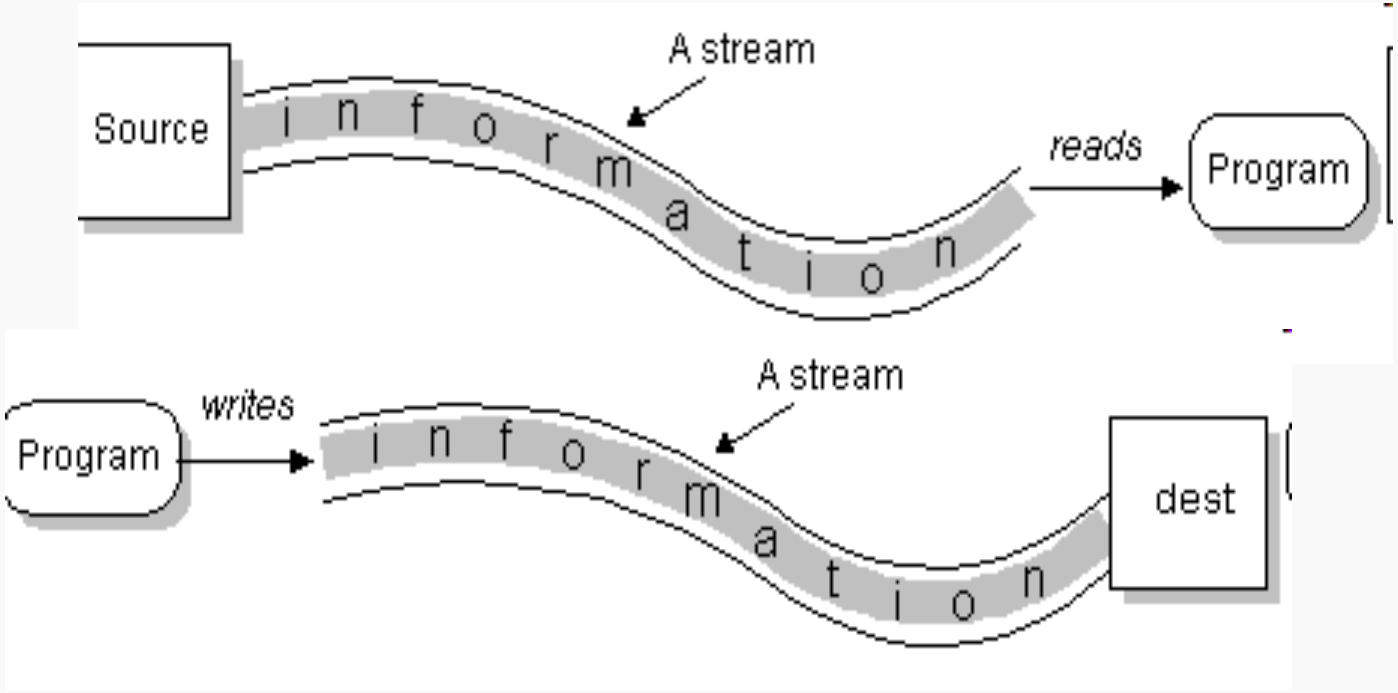

# Il linguaggio Java

## *Gli stream*

# Overview




Uno *stream* è un flusso *unidirezionale* di informazioni da una sorgente *esterna* ovvero verso una sorgente esterna a cui si accede in modo *sequenziale*

# Algoritmi di lettura/scrittura

Reading

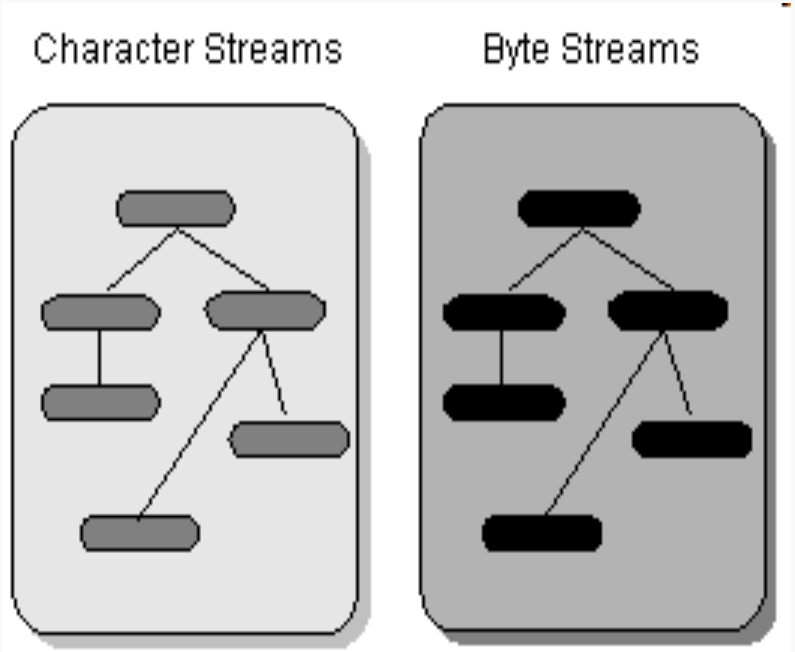
```
open a stream
while more information
    read information
close the stream
```

Writing

```
open a stream
while more information
    write information
close the stream
```

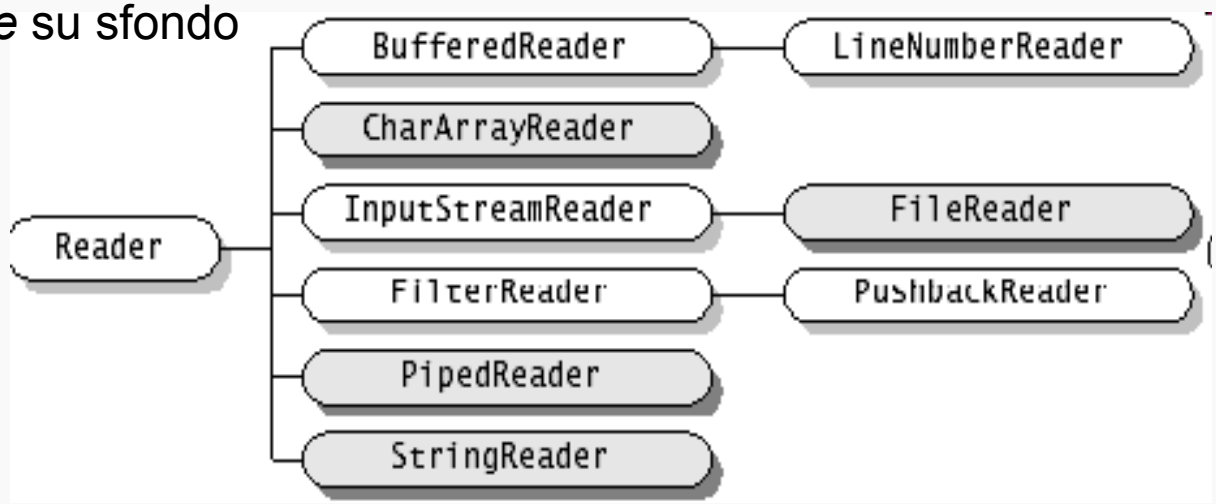
- Gli algoritmi di lettura e di scrittura sono *indipendenti* dalla sorgente/destinatario esterno e dalle informazioni lette/scritte

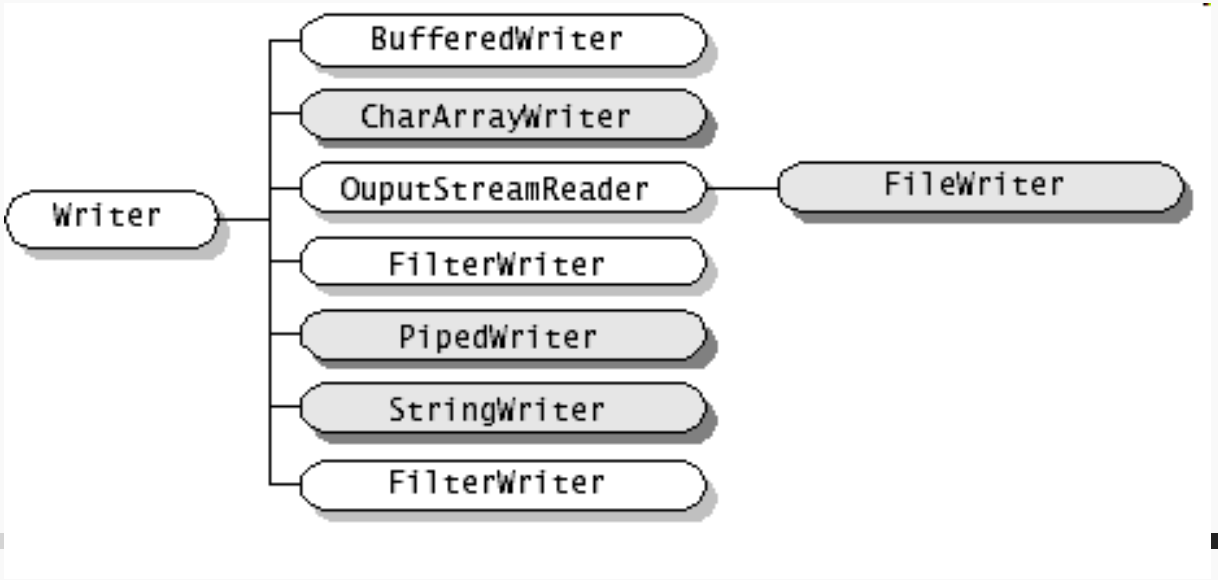
# Il package java.io



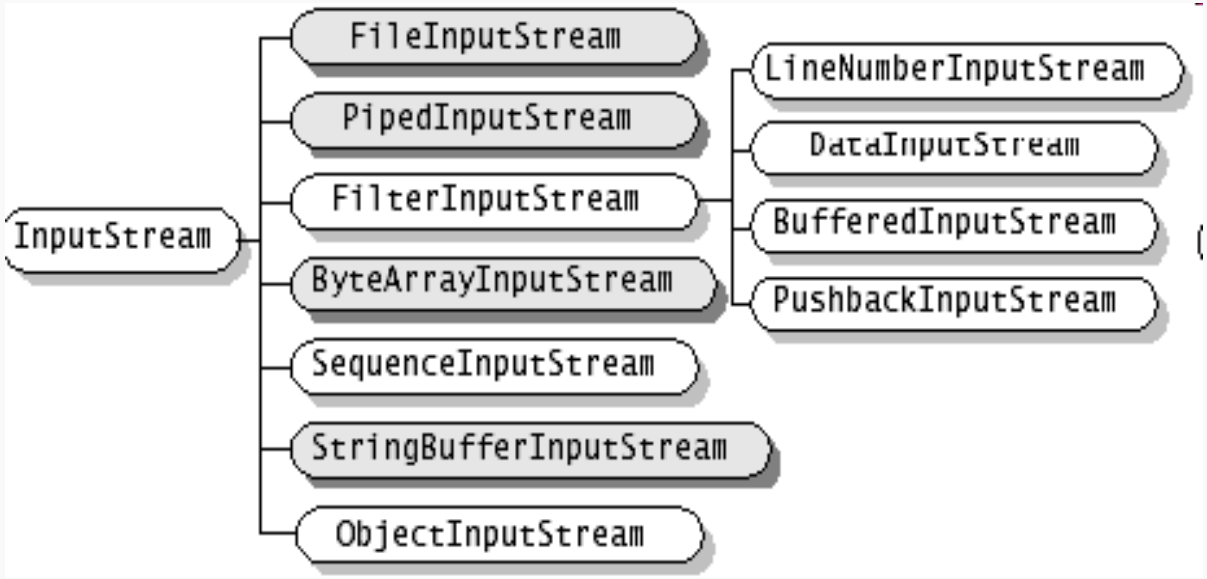
# Flussi Di Caratteri

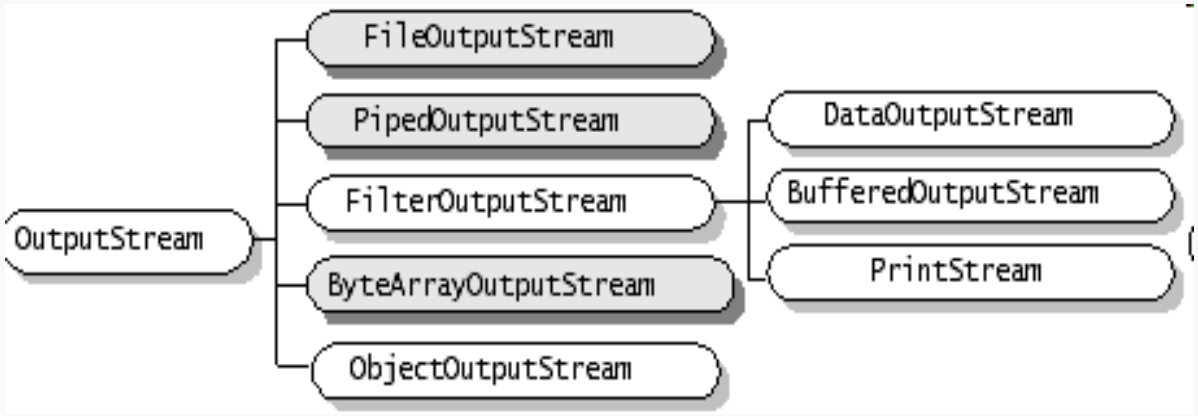
*Classi astratte* su sfondo bianco



# Flussi Di Byte



# API

## Reader

```
int read()
int read(char cbuf[])
int read(char cbuf[],
         int offset,
         int length)
```

## InputStream

```
int read()
int read(byte cbuf[])
int read(byte cbuf[],
         int offset,
         int length)
```

## Writer

```
int write(int c)
int write(char cbuf[])
int write(char cbuf[],
          int offset,
          int length)
```

## OutputStream

```
int write(int c)
int write(byte cbuf[])
int write(byte cbuf[],
          int offset,
          int length)
```

# Apertura e chiusura dei flussi

- Quando un flusso viene creato viene anche automaticamente aperto
- Un flusso può essere chiuso esplicitamente per mezzo del metodo **close**
- Quando un flusso viene raccolto, il GC provvede *automaticamente* anche a chiuderlo

# I flussi file

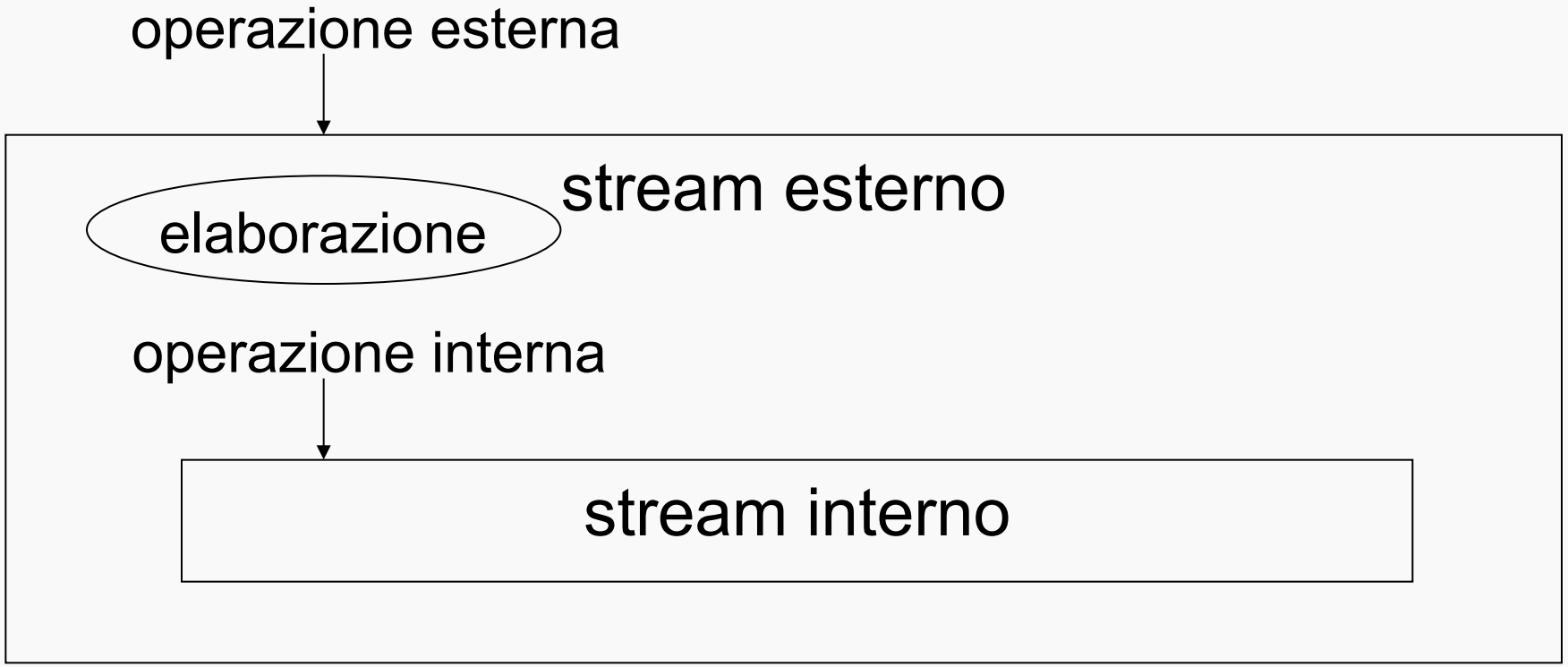
Flusso file di caratteri

- **FileReader**
- **FileWriter**

Flusso file di byte

- **FileInputStream**
- **FileOutputStream**

# Wrapping




lo stream esterno (outer stream) *avvolge* lo stream interno (inner stream)

# Lettura e scrittura bufferizzata

- **public class BufferedReader** legge da uno stream a caratteri, bufferizzando i caratteri in modo da fornire una lettura efficiente di caratteri, linee ed array
  - La dimensione del buffer può essere specificata;
  - la dimensione di default è sufficientemente grande per la maggior parte degli scopi;
- **public class BufferedWriter** scrive in uno stream a caratteri, bufferizzando i caratteri in modo da fornire una scrittura efficiente di caratteri, linee ed array

# Scrittura formattata

- **public class PrintWriter** permette di scrivere una *rappresentazione formattata* di un qualunque oggetto su di un flusso di caratteri
  - **void print(boolean b), void print(int i), void print(long l), …**
  - **void print(String s)**
  - **void print(Object o)**
  - **…**
- Se l'**auto-flush** è abilitato, esso verrà eseguito quando viene eseguito uno dei metodi **println**, **printf**, o **format**

# I flussi filtro

- I flussi filtro eseguono un *filtraggio* (*elaborazione*) sui dati letti dal, o da scrivere sul, flusso interno che avvolgono
- Il tipo di filtraggio dipende dal tipo del flusso filtro
  - Es.: bufferizzazione dei dati; conteggio dei dati in transito; conversione dei dati; …
- I flussi filtro discendono da
  - flussi di byte: **FilterInputStream**, **FilterOutputStream**
  - flussi di char: **FilterReader**, **FilterWriter**
- Per i flussi di byte
  - **public class DataInputStream** e **public class DataOutputStream** permettono di leggere e scrivere, rispettivamente, valori di tipo primitivo sul flusso di byte avvolto

# I flussi filtro (byte)

- Per ***utilizzare*** un flusso filtro bisogna avvolgere un flusso esistente in un flusso filtro (quando questo viene creato, ad esempio)
- Per ***definire*** un flusso filtro, bisogna
  - ***derivare*** **FilterInputStream**(*) o **FilterOutputStream**(*)
  - ***sovrascrivere*** i metodi **read** e **write**

(*) Classi astratte.


UNIVERSITÀ DI PISA

# *La serializzazione*

## *concetti base*

# Serializzazione

- La *serializzazione* è il processo per mezzo del quale lo stato di un oggetto viene scritto in un flusso (*serializzato*) in modo tale da poter essere successivamente riletto
- La serializzazione viene utilizzata per
  - Network programming
  - Remote Method Invocation (RMI)
  - Lightweight persistence

# Serializzazione

- Per la serializzazione di oggetti si utilizzano le classi
  **ObjectInputStream**
  **ObjectOutputStream**
- Tali oggetti devono avvolgere oggetti di classe
  **InputStream**
  **OutputStream**
  rispettivamente

# Serializzazione di oggetti

*Esempio*

```
FileOutputStream out = new FileOutputStream("theTime");
ObjectOutputStream s = new ObjectOutputStream(out);
s.writeObject("Today");
s.writeObject(new Date());
s.flush();
```

# Il metodo writeObject

- **public final void writeObject (Object obj) throws IOException**
  - serializza l'oggetto,
  - attraversa i suoi riferimenti ad altri oggetti e,
  - ricorsivamente, li serializza
- Il metodo **writeObject** lancia l'eccezione **NotSerializableException**(*) se l'oggetto non è serializzabile
- Un oggetto è serializzabile se implementa l'interfaccia **Serializable**

(*) sottoclasse di **IOEXception**

# Lettura di oggetti serializzati

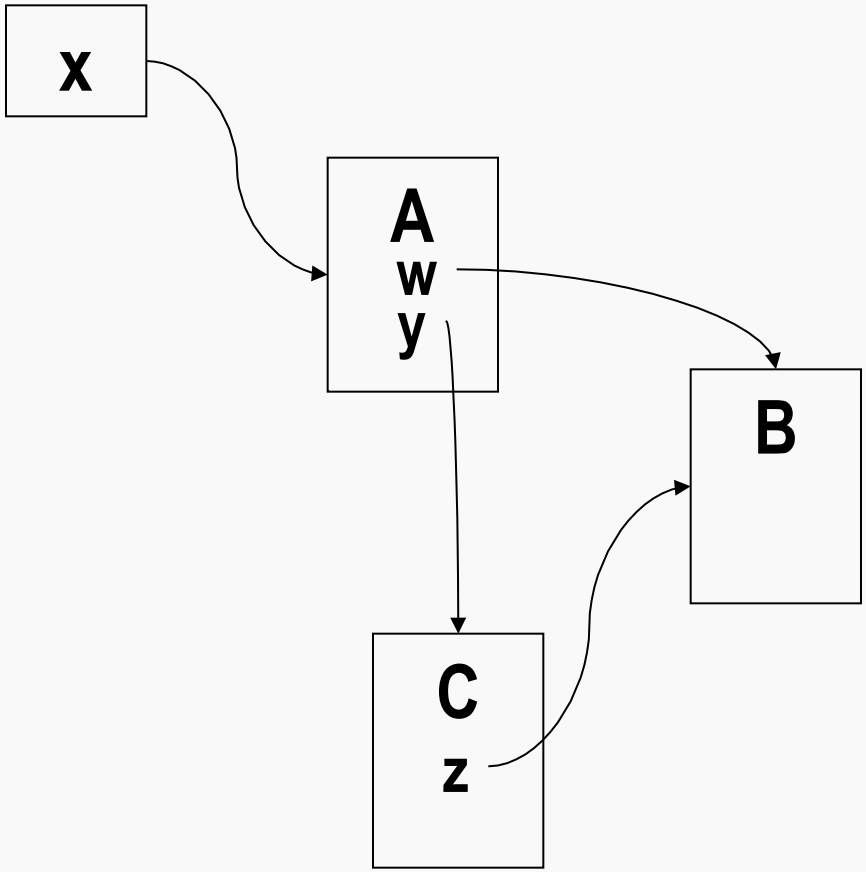
```
FileInputStream in = new FileInputStream("theTime");

ObjectInputStream s = new ObjectInputStream(in);

String today = (String)s.readObject();

Date date = (Date)s.readObject();
```

# Il metodo readObject

- **public final Object readObject() throws**
  **IOException,**
  **ClassNotFoundException**

  ritorna un **Object** che deve essere opportunamente "castato"
- Il metodo **readObject**
  - de-serializza l'oggetto,
  - attraversa i suoi riferimenti ad altri oggetti e,
  - ricorsivamente, li de-serializza
- Un oggetto è de-serializzabile se implementa l'interfaccia **Serializable**

# Serializzazione di una rete di oggetti




- **Efficienza**: ogni oggetto deve essere copiato sullo stream una sola volta
- **Consistenza**: devono essere mantenute le relazioni tra gli oggetti

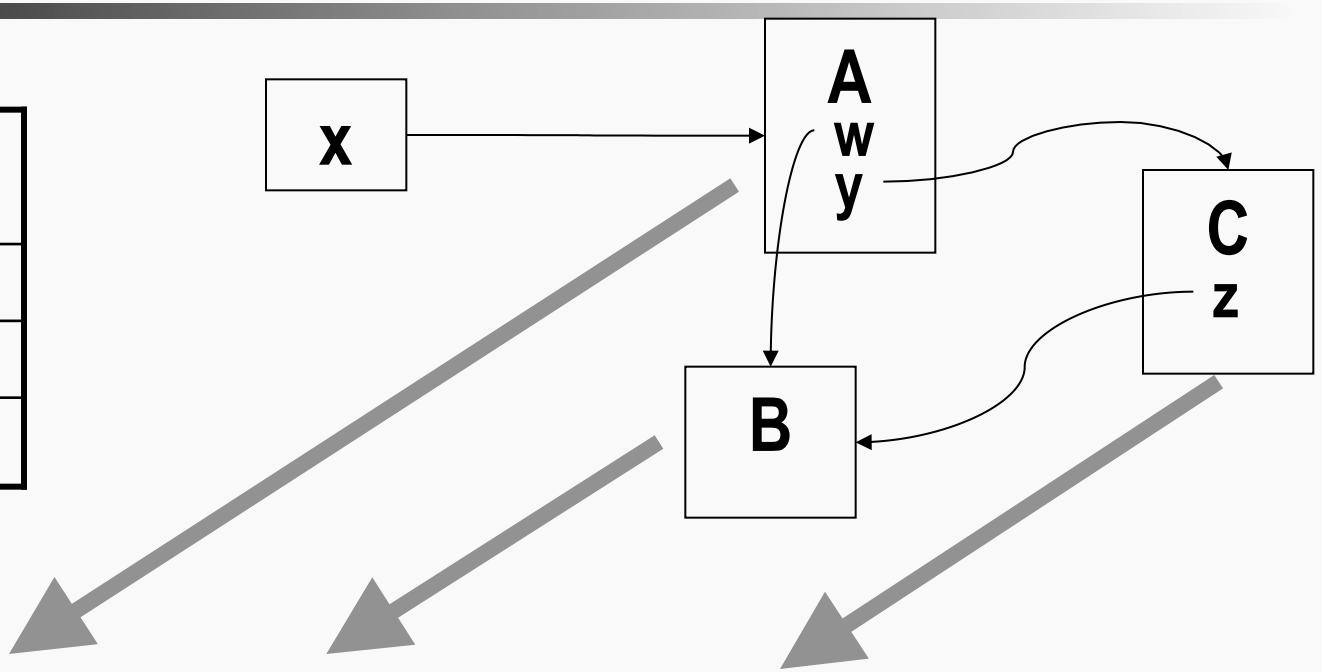
# Serializzazione di un oggetto

*Algoritmo semplificato*

- Ad ogni oggetto viene assegnato un numero di serie
- Quando si serializza un oggetto si verifica se tale oggetto è già stato serializzato
  - in tal caso, si riferisce l'oggetto tramite il suo numero di serie;
  - altrimenti si serializza l'oggetto

# Serializzazione di un oggetto

| Oggetto | Numero di serie |
|---|---|
| *A* | 1 |
| *B* | 2 |
| *C* | 3 |




| Serial nr = 1<br>w = 2<br>y = 3 | Serial nr = 2 | Serial nr. = 3<br>z = 2 |
|---|---|---|
| A | B | C |

# Serializzazione di una classe

- *Una classe è serializzabile se implementa l'interfaccia* **Serializable**

```
import java.io.Serializable;
public class MySerializableClass implements Serializable {
   // body
}
```

  - **Serializable** non contiene alcun membro
  - **NotSerializableException *Classname***

- *Se una classe è serializzabile, le sue sottoclassi sono serializzabili*

# Serializzazione: comportamento di default

Il comportamento di *default* consiste nella scrittura delle seguenti informazioni

- *la classe dell'oggetto*
  - **serialVersionUID**: definito di default (da evitare) o specificato dal programmatore (preferibile)

    **ANY-ACCESS-MODIFIER static final long serialVersionUID = 42L;**
    (recommeded **private**)
- *la signature della classe*
- *tutti i valori dei campi, inclusi i riferimenti, ma esclusi i campi* **static** *ed i campi* **transient**

# Serializzazione ed ereditarietà

## *Serializzazione di un sottotipo di una classe non serializzabile*

```
class A {
    [public|protected] A() {
        // constructor body
    }
    // other members
}

class B extends A,
        implements Serializable {
    // class body
}
```

***Durante la serializzazione***

- La sottoclasse (**B**) *può* gestire la serializzazione dei campi **public**, **protected** e **package** della superclasse
- La superclasse (**A**) deve avere un costruttore *no-arg* **public** o **protected**
  (*Errore a run-time se tale costruttore non è disponibile*)

***Durante la deserializzazione***

- I campi della sottoclasse serializzabile (**class B**) sono deserializzati (inizializzati dallo stream)
- I campi della classe non-serializzabile (**class A**) sono inizializzati per mezzo dal costruttore *no-arg*

# Personalizzare la serializzazione (1)

*Per personalizzare la serializzazione di una classe bisogna dotare la classe dei metodi*

```
private void writeObject(java.io.ObjectOutputStream out)
                                                throws IOException;
private void readObject(java.io.ObjectInputStream in)
                                throws IOException, ClassNotFoundException;
```

# Personalizzare la serializzazione (2)

```
private void writeObject(java.io.ObjectOutputStream out)
                                                    throws IOException {
    out.defaultWriteObject(); // default serialization mechanism
    // customised serialization code
}

private void readObject(java.io.ObjectInputStream in)
        throws IOException, ClassNotFoundException {
    in.defaultReadObject(); // default serialization mechanism
    // customised de-serialization code
    // followd by code to update the object, if necessary
}
```

# Personalizzare la serializzazione (2)

- I metodi **writeObject** e **readObject** si occupano di serializzare, de-serializzare, soltanto la classe immediata;
- la serializzazione della superclasse è gestita automaticamente.
- Tuttavia, se una classe deve coordinarsi con le superclassi per serializzarsi allora deve implementare l'interfaccia **Externalizable**

# *Interfaccia* Externalizable *(1)*

```
package java.io;
interface Externalizable extends Serializable {
    public void writeExternal(ObjectOutput out)
                                        throws IOException;
    public void readExternal(ObjectInput in)
                                        throws IOException, ClassNotFoundException;
}
```

- l'implementazione dell'interfaccia **Externalizable** permette un controllo completo della serializzazione
- per un oggetto **Externalizable**, solo l'identità della sua classe viene automaticamente scritta sullo stream

# *Interfaccia* Externalizable *(2)*

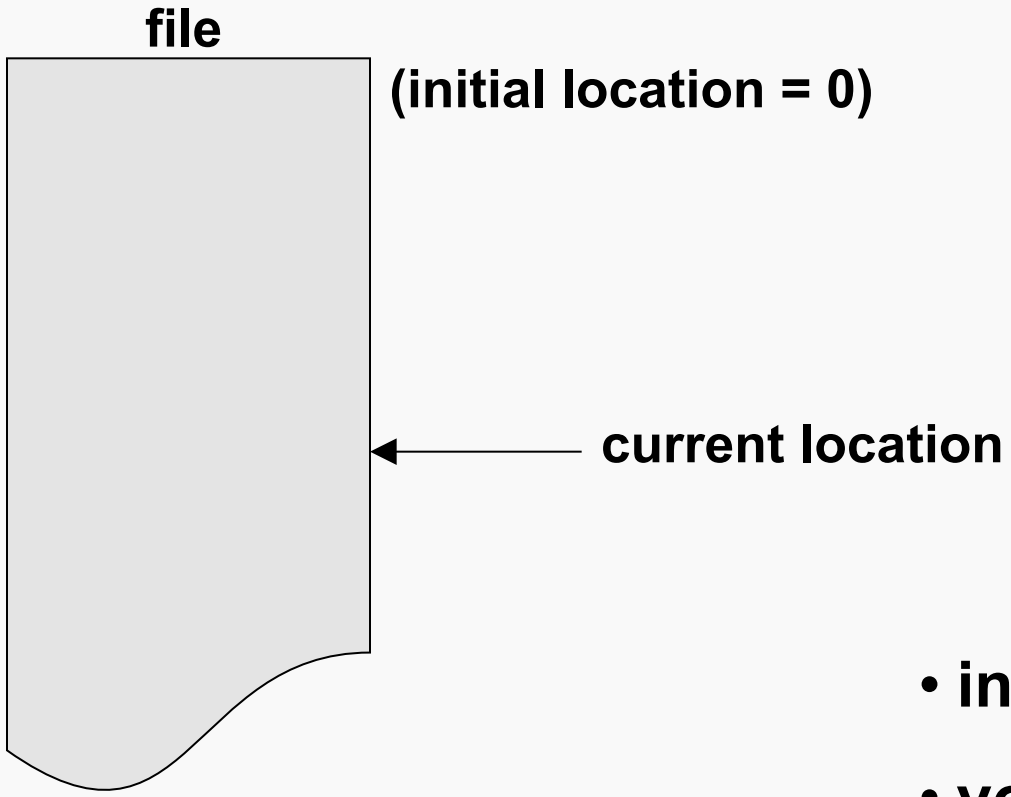
- Una classe **Externalizable**
  - deve implementare l'interfaccia **java.io.Externalizable**
  - deve implementare un metodo **writeExternal** per salvare lo stato dell'oggetto; deve anche esplicitamente coordinarsi con la sua superclasse per salvare il suo stato;
  - deve implementare un metodo **readExternal** per ripristinare lo stato dell'oggetto; deve anche esplicitamente coordinarsi con la sua superclasse per ripristinare il suo stato;
- se è stato definito un formato esterno, **writeExternal** e **readExternal** sono i soli responsabili del formato

# File ad accesso casuale

La classe **RandomAccessFile** può essere utilizzata contemporaneamente per leggere e per scrivere

- implementa le ***interfacce*** **DataInput** e **DataOutput** (i filtri funzionano anche su **RandomAccessFile**)
- richiede che sia specificato sia il file da aprire quando viene creato un oggetto sia la ***modalità di apertura*** ("r", "rw")
- supporta la nozione di ***file pointer*** (**getFilePointer**, **seek**)

# File pointer




- int skipBytes(int)
- void seek(long)
- long getFilePointer()

# Le interfacce DataInput e DataOutput

- La classe **ObjectOutputStream** (**ObjectInputStream**) implementa l'interfaccia **DataOutput** (**DataInput**) che definisce metodi per scrivere tipi primitivi come **writeInt** (**readInt**), **writeFloat** (**readFloat**), or **writeUTF** (**writeUTF**).
- Tali metodi possono essere utilizati per scrivere (leggere) tipi primitivi su (da) un **ObjectOutputStream** (**ObjectInputStream**)

# References

*Object Serialization Specification*

- HTML: http://java.sun.com/j2se/1.5.0/docs/guide/serialization/spec/serialTOC.html
- PDF: http://java.sun.com/j2se/1.5/pdf/serial-1.5.0.pdf