Input/Output

Prof. Michele Loreti

Programmazione Avanzata

Corso di Laurea in Informatica (L31)

Scuola di Scienze e Tecnologie

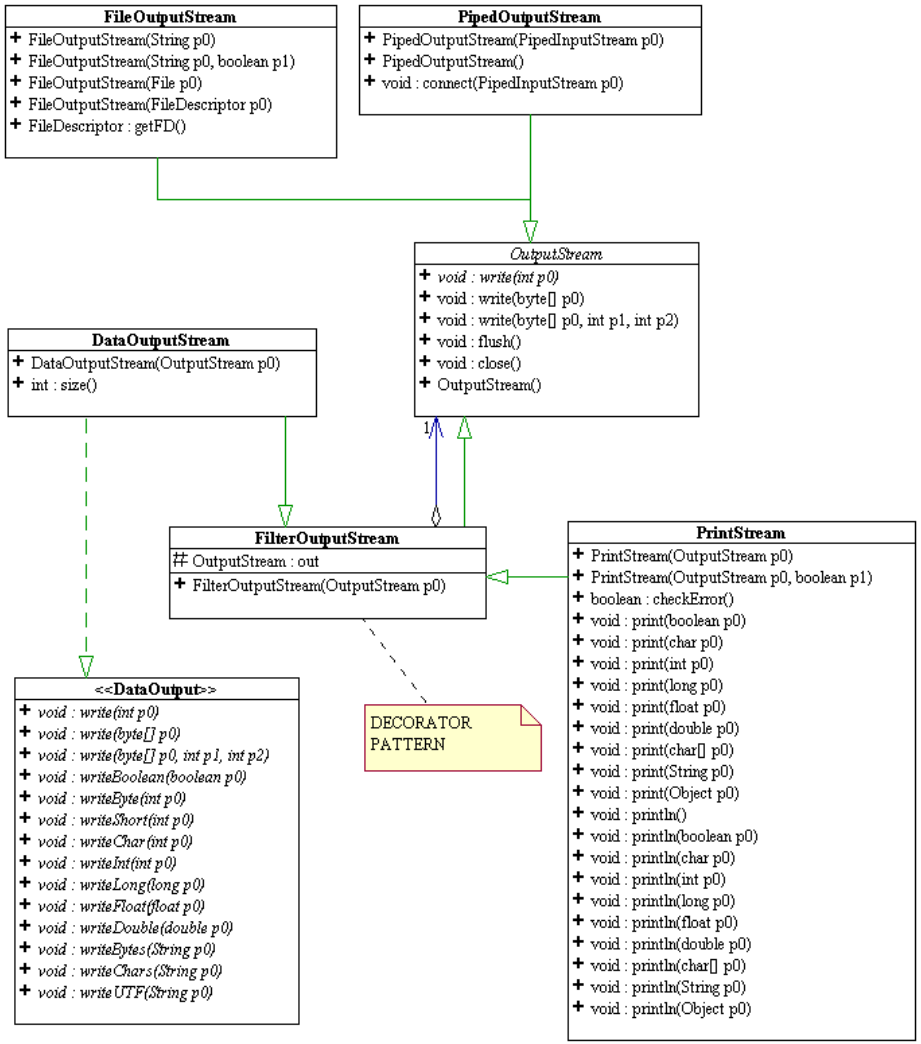
Lo Stream...

- Lo *Stream* rappresenta l'astrazione che è alla base delle comunicazioni in `Java`;
- Rappresenta un'estremo di un canale di comunicazione;
- Normalmente un canale di comunicazione collega un `OutputStream` con un `InputStream`;
- Il collegamento può realizzarsi per mezzo di molteplici supporti: rete, file, console, ...;
- Lo stream fornisce un'interfaccia generale per la gestione dei dati, senza prendere in considerazione un particolare mezzo di comunicazione.

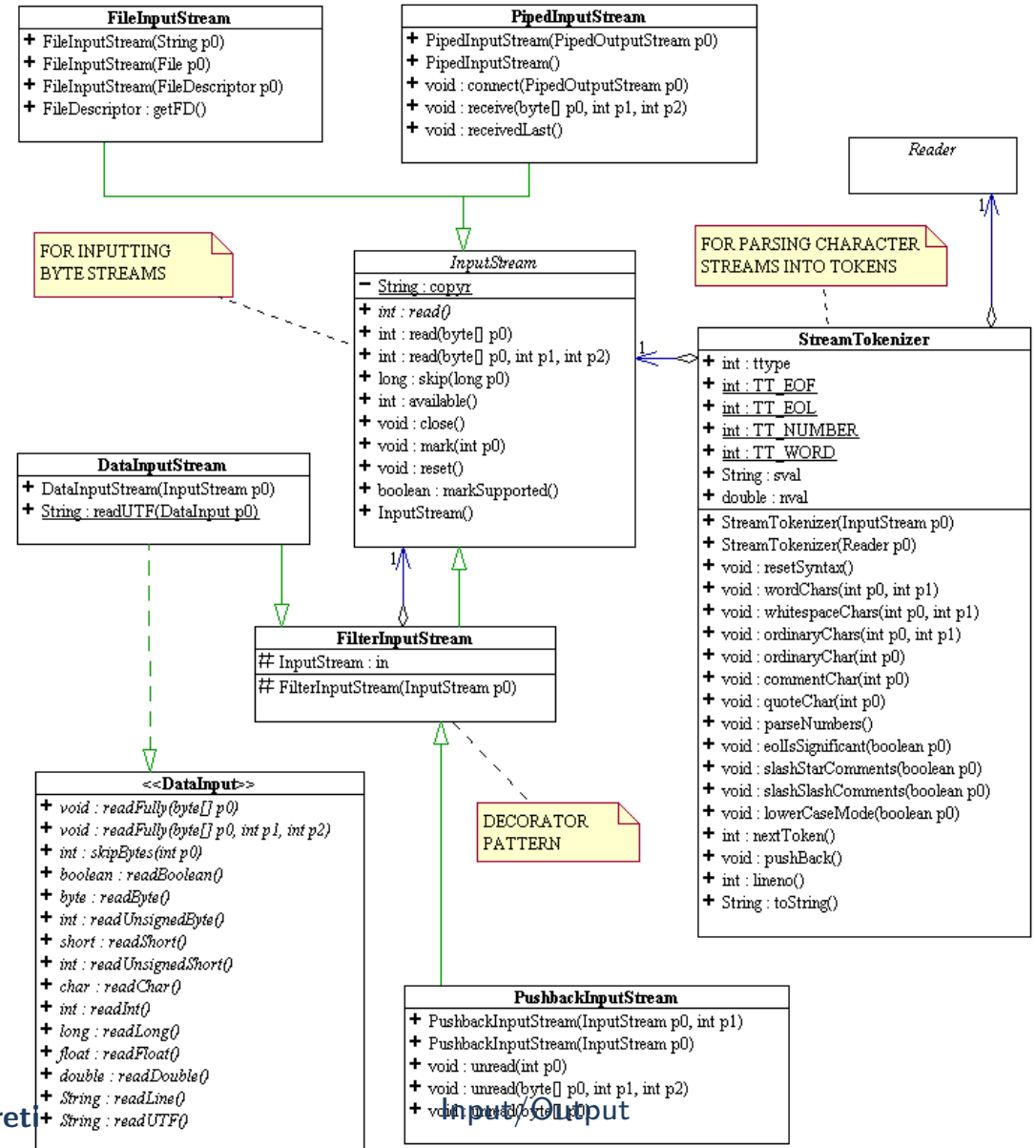
Lo Stream...

- Tipologia dell'accesso:
  - FIFO (First In First Out);
  - Sequenziale.
- Consentono o la sola lettura o la sola scrittura;
- Le operazioni di lettura/scrittura sono, in generale, bloccanti;
- Java fornisce classi particolari per la gestione degli Stream di caratteri.

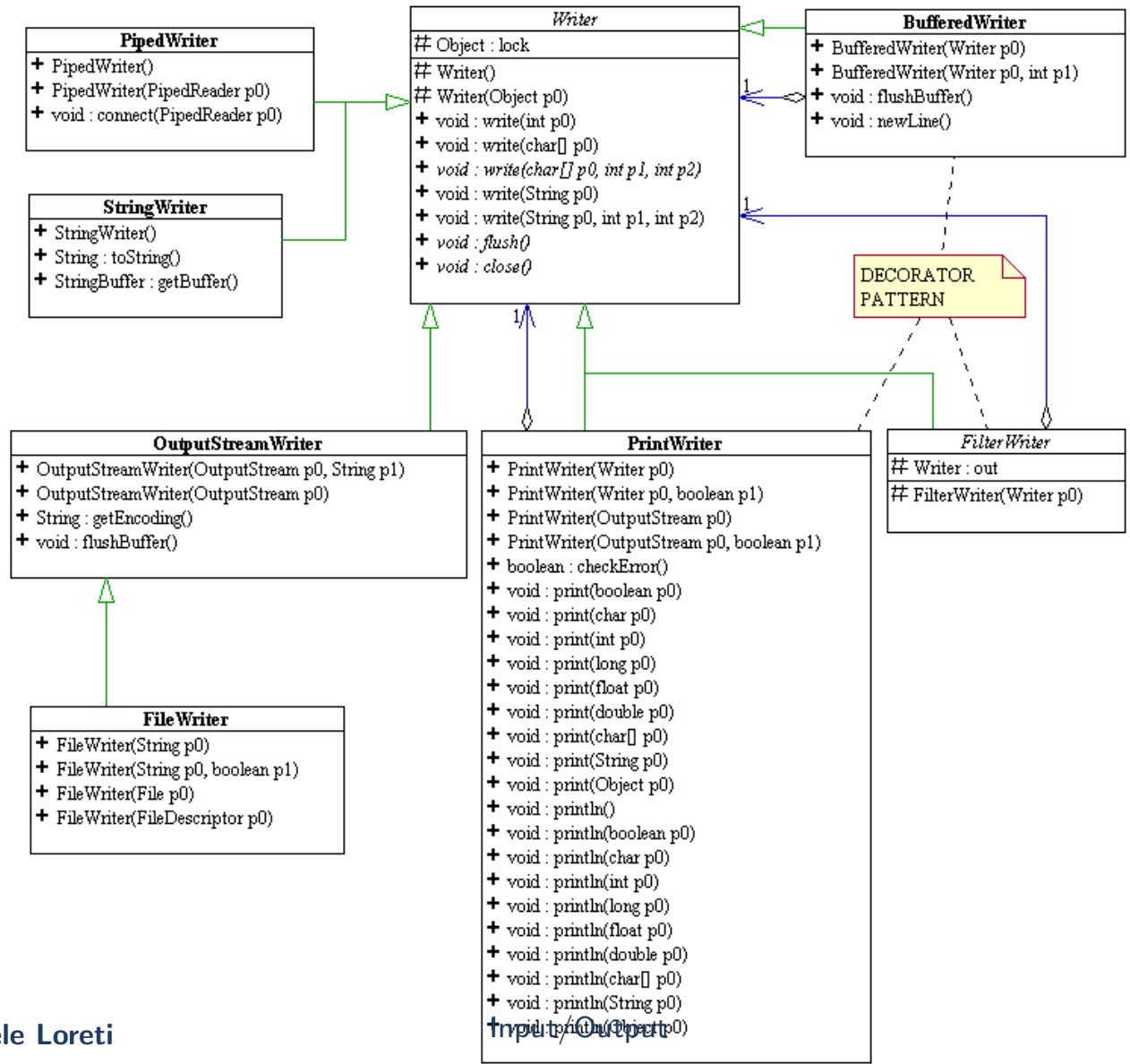
Overview delle classi...



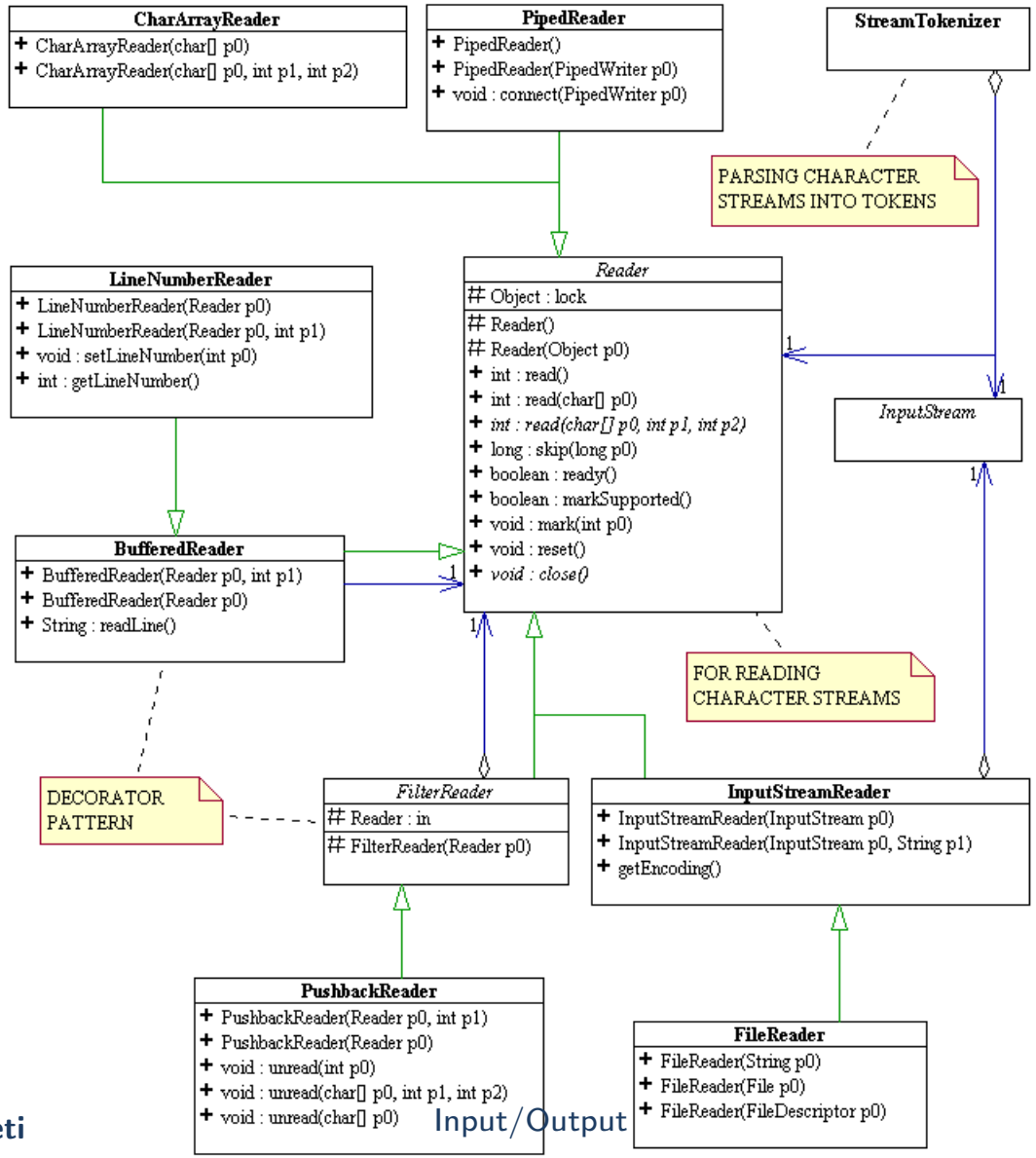
Overview delle classi...



Overview delle classi...



Overview delle classi...



OutputStream...

- La classe `OutputStream` è una classe astratta;
- Non si possono istanziare, direttamente, oggetti della classe `OutputStream`;
- Si possono costruire oggetti di una delle sue sottoclassi:
  - `FileOutputStream`, scrittura su file;
  - `PipedOutputStream`, scrittura su buffer in memoria.
- Oggetti della classe `OutputStream` sono il risultato dell'invocazione di alcuni metodi (`getOutputStream()`).

OutputStream: i metodi...

- `abstract void write(int b) throws IOException`
  - Scrive gli 8 bit meno significativi dell'intero `b`;
- `void write(byte[] b, int off, int len) throws IOException`
  - Scrive `len` byte di `b` a partire dalla posizione `off`;
- `void write(byte[] b) throws IOException`
  - Scrive i byte contenuti nell'array `b`;
- `void flush() throws IOException`
  - Svuota l'evenutal buffer di byte in memoria;
- `void close() throws IOException`
  - Chiude il canale di comunicazione sottostante liberando le risorse di sistema.

Esempio...

```
import java.io.*;
public class SimpleOut {
  public static void main(String[] args)
    throws IOException {
    for (int i=0;i<args.length;i++) {
       println( args[i] );
    }
  }
  public static void println( String m )
    throws IOEXception {
    for( int i=0 ; i<m.length() ;i++) {
      System.out.write(m.charAt( i ) & 0xff);
    }
    System.out.write('\n');
    System.out.flush();
  }
}
```

InputStream...

- La classe `InputStream` è una classe astratta;
- Non si possono istanziare, direttamente, oggetti della classe `InputStream`;
- Si possono costruire oggetti di una delle sue sottoclassi:
  - `FileInputStream`, scrittura su file;
  - `PipedInputStream`, scrittura su buffer in memoria.
- Oggetti della classe `InputStream` sono il risultato dell'invocazione di alcuni metodi (`getInputStream()`).

InputStream: i metodi...

- `abstract int read() throws IOException`
  - Legge un byte dallo stream ritornando `-1` se si è giunti alla fine del file;
- `int read(byte[] b, int off, int len) throws IOException`:
  - Legge (al più) `len` byte memorizzandoli nell'array `b` a partire dalla posizione `off`, restituisce il numero di byte *effettivamente* letti;
- `int read(byte[] b) throws IOException`:
  - Legge (al più) `b.lenght` byte memorizzandoli nell'array `b`, restituisce il numero di byte letti;

`InputStream`: i metodi...

- `int available() throws IOException`
  - Legge il numero di byte che possono *effettivamente* letti dallo strem;
- `void close() throws IOException`:
  - Chiude il canale di comunicazione sottostante all'`InputStream`;
- `long skip(long n) throws IOException`:
  - Tenta di ignorare `n` byte presenti sullo stream, ritorna il numero di byte *effettivamente* ignorati;

`InputStream`: mark e reset...

- I metodi `mark()` e `reset()` consentono di:
  - segnare un punto dello stream;
  - effettuare delle letture;
  - ritornare al punto segnato.

InputStream: mark e reset...

- `boolean markSupported()`
  - Verifica se il particolare `InputStream` supporta il mark/reset (nessuna delle classi base di `InputStream` supporta il mark/reset);
- `void mark(int readlimit)`
  - Fissa il mark dichiarando il massimo numero di byte che verranno letti prima di invocare un `reset()`;
- `void reset() throw IOException`
  - Riposiziona lo stream alla posizione attiva al momento dell'ultima invocazione di `mark`.

Esempio...

```
import java.io.*;
public class SimpleIn {
  public static void main(String[] args)
    throws IOException {
    int charRead;
    while ((charRead = System.in.read ()) >= 0) {
       System.out.write (charRead);
    }
  }
}
```

Esempio...

```
import java.io.*;
public class SimpleIn {
  public static void main(String[] args)
    throws IOException {
    int numberRead;
    byte[] buffer = new byte[8];
    while ((numberRead =
          System.in.read (buffer)) >= 0) {
       System.out.write (buffer,0,numberRead);
    }
  }
}
```

Alcuni *stream* di base...

- Accesso ai File:
  - `FileOutputStream`;
  - `FileInputStream`.
- Accesso ad Array di Byte:
  - `ByteArrayOutputStream`;
  - `ByteArrayInputStream`.
- Accesso ad area di memoria (*pipe*):
  - `PipedOutputStream`;
  - `PipedInputStream`.

La classe `File`

- La classe `File` rappresente un nome di file che sia *indipendente* dal sistema;
- Fornisce tre costruttori:
  - `File(String path);`
  - `File(String path, String name);`
  - `File(File dir, String name).`
- Variabili statiche per la gestione dei separatori nel nome del file e nel path di sistema;

La classe `File`

- Metodi per accedere allo stato del file:
  - `boolean exists();`
  - `boolean canRead();`
  - `long length();`
  - `...`
- Metodi statici:
  - `File[] listRoots();`
  - `File createTempFile(String prefix, String suffix, File directory);`
  - `File createTempFile(String prefix, String suffix).`

Altre classi...

- `FileDescriptor`
  - Fornisce le primitive per accedere ai struttura *file-descriptor* di un file;
  - Si interfaccia direttamente con il sistema operativo.
- `RandomAccessFile`
  - Fornisce un modo alternativo alla gestione dei file ovviando all'uso di `FileInputStream` e `FileOutputStream`;
  - Permette la contemporanea lettura e scrittura di un file;
  - Consente un accesso *random* al file.

`FileOutputStream`...

- Fornisce le primitive per la scrittura di dati su di un file;
- Costruttori:
  - `FileOutputStream(String name) throws IOException`
    - Crea un file chiamato `name` distruggendo ogni file con lo stesso nome;
  - `FileOutputStream(File file) throws IOException`
    - Crea il file corrispondente all'oggetto `file` distruggendo ogni file con lo stesso nome;
  - `FileOutputStream(String name, boolean append) throws IOException`
    - Crea un file chiamato `name` distruggendo ogni file, `append` indica se troncare (`fale`) o appendere i nuovi dati al file (`true`)

FileOutputStream...

- Fornisce un solo metodo aggiuntivo rispetto alla classe `OutputStream`:
  - `FileDescriptor getFD() throws IOException`
    - Ritorna l'oggetto `FileDescriptor` associato al file sul quale si sta scrivendo;
- I costruttori del `FileOutputStream` possono sollevare una `SecurityException`.

`FileInputStream`...

- Fornisce le primitive per la lettura di dati su da un file;
- Costruttori (`throws IOEXception`):
  - `FileInputStream(String name)`
    - Apre un file chiamato `name` per la lettura;
  - `FileInputStream(File file)`
    - Apre il file corrispondente all'oggetto `file` per la lettura;
  - `FileInputStream(FileDescriptor fdObj)`
    - Viene creato un `FileInputStream` associato al file-descriptor `fdObj` che, ovviamente, deve essere un file-descriptor valido.
- Fornisce i metodi standard di `InputStream`:
  - Non implementa le funzionalità mark-reset.

Esempio...

```
public class Copy {
 public static void main(String[] argv)
  throws IOException {
  if (args.length != 2) {
   throw
    new IllegalArgumentException(
        "Syntax: Copy <src> <dst>");
  }
  ...
```

Esempio...

```
    FileInputStream in =
        new FileInputStream(argv[0]);
    FileOutputStream out =
        new FileOutputStream(argv[1]);
    byte[] buffer = new byte[16];
    int numberRead;
    while ((numberRead = in.Read( buffer ))>= 0) {
     out.write (buffer,0,numberRead);
    }
    in.close();out.close();
    }
}
```

Comunicazioni ad alto livello...

- Inserire e rimuovere array di byte da uno stream può essere penalizzante;
- Sarebbe desiderabile avere primitive ad-hoc per la scrittura/lettura di tipi di dato d'alto livello;
- Queste primitive sono facilmente implementabili.

Esempio: `writeInt`...

```
void writeInt(OutputStream out, int value)
  throws IOException {
  out.write(value >> 24);
  out.write(value >> 16);
  out.write(value >> 8);
  out.write(value);
}
```

Esempio: readInt...

```
int readInt(InputStream out)
  throws IOException {
  int v0,v1,v2,v3;
  if (((v3 = in.read())==-1)||
      ((v2 = in.read())==-1)||
      ((v1 = in.read())==-1)||
      ((v0 = in.read())==-1)) {
    throw new IOException("EOF while reading int");
  }
  return (v3 << 24) | (v2 <<16) | (v1 << 8 ) | v0;
}
```

I filtri di Stream...

- Il modo migliore per aggiungere funzionalità agli stream è quello di utilizzare i filtri di stream;
- I filtri di stream applicano il pattern della *delegation*;
- Il pacchetto fornisce due filtri base:
  - `FilterOutputStream` e
  - `FilterInputStream`.
- Le classi hanno, rispettivamente:
  - la stessa interfaccia di `OutputStream` e `InputStream`;
  - un campo `protected` di tipo `OutputStream` e `InputStream`;
  - l'invocazione dei metodi ereditati da `OutputStream` e `InputStream` viene *rimbalzata* al campo protected.
- Le nuove funzionalità vengono aggiunte dalle classi derivate.

Alcuni filtri...

- `BufferedOutputStream` e `BufferedInputStream`
  - Versioni ottimizzate di `InputStream` e `OutputStream`;
- `DataOutputStream` e `DataInputStream`
  - Forniscono il supporto alla scrittura/lettura di dati d'alto livello (interi, booleani,...);
- `PushBackInputStream`
  - Consente di rimarcare alcuni byte come non letti reinserendoli, di fatto, nello stream.
- `SequenceInputStream`
  - Consente di accedere sequenzialmente ai dati contenuti da una serie di `InputStream`.
- Altre classi deprecate:
  - `LineNumberInputStream`;
  - `PrintStream`.

Gli stream di caratteri...

- Lo scambio di informazione testuale per mezzo degli stream è in generale limitante;
- La comunicazione si basa sullo scambio di caratteri a 8-bit (ASCII);
- Allo scopo `Java` fornisce degli stream appropriati per la comunicazione di caratteri a 16-bit:
  - `Writer`;
  - `Reader`.
- Vengono, inoltre, fornite delle classi di collegamento con gli stream standard:
  - `OutputStreamWriter`;
  - `InputStreamReader`.

Gli stream di caratteri...

- Specializzazioni:
  - `FileWriter`;
  - `FileReader`.
- Filtri per gli stream di caratteri:
  - `FilterWriter` e `FilterReader`;
  - `BufferedWriter` e `BufferedReader`;
  - `LineNumberReader`;
  - `PrintWriter`;
  - `PushbackReader`.

La codifica dei caratteri...

- Esistono diversi standard per la codifica dei caratteri;
- Ad esempio:

codifica	carattere	byte
US-ASCII	!	33
IBM-EBCDIC	!	90
ISO Latin	é	232
ISO Latin 2	č	232
UTF-8	é	195 168

- Java supporta diverse codifiche di caratteri:
  - `latin1`, ..., `latin5`;
  - `cyrillic`, `arabic`, ...;
  - `Unicode`, `UnicodeBig`, ...;
  - `ASCII`, `UTF8`.

Writer...

- Costruttori:
  - `protected Writer();`
  - `protected Writer(Object lock).`
- Campi:
  - `protected Object lock.`

Writer: i metodi...

- `void write(int c) throws IOException;`
- `void write(char[] buff) throws IOException;`
- `abstract void write(char[] buff, int off, int len);`
- `void write(String s) throws IOException;`
- `void write(String s, int off, int len) throws IOException;`
- `abstract void flush() throws IOException;`
- `abstract void close() throws IOException;`

Reader...

- Costruttori:
  - `protected Reader();`
  - `protected Reader(Object lock).`
- Campi:
  - `protected Object lock.`

Reader: i metodi...

- `int read() throws IOException;`
- `int read(char[] buff) throws IOException;`
- `abstract int read(char[] buff, int off, int len);`
- `long skip(long n) throws IOException;`
- `boolean ready() throws IOException;`
- `abstract void close() throws IOException;`
- `boolean markSupported() throws IOException;`
- `void mark(int readAheadLimit) throws IOException;`
- `void reset() throws IOException;`

Le classi *ponte*...

- `OutputStreamWriter`:
  - `OutputStreamWriter(OutputStream out);`
  - `OutputStreamWriter(OutputStream out, String enc) throws UnsupportedException;`
  - `String getEncoding().`
- `InputStreamReader`:
  - `InputStreamReader(InputStream out);`
  - `InputStreamReader(InputStream out, String enc) throws UnsupportedException;`
  - `String getEncoding().`

Esempio...

```
import java.io.*;
public class Convert {
  public static void main(String[] args)
    throws IOException {
    if (args.length != 4) {
      throw new IllegalArgumentException(
        "Convert <srcEnc> <source> <dstEnc> <dest>"
      );
    }
    FileInputStream fileIn =
        new FileInputStream( args[1] );
    FileOutputStream fileOut =
        new FileOutpurStream( args[2] );
    ...
```

Esempio...

```
    ...
    InputStreamReader iSR =
        new InputStreamReader( fileIn );
    OutputStreamWriter oSW =
        new OutputStreamWriter( fileOut );
    char[] buffer = new char[16];
    int numberRead;
    while (
      (numberRead = iSR.read( buffer )) > -1 ) {
      outputStreamWriter.write(
          buffer , 0, numberRead);
    }
    oSW.close();
    iSR.close();
  }
}
```

Object Stream...

- Alla base della comunicazione di rete c'è la necessità di scambiare informazioni;
- Lo scambio di informazione è schematizzabile in tre fasi:
  - *Marshalling*: i dati vengono trasformati in sequenze di byte;
  - *Delivery*: è la fase in cui la sequenza di byte viene inviata dal mittente al destinatario;
  - *Unmarshalling*: la sequenza di byte viene trasformata in informazione strutturata.
- Il marshalling/unmarshalling dei tipi base consiste in una semplice oeprazione di codifica;
- Il marshalling/unmarshalling di oggetti, i quali possono contenere riferimenti (a volte incrociati) ad altri oggetti, risulta, invece, un'operazione delicata e non banale.

Object Stream...

- In `Java` le operazioni di marshalling e unmarshalling vengono effettuate per mezzo degli object stream:
  - `ObjectOutputStream`;
  - `ObjectInputStream`.
- Il delivery viene effettuato per mezzo di uno stream standard.
- Sono dei filtri anche se non estendono `FilterOutputStream` e `FilterInputStream`.

ObjectOutputStream...

- Implementa l'interfaccia `ObjectOutput` (che estende `DataOutput`);
- Costruttori:
  - `ObjectOutputStream(OutputStream out) throws IOException`;
  - `protected ObjectOutputStream() throws IOException`.
- Metodi:
  - `void writeObject(Object o) throws IOException`;
    - L'oggetto `o` deve implementare l'interfaccia (vuota) `serializable`.
  - `...`

ObjectInputStream...

- Implementa l'interfaccia `InputOutput` (che estende `DataInput`);
- Costruttori:
  - `ObjectInputStream(InputStream in) throws IOException;`
  - `protected ObjectInputStream() throws IOException, SecurityException.`
- Metodi:
  - `Object readObject() throws IOException, ClassNotFoundException;`

Esempio...

```
interface figura extends Serializable {
  int getArea();
  int getPerimetro();
}

public void writeFigure(OutputStream out, figure f)
    throws IOException {
  ObjectOutputStream oos
        = new ObjectOutputStream(out);
  oos.writeObject(f);
}
```

Esempio...

```
public figure loadFigure(InputStream is)
    throws IOException,
           ClassNotFoundException,
           ClassCastException {

  ObjectInputStream ois = new ObjectInputStream(is);
  Object o = ois.readObject();

  if (o instanceof figure) {
    return (figure) o;
  }
  throw
   new ClassCastException("Read "+
                          o.getClass().getName()+
                          " instead of figure");
}
```

To be continued...