Laboratorio di Reti
Lezione 5
Stream based I/O, Serializzazione

15/10/2020

Laura Ricci

INPUT/OUTPUT IN JAVA

- I/O: reperire informazioni da una sorgente esterna o inviare ad una sorgente esterna
  - *file system*: files e directories
  - *connessioni di rete*
  - *keyboard*: `System.in`, `System.out`, `System.error`
  - *in-memory buffers* (array)
    - "vista" di un buffer di memoria come una sorgente o destinazione esterna.
      - un programma legge da un file csv.
      - per ottimizzare l'accesso ai dati si legge tutto il file in un buffer, in memoria centrale.
      - l'interfaccia verso il modulo che gestisce i dati deve rimanere la solita

- di particolare interesse per il corso:
  - connessioni di rete modellate come streams
  - in-memory buffers per la generazione di pacchetti UDP.

JAVA I/O

- definizione di un insieme di astrazioni per la gestione dell'I/O: una delle parti più complesse di un linguaggio
- diversi tipi di device di input/output: se il linguaggio dovesse gestire ogni tipo di device come caso speciale, la complessità sarebbe enorme
  - necessità di astrazioni opportune per rappresentare una device di I/O
- in JAVA, la prima astrazione definita è basata sul concetto di stream (o flusso)
- altre astrazioni per l'I/O
  - `File`: per manipolare descrittori di files
  - `Channels (NIO)`
  - ...

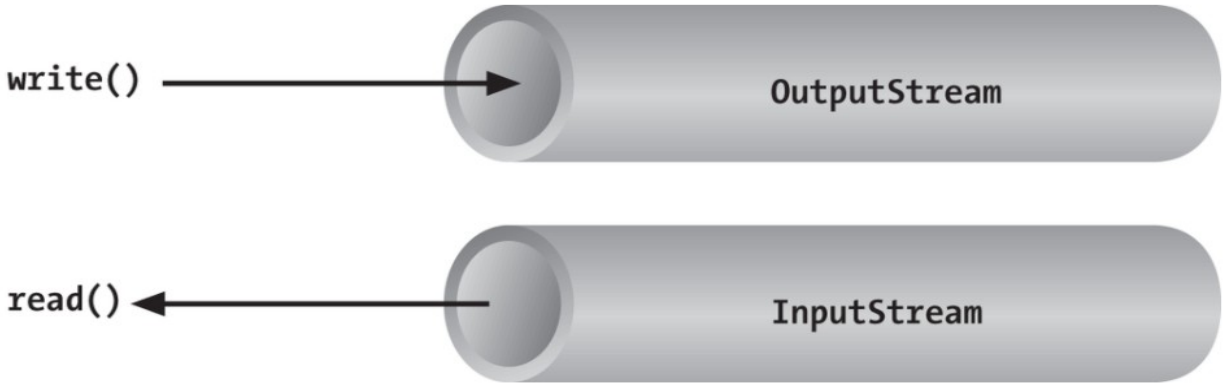
JAVA PACKAGES PER I/O

- programmare operazioni di I/O può risultare molto semplice. Diverso è il caso in cui si voglia programmare operazioni di I/O efficienti e portabili
  - molti packages per I/O (9 in JDK 1.7 !)

- classificazione I/O packages:
  - JAVA IO API, package standard I/O contenuto in `java.io`, introdotto già a partire da `JDK 1.0`
    - basato su stream
    - lavora su bytes e caratteri
  - JAVA NIO API: (New IO) funzionalità simili a `java.io`, ma con basato su canali e con comportamento non bloccante: migliori performance in alcuni scenari. Introdotta in `JDK 1.4`
  - JAVA NIO.2: alcuni ulteriori miglioramenti rispetto a `JAVA NIO`

L'ASTRAZIONE DEGLI STREAM

- uno stream rappresenta una connessione tra un programma JAVA ed un dispositivo esterno (file, buffer di memoria, connessione di rete,...)
- un flusso di informazione di lunghezza illimitata



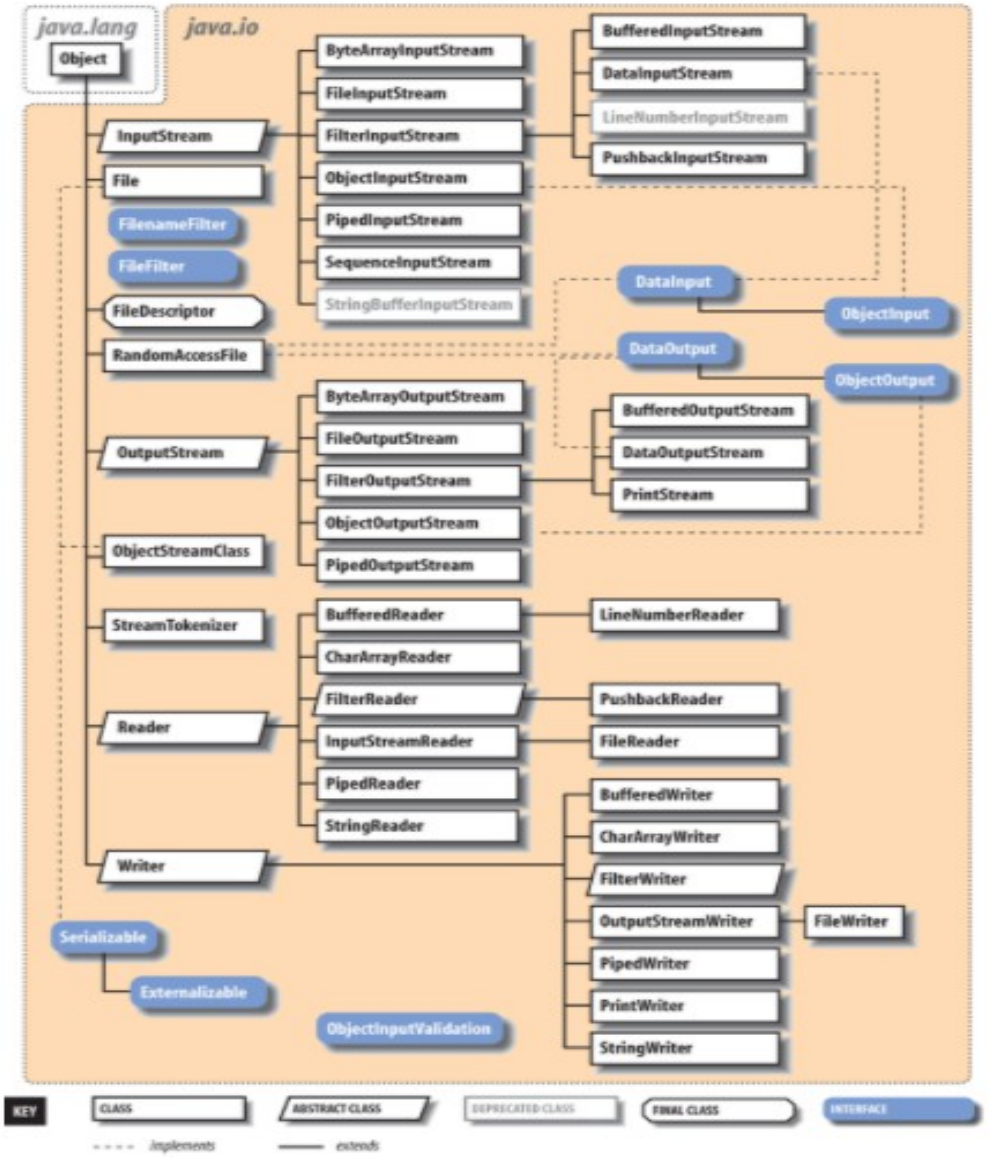
- un "tubo" tra una sorgente ed una destinazione (dal programma ad un dispositivo e viceversa)
- l'applicazione inserisce dati o li legge ad/da un capo dello stream
- i dati fluiscono da/verso la destinazione

JAVA STREAMS: CARATTERISTICHE GENERALI

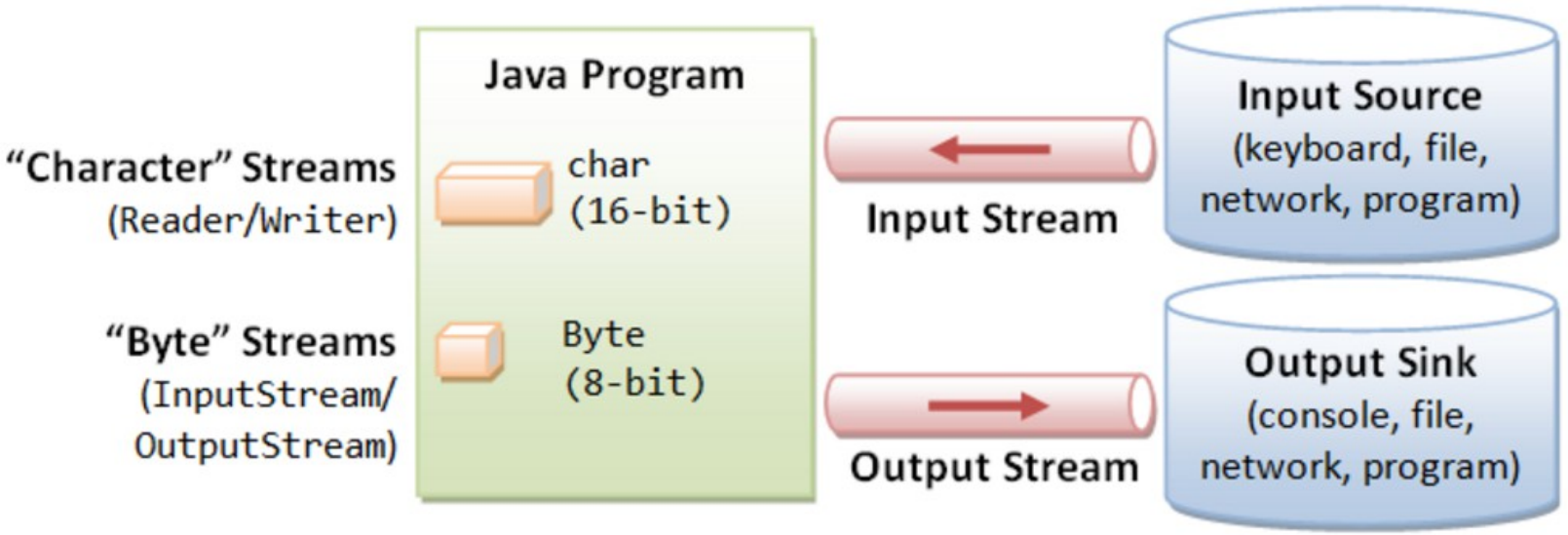
- accesso sequenziale
- mantengono l'ordinamento FIFO
- one way: read only oppure write only (a parte i file ad accesso random)
  - se un programma ha bisogno di dati in input ed output, è necessario aprire due stream, in input ed in output
- bloccanti: quando un'applicazione legge un dato dallo stream (o lo scrive) si blocca finchè l'operazione non è completata
- non è richiesta una corrispondenza stretta tra letture/scritture
  - una unica scrittura inietta 100 bytes sullo stream
  - i byte vengono letti con due write successive 'all'altro capo dello stream', la prima legge 20 bytes, la seconda 80 bytes)

LA GIUNGLA DELLE CLASSE IN JAVA IO

- 4 classi astratte fondamentali:
  - `InputStream, OutputStream, Reader, Writer`
- `Input/OutputStream`:
  - leggono e scrivono bytes
  - dati copiati byte a byte
    - non strutturati
    - senza alcuna traduzione
  - ideale per leggere/scrivere raw data in binario
    - un'immagine
    - la codifica di un video.
- `Readers/Writers` leggono/scrivono caratteri



LE CLASSI PRINCIPALI: CARATTERI E BYTE



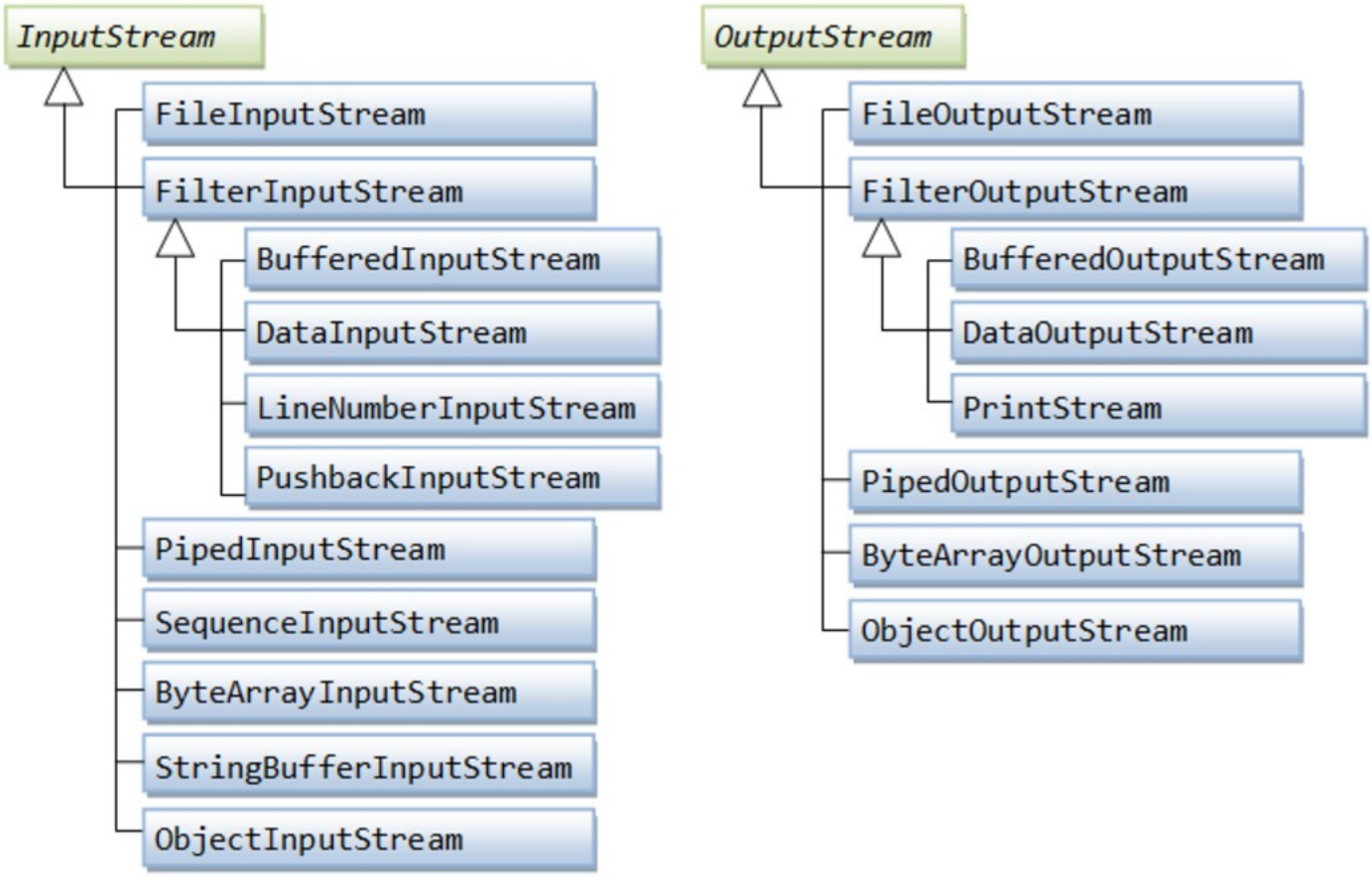
Internal Data Formats:

- Text (char): UCS-2
- int, float, double, etc.

External Data Formats:

- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

BYTE STREAM: GERARCHIA DI CLASSI



Input/OutputStream: sono classi astratte, diverse implementazioni

STREAM: CARATTERISTICHE GENERALI

- `Input/OutputStream`:
  - forniscono operazioni base
  - possono essere "associati" ad ogni tipo di device di input/output
- implementazioni diverse per tipi diversi di I/O:
  - console: `System.out`, `System.in`
  - files: `FileInputStream`, `FileOutputStream` per leggere/scrivere dati da un file
  - in-memory buffers: per trasferimento di dati da una parte all'altra di un programma JAVA: `ByteArrayOutputStreams`, `ByteArrayInputStreams`
    - utili per generare uno stream di byte, per poi trasferirlo in un pacchetto UDP.
  - connessioni TCP
  - `PipeInputStream`: pipes

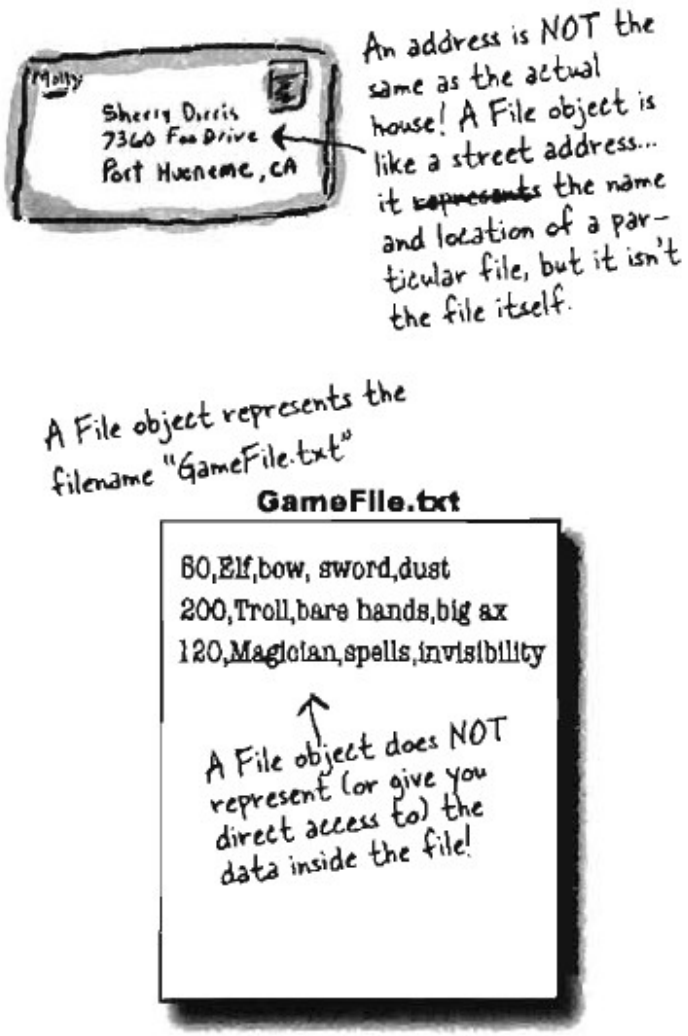
INPUTSTREAM

- `int read( )`
  - legge un byte dallo stream come un intero unsigned tra 0 e 255
  - restituisce -1 se viene individuata la "fine dello strem" (derivato dal C)
  - solleva una `IOException` se c'è un errore di I/O
  - bloccante
- `public int read(byte[] bytes)`
  - riempe il vettore passato in input con tutti i dati disponibili sullo stream fino alla capacità massima del vettore e restituisce il numero di byte letti
- `skip(long n), available(), close()` e molti altri metodi

```
int waiting = System.in.available(); //funziona solo per FileStream
if ( waiting > 0 ) {
        byte [] data = new byte [ waiting ];
        System.in.read( data );
      ...}
```

- diverse classi concrete implementano `InputStream`, `FileInputStream` legge un byte da un file, `System.in` legge byte a byte dalla keyboard

LA CLASSE FILE: DESCRITTORE DI FILE



un' istanza della classe `File` descrive:

- path per l'individuazione del file o della directory
- non una semplice stringa, ma offre metodi
  - per verificare l'esistenza del path
  - per restituire meta-informazioni sul file,…
- quando si vuole stabile una connessione (stream) con un file si può passare come parametro:
  - un oggetto di tipo File
  - una stringa
  - …..

LA CLASSE FILE: DESCRITTORE DI FILE

```
public class ListFiles {
   public static void main(String[] args) {
      File dir = new File(".");   // current working directory
      if (dir.isDirectory()) {
         // List only files that meet the filtering criteria
         String[] files = dir.list();
         for (String file : files) {
                if (file.endsWith(".java"))
                  System.out.println(file);}
         }
      }
   }
```

BYTE STREAM: COPIARE UN'IMMAGINE

```
import java.io.*;
public class CopyImage {
  public static void main(String[] args){
    {InputStream is = null; OutputStream os = null;
     try { is = new FileInputStream(new File("immagine.jpg"));
           os = new FileOutputStream(new File("immaginenew.jpg"));
           byte[] buffer = new byte[8192]; int length;
           while ((length = is.read(buffer)) > 0) {
                 os.write(buffer, 0, length); } }
       catch(Exception e){e.printStackTrace();}
       finally {
         if (is!=null)
            try {is.close();} catch(Exception e){e.printStackTrace();}
         if (os!=null)
            try {os.close();} catch(Exception e){e.printStackTrace();}
       }}}}
```

TRY FINALLY: PROBLEMI

```
private static void printFile() throws IOException {
    InputStream input = null;
    try {
        input = new FileInputStream("file.txt");
        int data = input.read();
        while(data != -1){
            System.out.print((char) data);
            data = input.read(); }
       } finally {
           if(input != null){
            input.close();
        } } }
```

- le istruzioni marcate in rosso, sono quelle che possono sollevare Eccezioni.
- il blocco finally block viene sempre eseguito, sia che una eccezione sia sollevata dal try block che no..
- chiusura dello stream deve essere effettuata in ogni caso

TRY FINALLY: SUPPRESSED EXCEPTIONS

```
private static void printFile() throws IOException {
    InputStream input = null;
    try {input = new FileInputStream("file.txt");
        int data = input.read();
        while(data != -1){
            System.out.print((char) data);
            data = input.read(); }
       } finally {
           if(input != null){
            input.close(); } } }
```

- supponiamo che una eccezione sia sollevata
  - nel blocco `try`, oppure
  - nel blocco `finally`, che viene eseguito in ogni caso
- quale eccezione viene propagata nello stack delle chiamate?
  - quella sollevata nel blocco `finally`, anche se quella più rilevante è quella del blocco `try`

TRY WITH RESOURCES

- introdotto in JAVA 7
- supporto per la chiusura sistematica delle risorse e/o connessioni aperte da un programma
  - si occupa di chiudere automaticamente le risorse aperte.
- si utilizza un blocco `try` con uno o più argomenti tra parentesi.
  - gli argomenti sono le risorse che JAVA garantisce di chiudere al termine del blocco
  - suppressed exceptions:
    - quando si verificano delle eccezioni sia nel blocco try-with-resources sia durante la chiusura, la JVM sopprime l'eccezione generata nella chiusura automatica.
  - possono comunque essere inseriti blocchi catch e finally che vengono comunque eseguiti dopo la chiusura delle risorsa.

TRY WITH RESOURCES

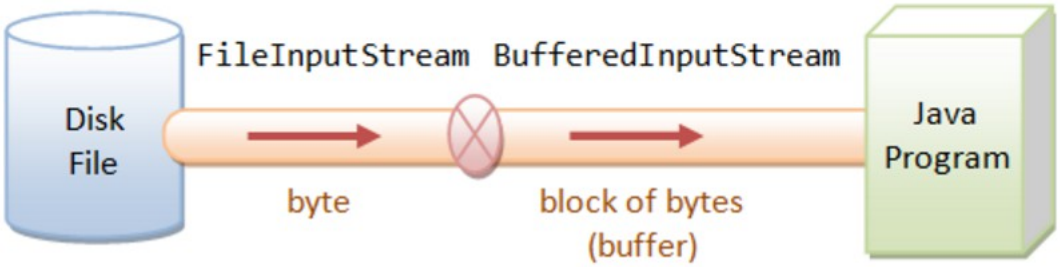
```
import java.io.*;
public class trywithresources
       { public static void main (String args[])throws IOException {
         try(  FileInputStream input = new FileInputStream(new
                                                File("immagine.jpg"));
               BufferedInputStream bufferedInput = new
                                                 BufferedInputStream(input))
           {
            int data = bufferedInput.read();
            while(data != -1){
                 System.out.print((char) data);
                 data = bufferedInput.read();
                  }
    } }}
```

JAVA: FILTER STREAMS

- `InputStream` and `OutputStream` operano su "row bytes"
- classi filtro compiono trasformazioni sui dati a basso livello. Tipi di filtri:
- Filter Stream: trasformazioni effettuate
  - crittografia
  - compressione
  - Buffering
  - traduzione dei dati in un formato a più alto livello
- Readers Writes
  - orientati al testo e permettono di decodificare bytes in caratteri
- I filtri possono essere organizzati in catena. Ogni elemento della catena
  - riceve dati dallo stream o dal filtro precedente
  - passa i dati al programma o al filtro successivo

BUFFERED INPUT ED OUTPUT STREAM

- implementano una bufferizzazione per stream di input e di output,
- i dati vengono scritti e letti in blocchi di bytes, invece che un solo blocco per volta
- miglioramento significativo della performance




```
FileOutputStream outputFile = new FileOutputStream("primitives.data");

BufferedOutputStream bufferedOutput = new BufferedOutputStream(outputFile);
```

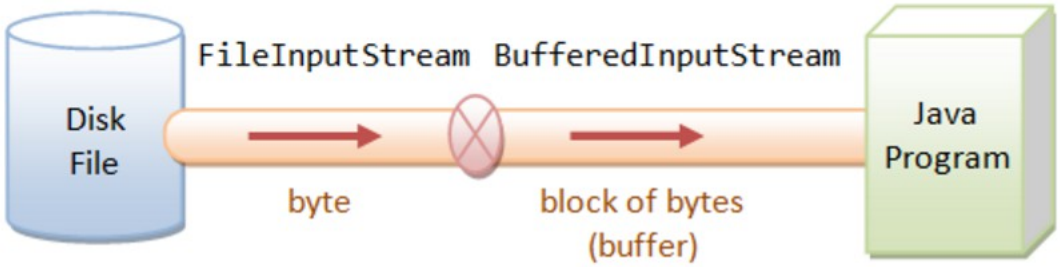
VALUTARE EFFETTI BUFFERIZZAZIONE

```
import java.io.*;
public class FileCopyNoBufferJDK7 {
   public static void main(String[] args) {
      String inFileStr  = "blue_i1.jpg";
      String outFileStr = "blue_i1_new.jpg";
      long startTime, elapsedTime;  // for speed benchmarking
      // Check file length
      File fileIn = new File(inFileStr);
      System.out.println("File size is " + fileIn.length() + " bytes");
      // "try-with-resources" automatically closes all opened resources.
      try (FileInputStream  in  = new FileInputStream(inFileStr);
           FileOutputStream out = new FileOutputStream(outFileStr))
          { startTime = System.nanoTime();
            int byteRead;
```

VALUTARE EFFETTI BUFFERIZZAZIONE

```
        // Read a raw byte, returns an int of 0 to 255.
        while ((byteRead = in.read()) != -1)
         {
           // Write the least-significant byte of int, drop the upper 3 bytes
           out.write(byteRead);
          }
           elapsedTime = System.nanoTime() - startTime;
           System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + "
                                msec");
         } catch (IOException ex) { ex.printStackTrace(); }
   }
}
File size is 955399 bytes
Elapsed Time is 4684.12669 msec
```

VALUTARE EFFETTI BUFFERIZZAZIONE



Modificando il programma precedente come segue:

```
try
    (BufferedInputStream in = new BufferedInputStream(new
                              FileInputStream(inFileStr));
     BufferedOutputStream out = new BufferedOutputStream(new
                              FileOutputStream(outFileStr)))
  { ……  }
```

si ottiene:

```
File size is 955399 bytes
Elapsed Time is 44.777895 msec
```

FORMATTED DATA STREAMS

```
import java.io.*;
public class TestDataIOStream {
   public static void main(String[] args) {
      String filename = "data-out.dat"; String message = "Hi,$%&!";
      // Write primitives to an output file
      try (DataOutputStream out =
               new DataOutputStream(
                  new BufferedOutputStream(
                     new FileOutputStream(filename)))) {
         out.writeByte(127);
         out.writeShort(-1);
         out.writeInt(43981);
         out.writeLong(305419896);
         out.writeFloat(11.22f);
         out.writeDouble(55.66);
         out.writeBoolean(true); out.writeBoolean(false);
         for (int i = 0; i < message.length(); ++i) {
```

FORMATTED DATA STREAMS

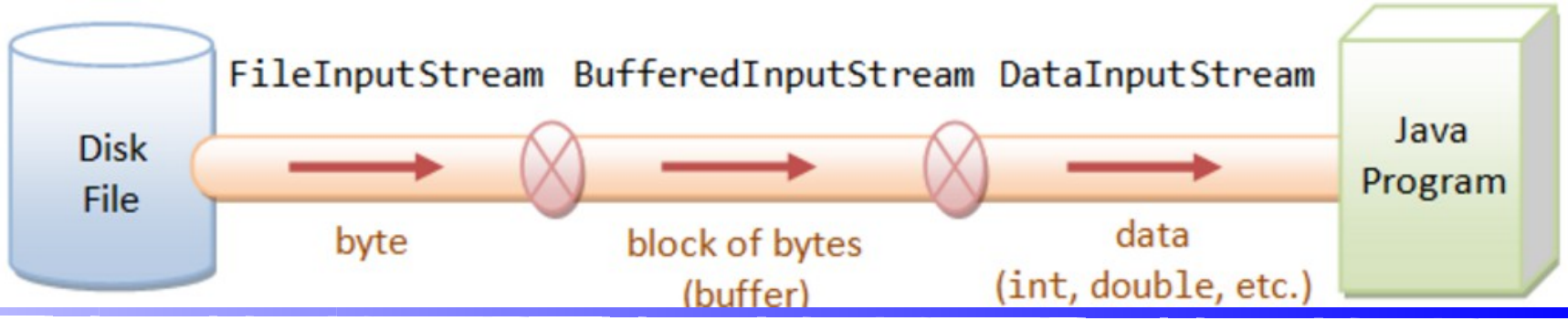
```
    for (int i = 0; i < message.length(); ++i) {
        out.writeChar(message.charAt(i)); }
    out.writeChars(message);
    out.writeBytes(message);
    out.flush();
} catch (IOException ex) { ex.printStackTrace(); }
```

FORMATTED DATA STREAMS

```
// Read raw bytes and print in Hex
try (BufferedInputStream in =
        new BufferedInputStream(
           new FileInputStream(filename))) {
   int inByte;
   while ((inByte = in.read()) != -1) {
      System.out.printf("%02X ", inByte);
      System.out.println();// Print Hex codes
   }
   System.out.printf("%n%n");
} catch (IOException ex) { ex.printStackTrace();}
```

FORMATTED DATA STREAMS

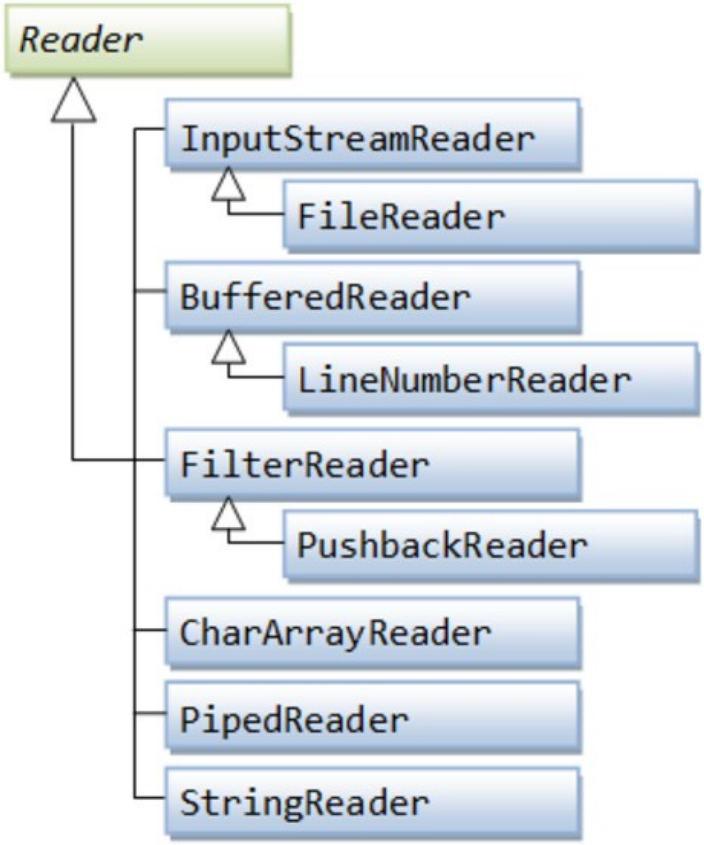
```
// Read primitives
try (DataInputStream in =
        new DataInputStream(
           new BufferedInputStream(
              new FileInputStream(filename)))) {
   System.out.println("byte:    " + in.readByte());
   System.out.println("short:   " + in.readShort());
   System.out.println("int:     " + in.readInt());
   System.out.println("long:    " + in.readLong());
   System.out.println("float:   " + in.readFloat());
   System.out.println("double:  " + in.readDouble());
   System.out.println("boolean: " + in.readBoolean());
```

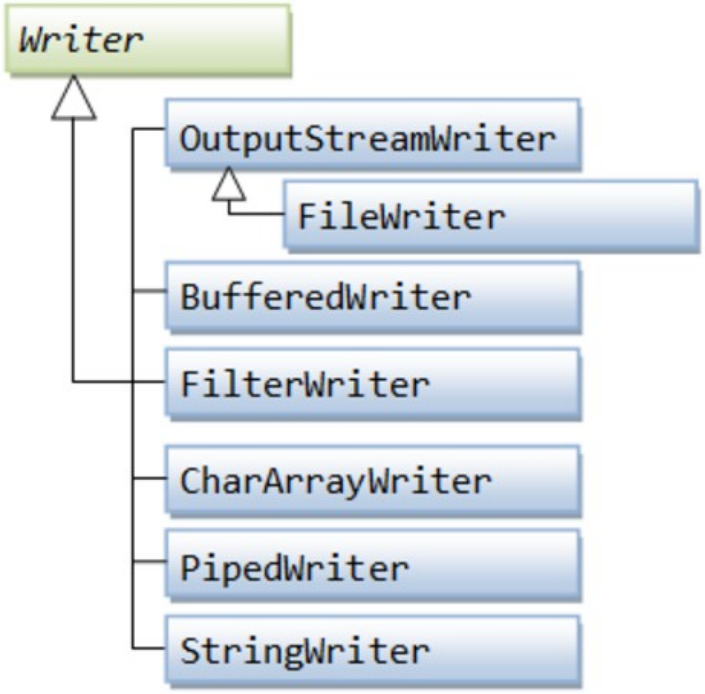


FORMATTED DATA STREAMS

```
System.out.print("char:    ");
for (int i = 0; i < message.length(); ++i) {
        System.out.print(in.readChar()); }
System.out.println();
System.out.print("chars:   ");
for (int i = 0; i < message.length(); ++i) {
        System.out.print(in.readChar()); }
System.out.println();
System.out.println();
System.out.print("bytes:   ");
for (int i = 0; i < message.length(); ++i) {
        System.out.print((char)in.readByte()); }
 System.out.println();
   }
catch (IOException ex) { ex.printStackTrace(); } } }
```

CHARACTER STREAMS





SET OF CHARACTER YOU CAN USE

ISO-8859-1 - Western Alphabet

	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

ISO-8859-5 - Cyrillic Alphabet

	Ё	Ђ	Ѓ	Є	S	I	Ї	J	Љ	Њ	Ћ	Ќ	-	Ў	Џ
А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я
№	ё	ђ	ѓ	є	s	i	ї	j	љ	њ	ћ	ќ	§	ў	џ

JIS X 0208 - Japanese Alphabet

	概	帰	毅	気	汽	畿	祈	季	稀	紀	徽	規	記	貴	起
軌	輝	飢	騎	鬼	亀	偽	儀	妓	宜	戯	技	擬	欺	犠	疑
祇	義	蟻	誼	議	掬	菊	鞠	吉	吃	喫	桔	橘	詰	砧	杵
黍	却	客	脚	虐	逆	丘	久	仇	休	及	吸	宮	弓	急	救
朽	求	汲	泣	灸	球	究	窮	笈	級	糾	給	旧	牛	去	居
巨	拒	拠	挙	渠	虚	許	距	鋸	漁	禦	魚	亨	享	京	

ISO-8859-7 - Greek Alphabet

	‘	’	£			¦	§	¨	©		«	¬	-		―
°	±	²	³	΄	΅	Ά	·	Έ	Ή	Ί	»	Ό	½	Ύ	Ώ
ΐ	Α	Β	Γ	Δ	Ε	Ζ	Η	Θ	Ι	Κ	Λ	Μ	Ν	Ξ	Ο
Π	Ρ		Σ	Τ	Υ	Φ	Χ	Ψ	Ω	Ϊ	Ϋ	ά	έ	ή	ί
ΰ	α	β	γ	δ	ε	ζ	η	θ	ι	κ	λ	μ	ν	ξ	ο
π	ρ	ς	σ	τ	υ	φ	χ	ψ	ω	ϊ	ϋ	ό	ύ	ώ	

CHARACTERS STREAMS: ENCODING

- Character Set
- Coded Character Set: un character set in cui ad ogni carattere viene assegnato un valore numerico, o code point
- Encoding: come i numeri che appartengono ad un coded character set sono rappresentati : 8 bit, una parola, una sequenza di bytes,...

- coded character sets comuni:
  - ASCII (128 caratteri, 7 bits encoding all'interno di un byte)
  - ISO 8859-1 (caratteri identici as ASCII da 0 a 128, altri caratteri da 160 a 255, 8 bit encoding, utilizzato da sistemi UNIX-based)
  - Windows 1252 (CP 1252) (come ASCII da 0 a 128, come ISO 8859 da 160 a 255 + altri caratteri, 8 bit encoding, utilizzato in Windows).
  - Unicode Universal character set, gestito dall'Unicode Consortium (unicode.org)

CODE POINTS

A = 41_{hex} (ASCII)

a = 61_{hex} (ASCII)

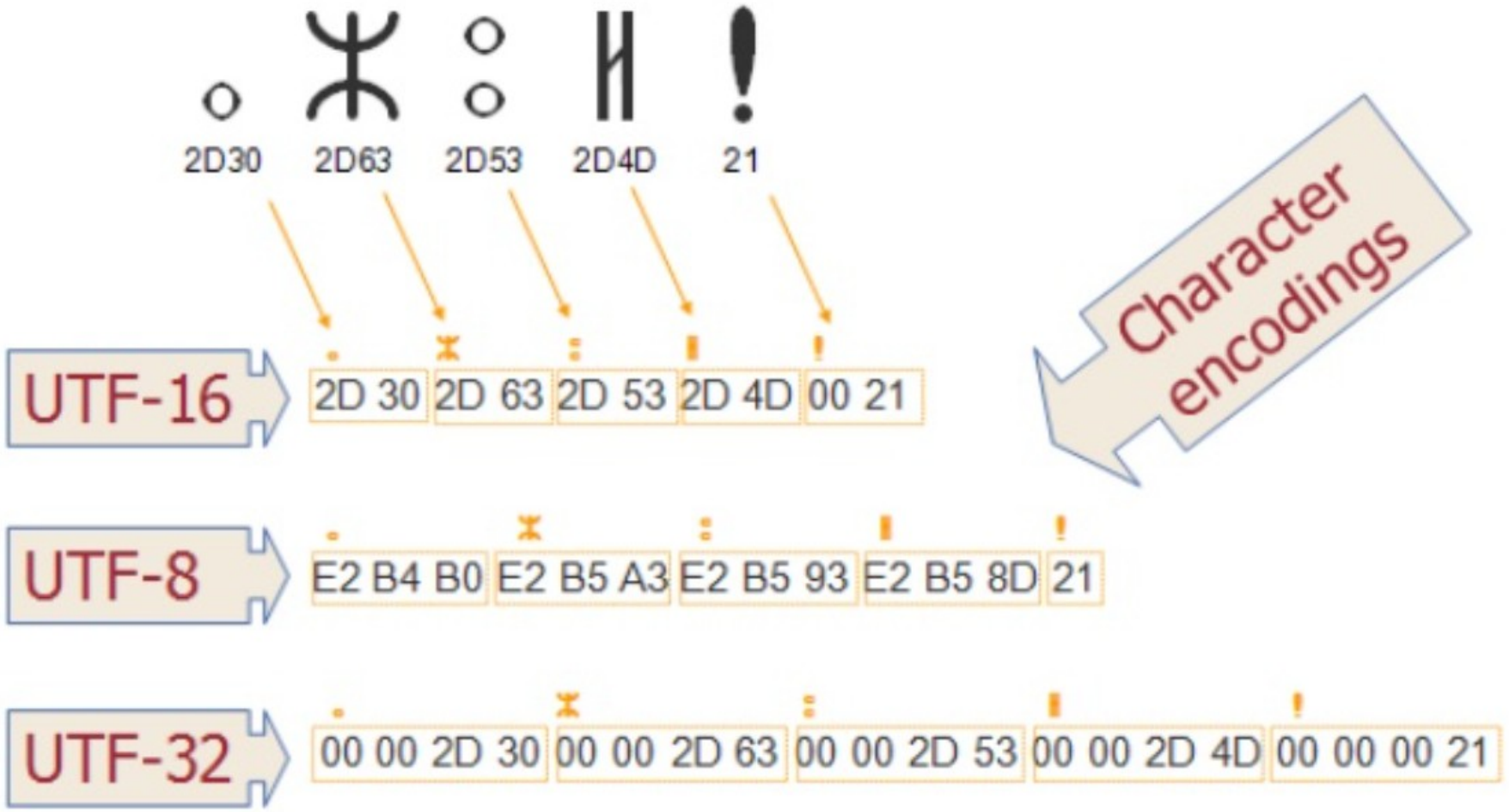
A = 00_{hex} 41_{hex} (UNICODE)

a = 00_{hex} 61_{hex} (UNICODE)

イニング = 33_{hex} 34_{hex} (UNICODE)

絳 = 42_{hex} $F4_{hex}$ (UNICODE)

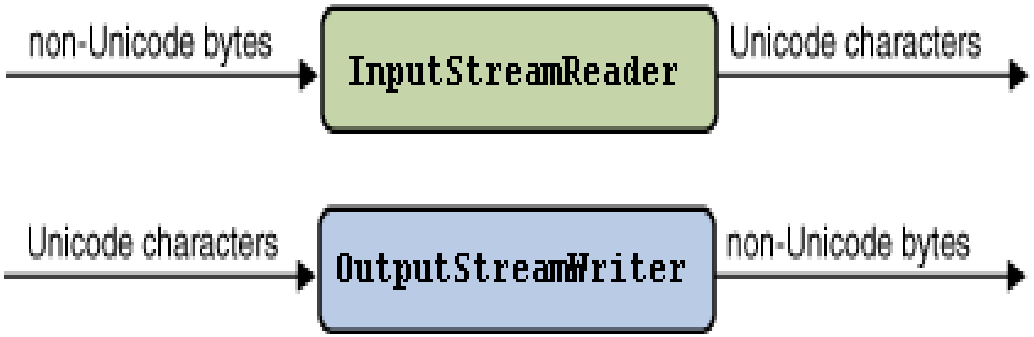
CHARACTER ENCODINGS



INPUTSTREAMREADER

- deve essere istanziato su un `InputStream`
- prende come parametro la codifica dei caratteri presenti sullo stream di byte
- permette di specificare diversi encoding (`UTF-8`, `UTF-16`,...)
- default charset
  - settato dalla JVM al momento dello start-up
  - dipendente dal sistema operativo

```
System.out.println("Codifica"+Charset.defaultCharset().displayName());
> windows-1252”
```

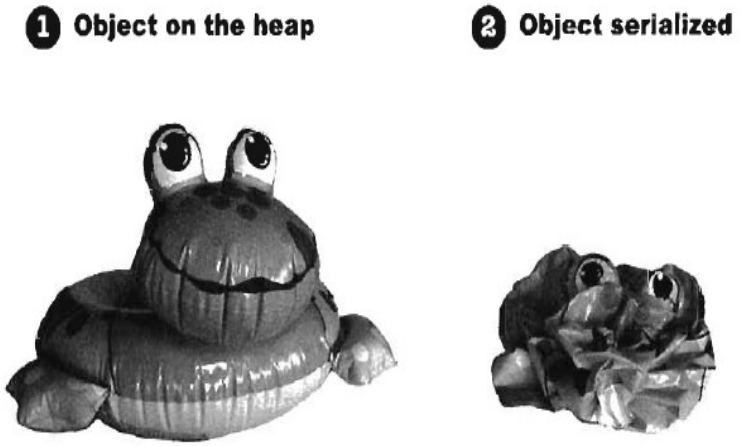


READERS E WRITERS

```
import java.io.*;
public class Encoding {
  public static void main(String[] args) throws
                  java.io.UnsupportedEncodingException {
  try {
   BufferedReader rdr = new BufferedReader(
                        new InputStreamReader(
                         new FileInputStream("filename"),
                                                "UTF-8"));
   String line = rdr.readLine();
   System.out.println(line);
         } catch (IOException exc) {
                  System.err.println("I/O error"); }
    }
```

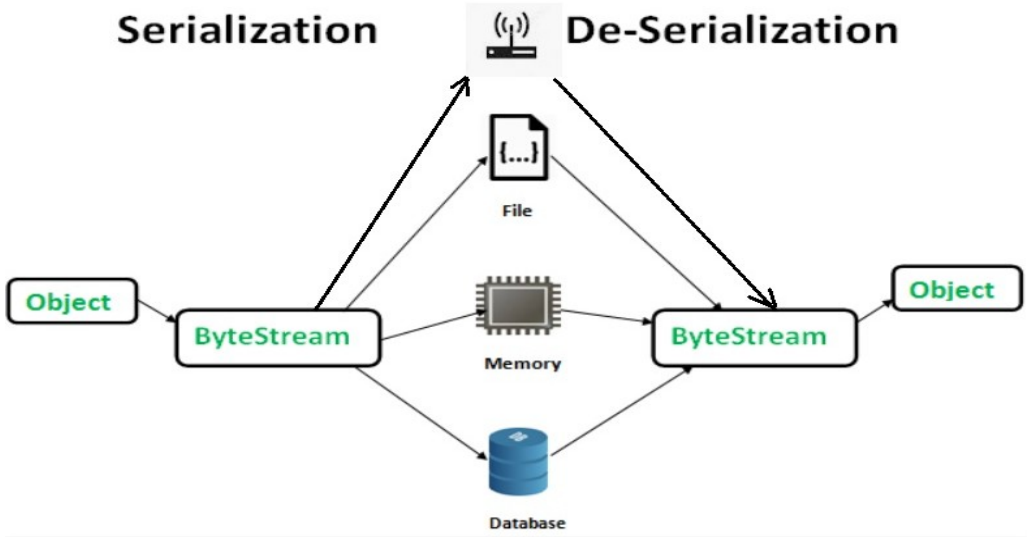
SERIALIZZAZIONE DI OGGETTI

- gli oggetti esistono in memoria fino a che la JVM è in esecuzione:
  - per la loro persistenza al di fuori della JVM, occorre
    - creare una rappresentazione dell' oggetto indipendente dalla JVM
    - meccanismi di serializzazione
- ogni oggetto è caratterizzato da uno stato e da un comportamento
  - comportamento: specificato dai metodi della classe
  - stato: "vive" con l'istanza dell'oggetto
  - la serializzazione effettua il flattening dello stato dell'oggetto
  - la deserializzazione ricostruisce lo stato dell'oggetto



PERSISTANZA ED INVIO DI OGGETTI

- l'oggetto serializzato può quindi essere scritto su un qualsiasi stream di output



- come useremo la serializzazione in questo corso?
  - per inviare oggetti
    - su uno stream che rappresenta una connessione TCP
    - come parametri di metodi invocati via Remote Method Invocation
  - per generare pacchetti UDP, si scrive l'oggetto serializzato su uno stream di byte e poi si genera un pacchetto UDP

SERIALIZZAZIONE JAVA: HOW TO DO

- `Serializable Interface`
  - per rendere un oggetto "persistente", l'oggetto deve implementare l'interfaccia `Serializable`
  - marker interface: nessun metodo, solo informazione su un oggetto per il compilatore e la JVM
  - controllo limitato sul meccanismo di linearizzazione dei dati
  - tutti i tipi di dato primitivi sono serializzabili
  - gli oggetti, se implementano `Serializable`, sono serializzabili
    - a parte alcuni oggetti....(vedi slide successive)
- `Externizable Interface`
  - estende `Serializable`
  - consente creare un proprio protocollo di serializzazione
    - ottimizzare la rappresentazione serializzata dell'oggetto
    - implementazione metodi `readExternal` e `writeExternal`

SERIALIZZAZIONE JAVA: HOW TO DO

```
import java.io.Serializable;
import java.util.Date;
import java.util.Calendar;
public class PersistentTime implements Serializable
       { private static final long serialVersionUID = 1;
         private Date time;
         public PersistentTime()
             {time = Calendar.getInstance().getTime(); }
         public Date getTime()
              {return time; } }
```

in rosso le parti relative alla serializzazione

Regola #1: per serializzare un oggetto persistente la classe di cui l'oggetto è istanza deve implementare l'interfaccia `Serializable` oppure ereditare l'implementazione dalla sua gerarchia di classi

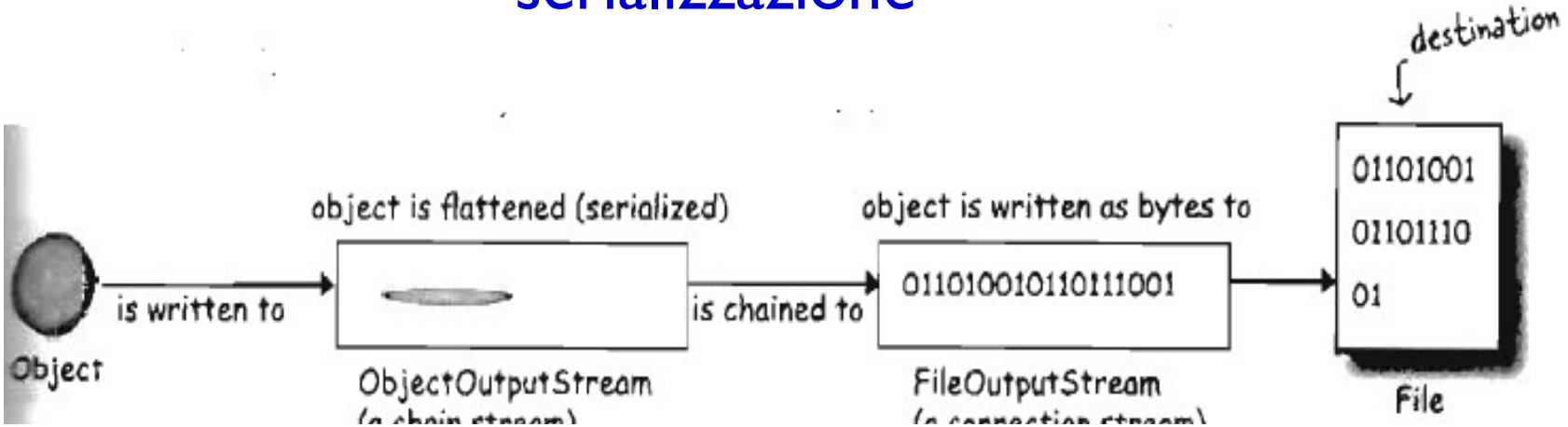
SERIALIZZAZIONE JAVA: HOW TO DO

```
import java.io.*;
public class FlattenTime
   {public static void main(String [] args)
       {String filename = "time.ser";
        if(args.length > 0) { filename = args[0]; }
        PersistentTime time = new PersistentTime();
        try(
           FileOutputStream fos = new FileOutputStream(filename);
           ObjectOutputStream out = new ObjectOutputStream(fos);)
                { out.writeObject(time);}
           catch(IOException ex) {ex.printStackTrace();
         }}}
```

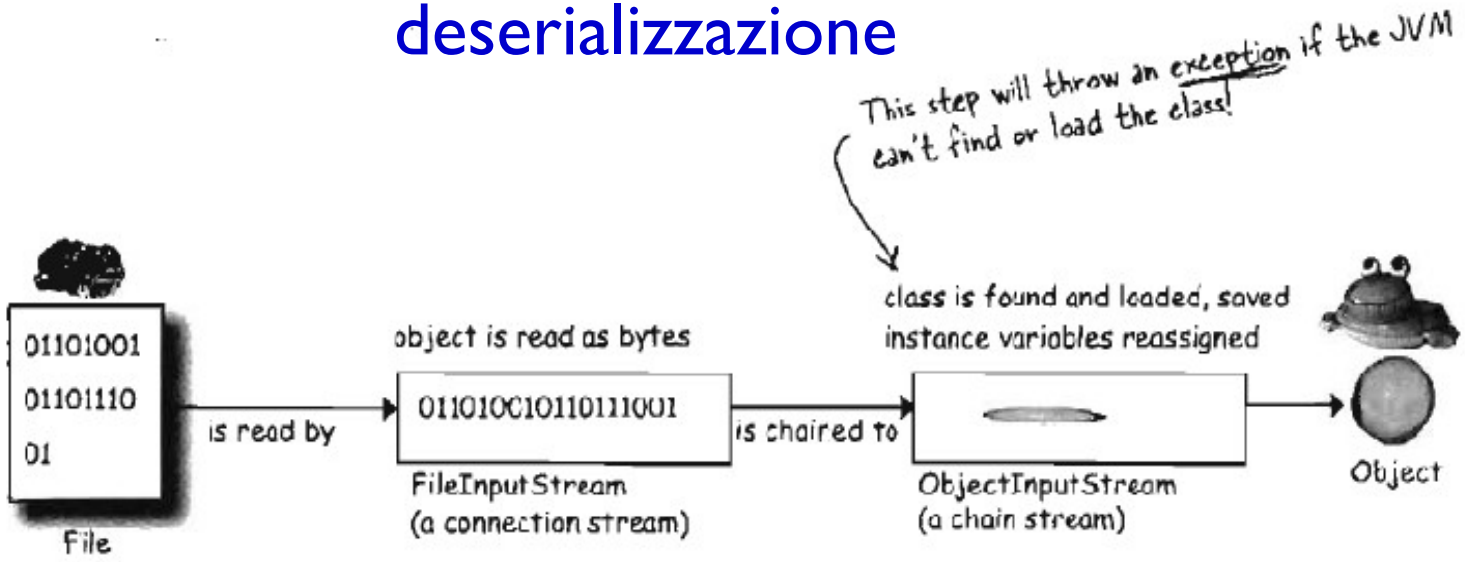
- la serializzazione vera e propria è gestita dalla classe `ObjectOutputStream`
- tale stream deve essere concatenato con uno stream di bytes, che può essere un `FileOutputStream`, uno stream di bytes associato ad un socket, uno stream di byte generato in memoria,...

SERIALIZZAZIONE E DESERIALIAZZAZIONE

serializzazione



deserializzazione



SERIALIZZAZIONE E GERARCHIA DELLE CLASSI

- La serializzazione è un processo ricorsivo, l'oggetto serializzato può contenere altri oggetti
- quando un oggetto viene serializzato, si percorre la gerarchia delle superclassi e si salva lo stato di ogni classe, fino a che non si trova la prima classe non serializzabile

DESERIALIZZAZIONE

```
public class InflateTime
    {public static void main(String [] args)
      {String filename = "time.ser";
       if(args.length > 0) {filename = args[0]; }
       PersistentTime time = null; FileInputStream fis = null;
       ObjectInputStream in = null;
       try(
         FileInputStream fis = new FileInputStream(filename);
         ObjectInputStream in   = new ObjectInputStream(fis);)
         {time = (PersistentTime)in.readObject();}
  catch(IOException ex)
     { ex.printStackTrace(); }
  catch(ClassNotFoundException ex)
     {ex.printStackTrace();}
```

in rosso le parti relative alla deserializzazione

DESERIALIZZAZIONE

```
// print out restored time
   System.out.println("Flattened time: " + time.getTime());
   System.out.println();
     // print out the current time
   System.out.println("Current time: "+
                          Calendar.getInstance().getTime());}
}
```

```
Output ottenuto:
Flattened time: Mon Mar 12 19:11:55 CET 2012
Current   time: Mon Mar 12 19:16:24 CET 2012
```

`ClassNotFoundException`: l'applicazione tenta di caricare una classe, ma non trova nessuna definizione di una classe con quel nome

DESERIALIZZAZIONE

- il metodo `readObject()` legge la sequenza di bytes memorizzati in precedenza e crea un oggetto che è l'esatta replica di quello originale
  - `readObject` può leggere qualsiasi tipo di oggetto, è necessario effettuare un cast al tipo corretto dell'oggetto

- la JVM determina, mediante informazione memorizzata nell'oggetto serializzato, il tipo della classe dell'oggetto e tenta di caricare quella classe o una classe compatibile

- se non la trova viene sollevata una `ClassNotFoundException` ed il processo di deserializzazione viene abortito

- altrimenti, viene creato un nuovo oggetto sullo heap
  - lo stato di tutti gli oggetti serializzati viene ricostruito cercando i valori nello stream, senza invocare il costruttore (uso di Reflection)
  - si percorre l'albero delle superclassi fino alla prima superclasse non-serializzabile. Per quella classe viene invocato il costruttore

COSA NON E' SERIALIZZABILE?

- oggetti contenti riferimenti specifici alls JVM o al SO (JAVA native class)
  - `Thread`, `OutputStream`, `Socket`, `File`, non possono essere ricreati, perché contengono riferimenti specifici al particolare ambiente di esecuzione

- le variabili marcate come `transient`
  - ad esempio variabili che non devono essere scritte per questioni di privacy, es. numero carta di credito

- le variabili statiche: sono associate alla classe e non alla specifica istanza dell'oggetto che si sta serializzando
  - lette dalla classe in fase di deserializzazione

- tutti i componenti di un oggetto devono essere serializzabili: se ne esiste uno non serializzabile e non transient si solleva una `notSerializableException`
  - regola #2: per rendere un oggetto persistente occorre marcare tutti i campi che non sono serializzabili come transient

LA SERIALIZZAZIONE: "UNDER THE HOOD"

- la serializzazione è un meccanismo molto complesso e computazionalmente pesante
- La serializzazione standars di JAVA utilizza diversi meccanismi, che è interessante conoscere perchè utili anche nel caso di altri meccanismi di serializzazione
  - caching
  - controllo delle versioni
  - performance

CACHING: UNDER THE HOOD

```
public class BigData {
  private static final long TERA_BYTE =
                        1024L * 1024 * 1024 * 1024;
  public static void main(String[] args) throws IOException
    {
    long bytesWritten = 0;
    byte[ ] data = new byte[100 * 1024 * 1024];
    ObjectOutputStream out = new ObjectOutputStream(
                new BufferedOutputStream(
                        new FileOutputStream("bigdata.bin") ) );
    long time = System.currentTimeMillis();
```

CACHING: UNDER THE HOOD

```
for (int i = 0; i < 10 * 1024 * 1024; i++)
    { out.writeObject(data);
      bytesWritten += data.length; }
out.writeObject(null);
out.close();
time = System.currentTimeMillis() - time;
System.out.printf("Wrote %d TB%n", bytesWritten /
                            TERA_BYTE);
System.out.println("time = " + time);
  } }
```

```
Wrote 1000 TB   Time = 3693
```

Ma la dimensione del file bigdata.bin è solo di 150 M. Come è possibile???

SERIALIZZAZIONE: CACHING

- ogni volta che un oggetto viene serializzato e inviato ad una `ObjectOutputStream`, un suo riferimento viene memorizzato in una "identity hash table"

- se l'oggetto viene scritto nuovamente sull' `OutputStream`, non viene nuovamente serializzato, viene inserito un puntatore all'oggetto precedente
  - minimizzazione e scritture e risoluzione di relazioni circolari tra oggetti

- comportamento analogo, quando si legge da uno stream: l'oggetto letto viene memorizzato in una "identity hash table", la prima volta
  - letture future fanno riferimento allo stesso oggetto

- possibili inconsistenze quando lo stato dell'oggetto viene modificato
  - la modifica viene persa, perchè il riferimento all'oggetto rimane il solito, anche se il suo stato è stato modificato.
  - problema nel caso di invio di uno stream di oggetti su una connessione di rete

IL CONTROLLO DELLE VERSIONI

- per deserializzare un oggetto occorre conoscere
  - i byte che rappresentano l'oggetto serializzato
  - il codice della classe che descrive la specifica dell'oggetto
- la deserializzazione può avvenire in un ambiente diverso, ad esempio
  - mediante l'utilizzo di un compilatore diverso
  - a distanza di tempo rispetto al momento in cui è stata effettuata la serializzazione
  - su un computer diverso, a cui è stato mandato l'oggetto serializzato tramite una connessione di rete
- cosa succede se la classe utilizzata per la serializzazione cambia quando l'oggetto viene deserializzato?

IL CONTROLLO DELLE VERSIONI

```
public class Employee {
    private String id;
    private String name;
    private int age;}
```

- supponiamo di serializzare i dati di un insieme di impiegati utilizzando la classe precedente
- successivamente la classe viene modificata come segue

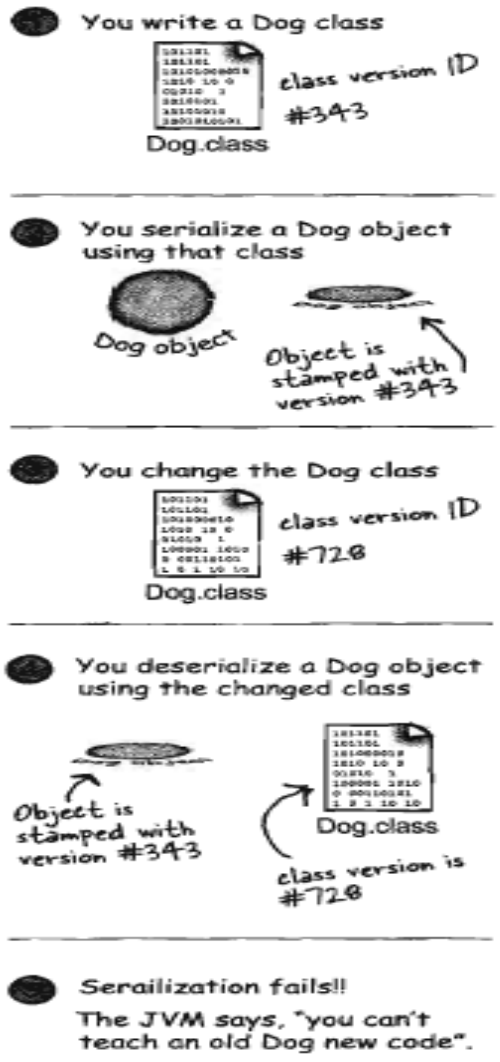
```
public class Employee {
    private String id;
    private String name;
    private Date dateOfBirth; }
```

- si può utilizzare la classe modificata per deserializzare un oggetto che è stato serializzato con la classe prima della modifica?

serialVersionUID (SUID)

- identificatore unico che identifica una classe
- utilizzato per verificare che la compatibilità tra le classi usate per la serializzazione e la deseraliazzazione, in fase di deserializzazione
- può essere gestito in due modi diversi
- identificatore implicito: generato dal compilatore, non specificato dall'utente
  - 64-bit hash (SHA) generato a partire dalla struttura della classe, durante la serializzazione (nome della classe, nomi delle interfacce, metodi e campi)
  - in alcuni casi, compilatori diversi possono generare identificatori diversi per la stessa classe
- identificatore esplicito: il programmatore gestisce esplicitamente la compatibilità delle classi, gestendo i loro identificatori. Ma come si generano gli identificatori?
  - dichiarato a scelta dal programmatore
  - usando l'IDE Eclipse
  - con un generatore a linea di comando

serialVersionUID (SUID) IMPLICITO



CLASSI BACKWARD COMPATIBILI

- la classe usata per la serializzazione può essere modificata, ma essere sempre backward-compatible
  - aggiungere attributi e/o oggetti
  - trasformare attributi transient in non-transient
  - cambiare una variabile di istanza in `static`
  - e molti altri cambiamenti.....
- in fase di deserializzazione, il meccanismo di default semplicemente imposta i valori dei campi mancanti con valori di default
- il programmatore può "fixare" i valori dei campi aggiunti all'oggetto deserializzato
- modifiche che rendono la classe non backward-compatibile
  - rimuovere attributi
  - trasformare attributi `non-transient` in `transient`

serialVersionUID ESPLICITO

- il programmatore può generare esplicitamente `serialVersionUID` per le classi interessate alla serializzazione/deserializzazione
  - classi compatibili: stesso `serialVersionUID` in entrambe le classi
  - classi incompatibili: diverso `serialVersionUID` per la classe modificata

- il supporto:
  - controlla se l'utente ha dichiarato esplicitamente il `serialVersionUID` ed, in questo caso, usa questo valore.
  - altrimenti genera un identificatore implicito

- generazione esplicita di identificatori
  - mediante tool automatici di JAVA, dato il nome della classe restituiscono un identificatore unico della classe

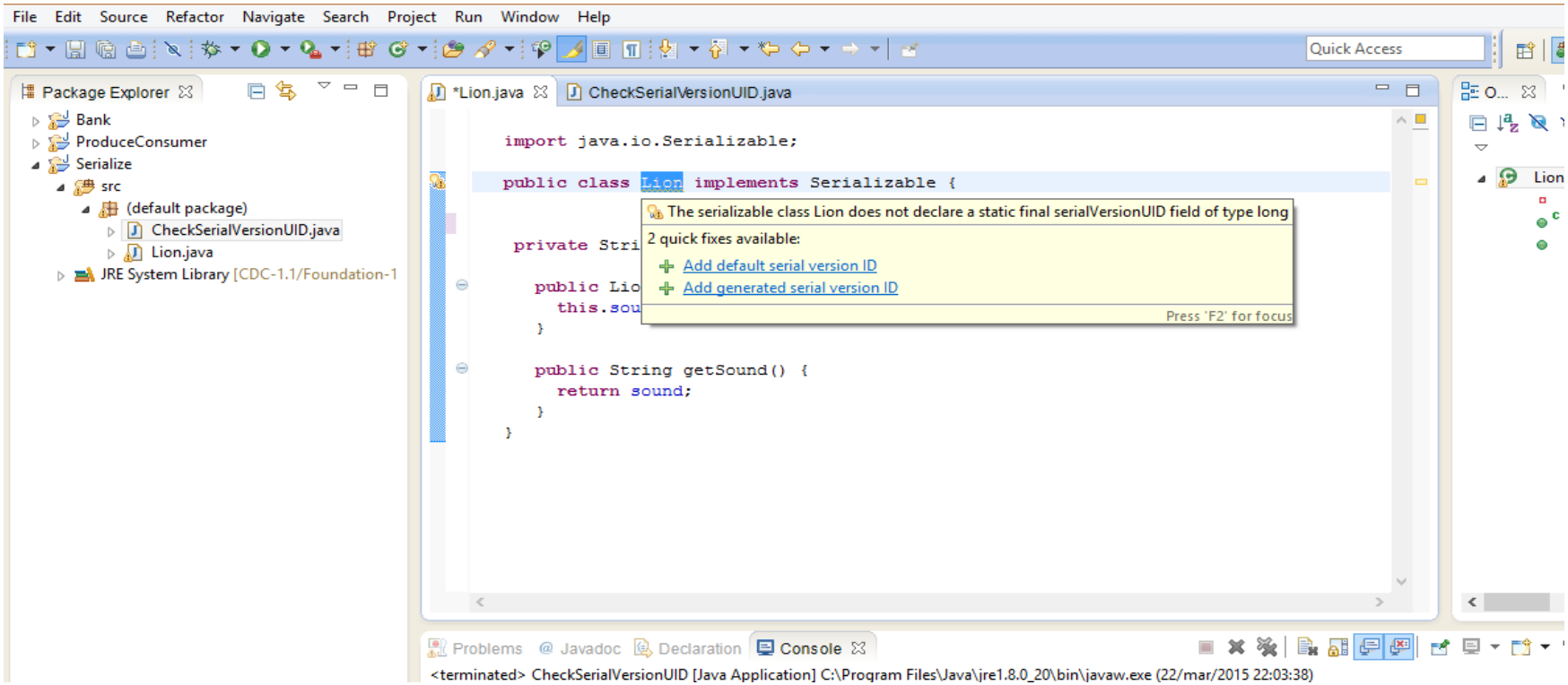
- dichiarazione identificatori espliciti

```
private static final long serialVersionUID = 42L;
```

serialVersionUID ESPLICITO

generazione di `serialVersionUID` unico mediante l'algoritmo utilizzato da JAVA:

- in Eclipse puntare il mouse sul nome di una classe `Serializzabile` (`addGeneratedSerialVersionUID`)
- in altri ambienti: usare tool di generazione specifici (`serialver`)



CONTROLLO DELLE VERSIONI: UN ESEMPIO

```
import java.io.Serializable; import java.util.*;
public class Employee implements Serializable {
   private static final long serialVersionUID = 1L;
   private String name;
   private String address;
   private byte age;
   public String getName() { return name;   }
   public void setName(String name) { this.name = name;}
   public byte getAge() { return this.age;}
   public void setAge(byte age) {  this.age = age;}
   public String getAddress() { return address;}
   public void setAddress(String address) { this.address = address;}
   public String whoIsThis()
   { StringBuffer employee = new StringBuffer();
   employee.append(getName()).append("is").append(getAge()).append("years
                        old and lives at").append(getAddress());
      return employee.toString();    }}
```

CONTROLLO DELLE VERSIONI: UN ESEMPIO

```
public class Writer {
    public static void main(String[] args) throws IOException {
        Employee employee = new Employee();
        employee.setName("Maria");
        employee.setAge((byte) 30);
        employee.setAddress("Via Bianchi");
        FileOutputStream fout = new FileOutputStream("./employee.obj");
        ObjectOutputStream oos = new ObjectOutputStream(fout);
        oos.writeObject(employee);
        oos.close();
        System.out.println("Process complete"); } }
```

CONTROLLO DELLE VERSIONI: UN ESEMPIO

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
public class Reader {
    public static void main(String[] args) throws ClassNotFoundException,
          IOException {
              Employee employee = new Employee();
              FileInputStream fin = new FileInputStream("./employee.obj");
              ObjectInputStream ois = new ObjectInputStream(fin);
              employee = (Employee) ois.readObject();
              ois.close();
              System.out.println(employee.whoIsThis()); }}
```

"GIOCARE" CON L'ESEMPIO

- *prova 1*
  - eseguire prima `Writer`, che serializza l'oggetto con ***`serialVersionUID`*** `= 1L`
  - in seguito eseguire `Reader`, senza modificare la classe `Employee`
  - risultato: serializzazione e deserializzazione corrette, usano la stessa classe, con lo stesso ID, il programma stampa:

```
Maria is 30 years old and lives at Via Bianchi
```

- *prova 2*
  - eseguire prima `Writer`, che serializza l'oggetto con ***`serialVersionUID`*** `= 1L`
  - modificare quindi la classe `Employee`, aggiungendo un campo ed i metodi set e get relativi (vedi slide successiva)
  - in seguito eseguire `Reader`, che farà riferimento alla classe modificata
  - serializzazione e deserializzazione fanno riferimento a due classi diverse, ma con lo stesso ***`serialVersionUID`***
  - il risultato è:

```
Maria is 30 years old and lives at Via Bianchi
```

"GIOCARE" CON L'ESEMPIO

```
import java.io.Serializable; import java.util.*;
public class Employee implements Serializable {
   private static final long serialVersionUID = 1L;
   private String name;
   private String address;
   private byte age;
   private Date dateOfBirth;
   public void setDate(Date date) {this.dateOfBirth = date;}
   public Date getDate() {return this.dateOfBirth;}
   …….
```

- modificare la classe dopo la serializzazione
- un nuovo campo `dateOfBirth`
- i metodo get e set relativi

"GIOCARE" CON L'ESEMPIO

- *prova 3*
  - eseguire prima `Writer`, senza specificare alcun ***serialVersionUID*** in `Employee`
  - modificare quindi la classe `Employee`, aggiungendo un campo e get e set relativi (come nella slide precedente), ma non aggiungere ***serialVersionUID***
  - in seguito eseguire `Reader`, che farà riferimento alla classe modificata
  - serializzazione e deserializzazione fanno riferimento a due classi diverse, e con il ***serialVersionUID implicito e diverso***
  - il risultato è:

```
Exception in thread "main" java.io.InvalidClassException: Employee; local class incompatible: stream classdesc serialVersionUID =
-7901079435486647298, local class serialVersionUID = 7127134952141211549
 at java.io.ObjectStreamClass.initNonProxy(Unknown Source)
 at java.io.ObjectInputStream.readNonProxyDesc(Unknown Source)
 at java.io.ObjectInputStream.readClassDesc(Unknown Source)
 at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
 at java.io.ObjectInputStream.readObject0(Unknown Source)
 at java.io.ObjectInputStream.readObject(Unknown Source)
 at java.io.ObjectInputStream.readObject(Unknown Source)
  at Reader.main(Reader.java:17)
```

"GIOCARE" CON L'ESEMPIO

- *prova 4*
  - eseguire prima `Writer`, specificando ***`serialVersionUID=1L`*** in `Employee`
  - modificare quindi la classe `Employee`, aggiungendo un campo e get e set relativi (come nella slide precedente), e modificare *`serialVersionUID=2L`*
  - in seguito eseguire `Reader`, che farà riferimento alla classe modificata
  - serializzazione e deserializzazione fanno riferimento a due classi diverse, e con un ***`serialVersionUID,`*** specificato esplicitamente dal programmatore e diverso
  - il risultato è:

```
Exception in thread "main" java.io.InvalidClassException: Employee; local class incompatible: stream classdesc serialVersionUID =
-7901079435486647298, local class serialVersionUID = 7127134952141211549
 at java.io.ObjectStreamClass.initNonProxy(Unknown Source)
 at java.io.ObjectInputStream.readNonProxyDesc(Unknown Source)
 at java.io.ObjectInputStream.readClassDesc(Unknown Source)
 at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
 at java.io.ObjectInputStream.readObject0(Unknown Source)
 at java.io.ObjectInputStream.readObject(Unknown Source)
 at java.io.ObjectInputStream.readObject(Unknown Source)
  at Reader.main(Reader.java:17)
```

CONTROLLO VERSIONI

- infine JAVA suggerisce di indicare esplicitamente un `SerialVersionUID`
- da JAVAdoc: "`the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected InvalidClassExceptions during deserialization`"
- spesso si ottiene una eccezione anche se le classe utilizzate in fase di serializzazione e deserializzazione sono in realtà le stesse.
  - compilatori diversi generalo lo stesso serialVersionUID per la stessa classe
- per questo è consigliato di specificare comunque esplicitamente un `serialVersionUID`

SERIALIZZAZIONE JAVA: SVANTAGGI

La serializzazione registra un descrittore dell'oggetto di dimensione significativa

- i "magic data"
  - `STREAM_MAGIC` = "acde"
  - `STREAM_VERSION` = versione della JVM
- i metadati che descrivono la classe associata all'istanza dell'oggetto serializzato
  - la descrizione include il nome della classe, il `serialVersionUID` della classe, il numero di campi, altri flag.
- i metadati di eventuali superclassi, fino a raggiungere `java.lang.Object`
- i valori associati all'oggetto istanza della classe, partendo dalla super classe a più alto livello
- i dati degli oggetti eventualmente riferiti dall'oggetto istanza della classe, iniziando dai metadati e poi registrando i valori. (Le istanze della classe Figlio, nell'esempio precedente).
- non si registrano i metodi della classe

SERIALIZZAZIONE JAVA: SVANTAGGI

```
Length: 220
Magic: ACED
Version: 5
  OBJECT
    CLASSDESC
    Class Name: "SimpleClass"
    Class UID:  -D56EDC726B866EBL
    Class Desc Flags: SERIALIZABLE;
    Field Count: 4
    Field type: object
    Field name: "firstName"
    Class name: "Ljava/lang/String;"
    Field type: object
    Field name: "lastName"
    Class name: "Ljava/lang/String;"
        Field type: float
    Field name: "weight"
    Field type: object
    Field name: "location"
    Class name: "Ljava/awt/Point;"
    Annotation: ENDBLOCKDATA
    Superclass description: NULL
   STRING: "Brad"
   STRING: "Pitt"
   float: 180.5
   OBJECT
      CLASSDESC
      Class Name: "java.awt.Point"
      Class UID:  -654B758DCB8137DAL
      Class Desc Flags: SERIALIZABLE;
      Field Count: 2
      Field type: integer
      Field name: "x"
      Field type: integer
      Field name: "y"
      Annotation: ENDBLOCKDATA
      Superclass description: NULL
     integer: 49.345
     integer: 67.567
```

- la classe `SimpleClass` ha i campi
  - `firstName`,
  - `lastName`,
  - `weight`
  - `Location`
- l'oggetto istanza della classe contiene i campi

  `{"Brad","Pitt",180.5, {49.345, 67.567}}`
- a fianco: risultato della serializzazione
- per la memorizzazione di un oggetto di 20 bytes utilizzati circa 220 bytes!
  - molto costoso in termini di spazio utilizzato
  - alto consumo di banda, se l'oggetto deve essere spedito

SERIALIZZAZIONE JAVA: SVANTAGGI

- oggetto serializzato aumenta molto di dimensione rispetto all'oggetto originario
- limitata interoperabilità
  - utilizzabile solo quando sia l'applicazione che serializza l'oggetto che quella che lo deserializza sono scritte in JAVA
- per aumentare l'interoperabilità occorre utilizzare altre soluzioni:
  - il formato standard JSON (presentato in una prossima lezione...)
  - serializzazione in XML
  - ...altri formati....

ESERCIZIO: FILE CRAWLER

- si scriva un programma JAVA che
  - riceve in input un *filepath* che individua una directory D
  - stampa le informazioni del contenuto di quella directory e, ricorsivamente, di tutti i file contenuti nelle sottodirectory di D
- il programma deve essere strutturato come segue:
  - attiva un thread produttore ed un insieme di k thread consumatori
  - il produttore comunica con i consumatori mediante una coda
  - il produttore visita ricorsivamente la directory data ed, eventualmente tutte le sottodirectory e mette nella coda il nome di ogni directory individuata
  - i consumatori prelevano dalla coda i nomi delle directories e stampano il loro contenuto
  - la coda deve essere realizzata con una LinkedList. Ricordiamo che una Linked List non è una struttura thread-safe. Dalle API JAVA "*Note that the implementation is not synchronized. If multiple threads access a linked list concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally*"