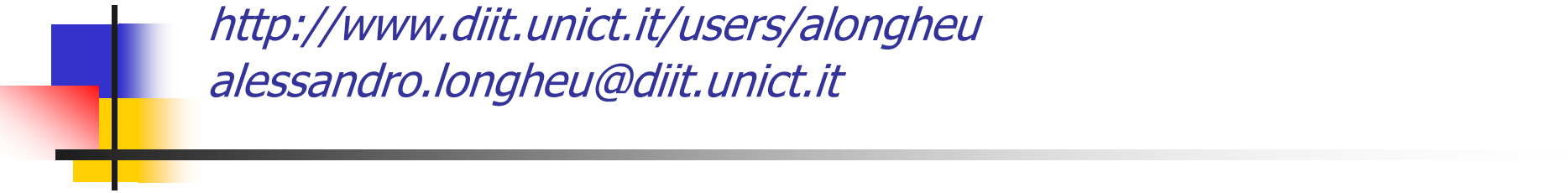
Linguaggi

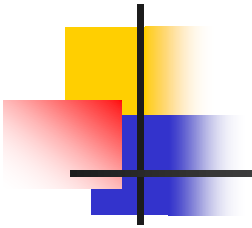
Corso di Laurea in Ingegneria delle Telecomunicazioni
A.A. 2010-2011

Alessandro Longheu
http://www.diit.unict.it/users/alongheu
alessandro.longheu@diit.unict.it



- lezione 09 -

I/O in Java

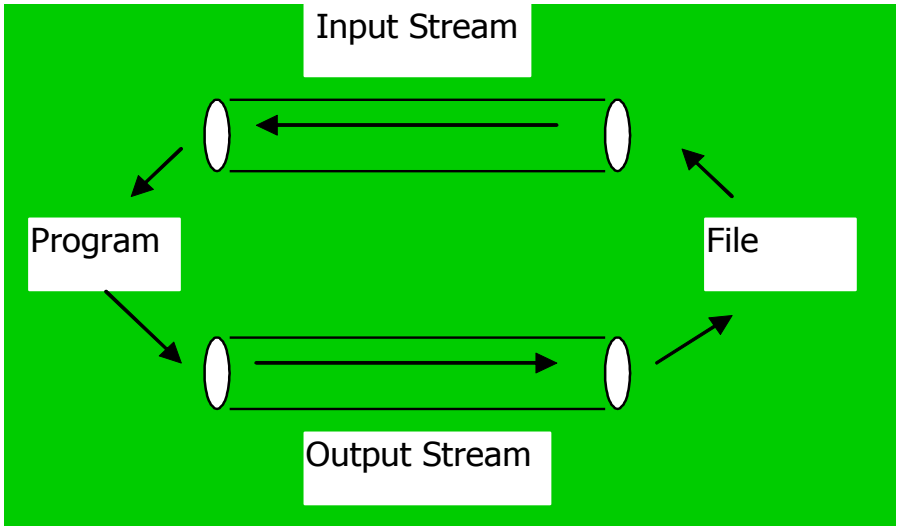


Il package java.io

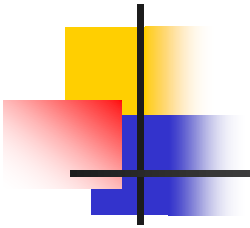
- Il package java.io definisce i concetti base per gestire l'I/O da qualsiasi sorgente e verso qualsiasi destinazione.
- L'obiettivo è fornire
  - un'astrazione che incapsuli tutti i dettagli di una sorgente dati o di un dispositivo di output
  - un modo semplice e flessibile per aggiungere ulteriori funzionalità a quelle fornite dal canale "base"

Stream vs Buffer

- Uno stream o flusso è una sequenza ordinata di dati:
  - monodirezionale (o di input, o di output)
  - adatto a trasferire byte (o anche caratteri)

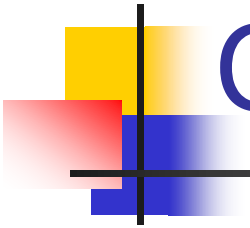


- Buffer: deposito di dati da (su) cui si può leggere (scrivere)
- Canale: connessione con entità in grado di eseguire operazioni di I/O; i canali includono i buffer, i file e i socket



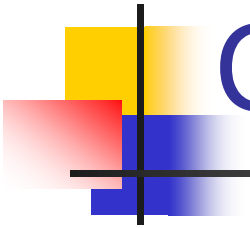
Stream vs Buffer

- I flussi permettono operazioni di I/O bloccanti. Il relativo package in Java è il più vecchio *java.io*
- I canali permettono operazioni di I/O bloccanti ma anche operazioni non bloccanti. Il relativo package è *java.nio*, dove la "n" sta per new ma anche per non bloccante.
- Esiste anche *java.net*, che consente l'utilizzo dei socket ed è basato su flussi o canali



Classi stream

- Il package java.io distingue fra:
  - stream di byte (analoghi ai file binari del C)
  - stream di caratteri (analoghi ai file di testo del C, ma con Unicode a 16 bit)
- Questi concetti si traducono in altrettante classi base astratte:
  - stream di byte: InputStream e OutputStream
  - stream di caratteri: Reader e Writer
- Le classi InputStream/OutputStream sono la radice di tutte le classi di byte e le classi Reader/Writer sono la radice di tutte le classi basate sui flussi di caratteri
- Per utilizzare una qualsiasi classe di I/O occorre effettuare l'import di java.io.* (o di una specifica classe)

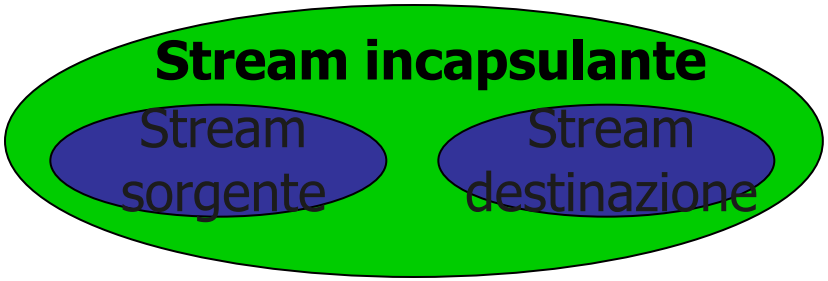


Classi stream

- Le classi e le interfacce di java.io si possono dividere in 5 categorie:
  - **Classi generali** che permettono di costruire tipi differenti di flussi di byte o di caratteri: flussi di input e di output, di byte e caratteri, e le classi per convertire gli uni negli altri
  - Un insieme di classi che definiscono **varianti** di tipi di flussi: filtrati, bufferizzati, piped
  - Un insieme di classi e interfacce riferiti a flussi di dati che permettono di leggere e scrivere **valori primitivi**
  - Le classi e le interfacce che consentono di **interagire con i file** indipendentemente dal SO
  - Le classi e le interfacce che consentono la **serializzazione** degli oggetti: cioè che permettono di trasformare gli oggetti in flussi di dati o viceversa

Classi stream

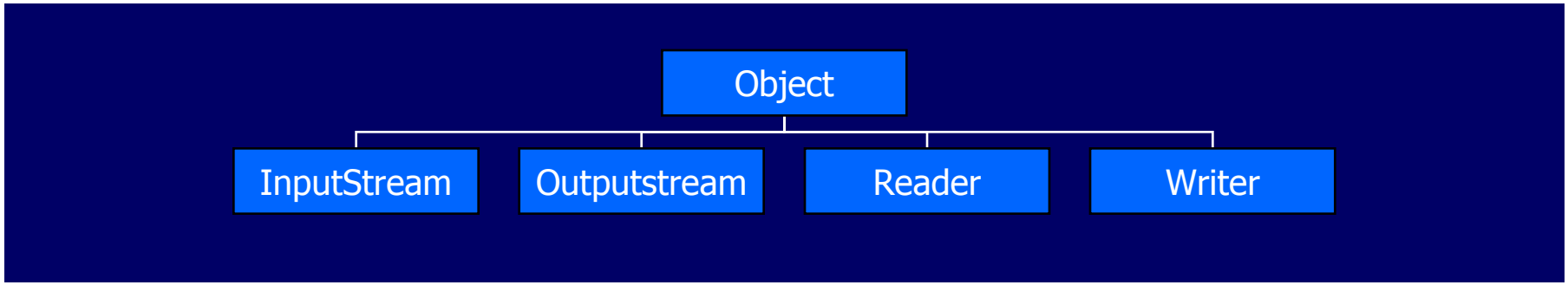
- L'approccio degli stream è denominato "a cipolla" :
  - alcuni tipi di stream rappresentano sorgenti di dati o dispositivi di uscita
    - file, connessioni di rete,
    - array di byte, ...



- gli altri tipi di stream sono pensati per "avvolgere" i precedenti per aggiungere ulteriori funzionalità.
- Così è possibile configurare il canale di comunicazione con tutte e sole le funzionalità che servono senza doverle replicare e reimplementare più volte.

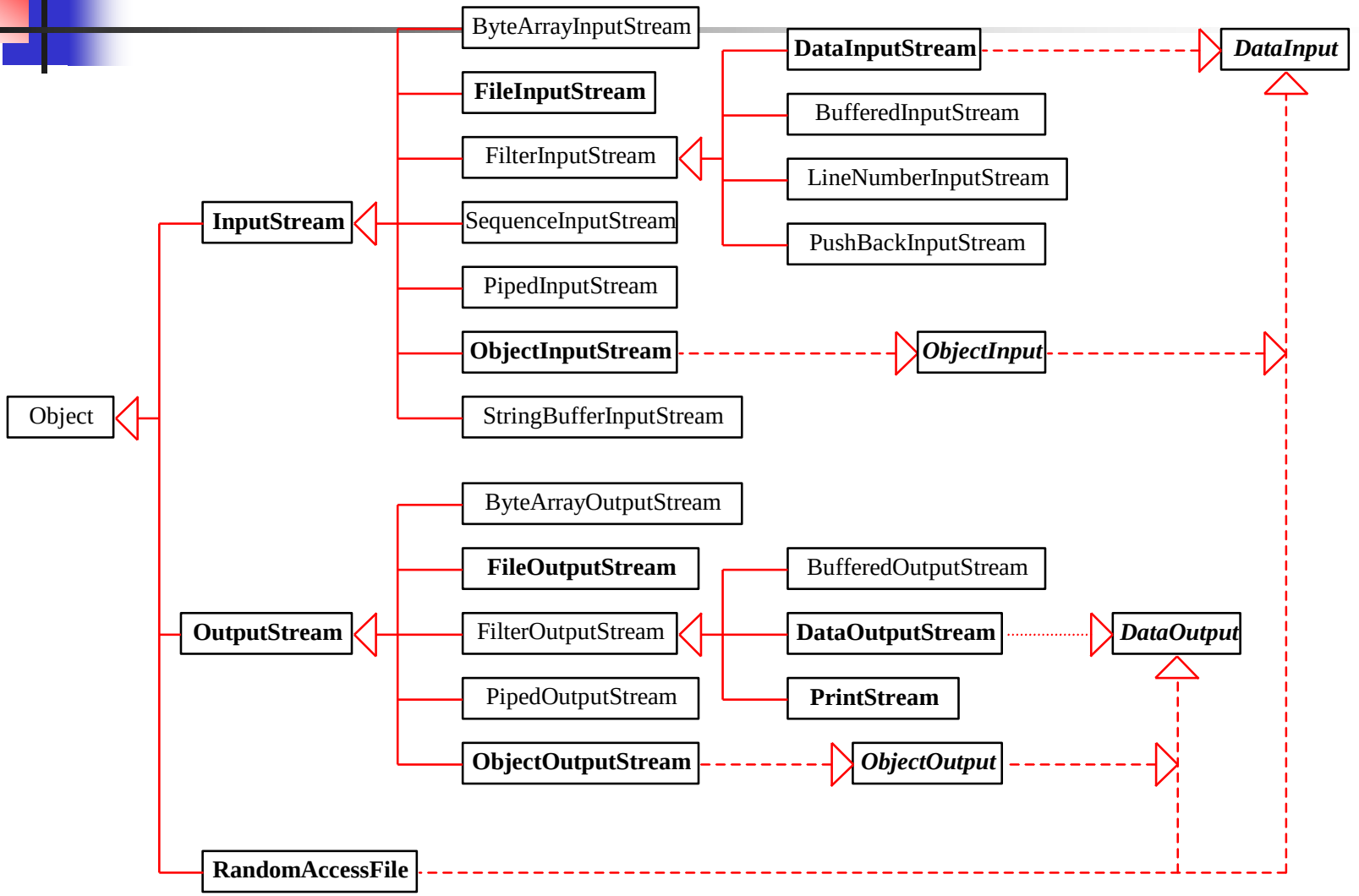
Classi stream

- Le quattro classi base astratte di java.io

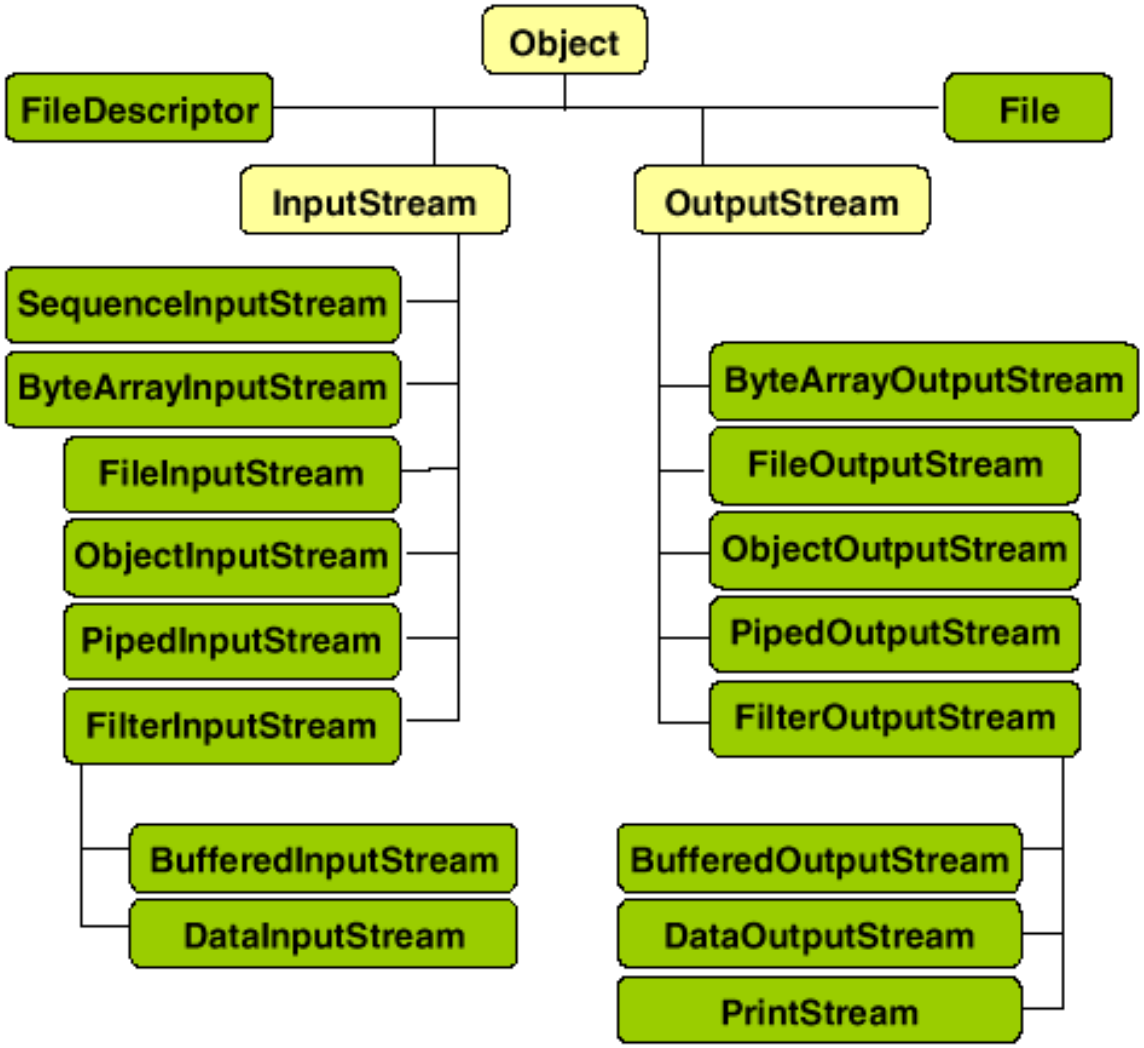


stream di byte stream di caratteri

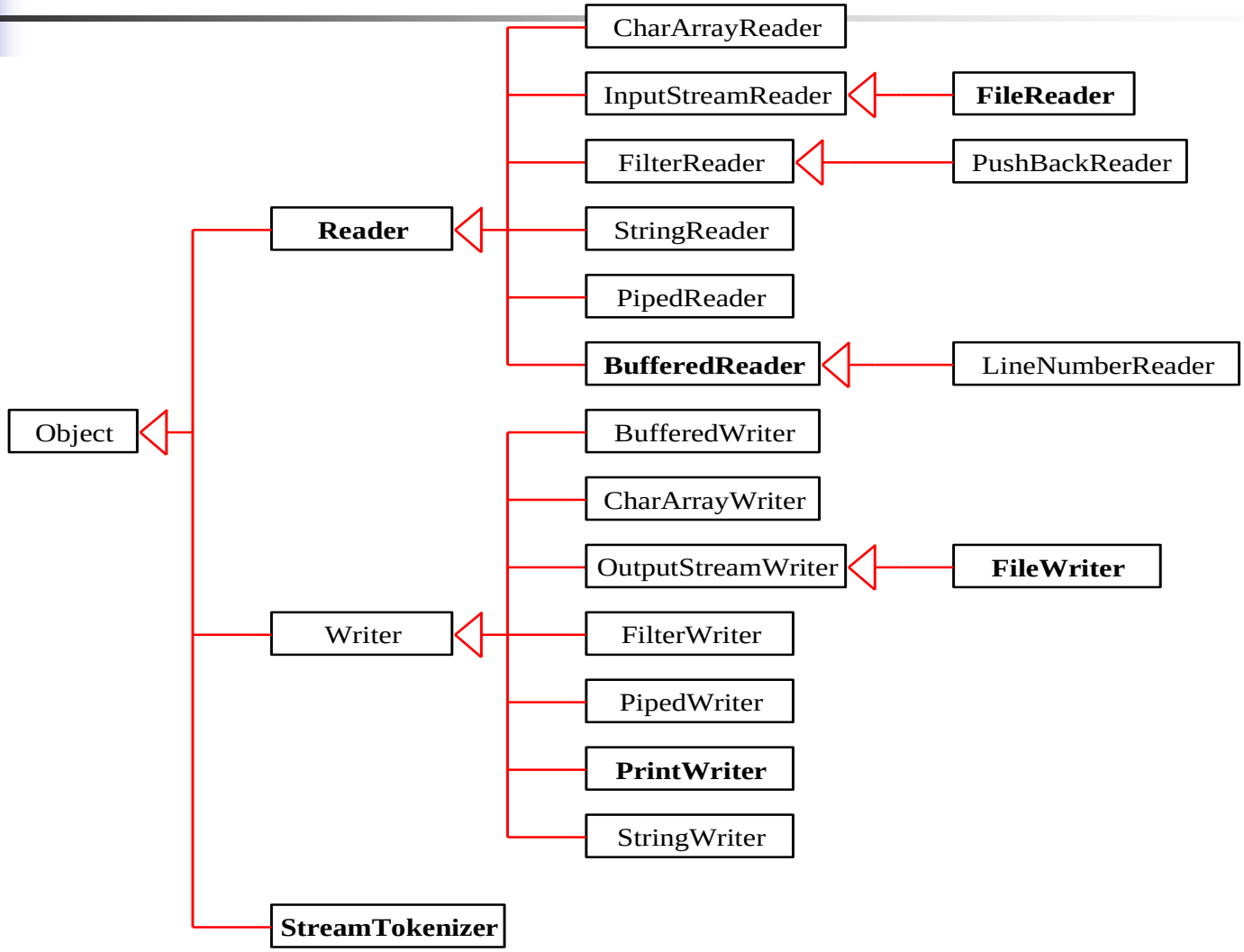
Classi stream di byte



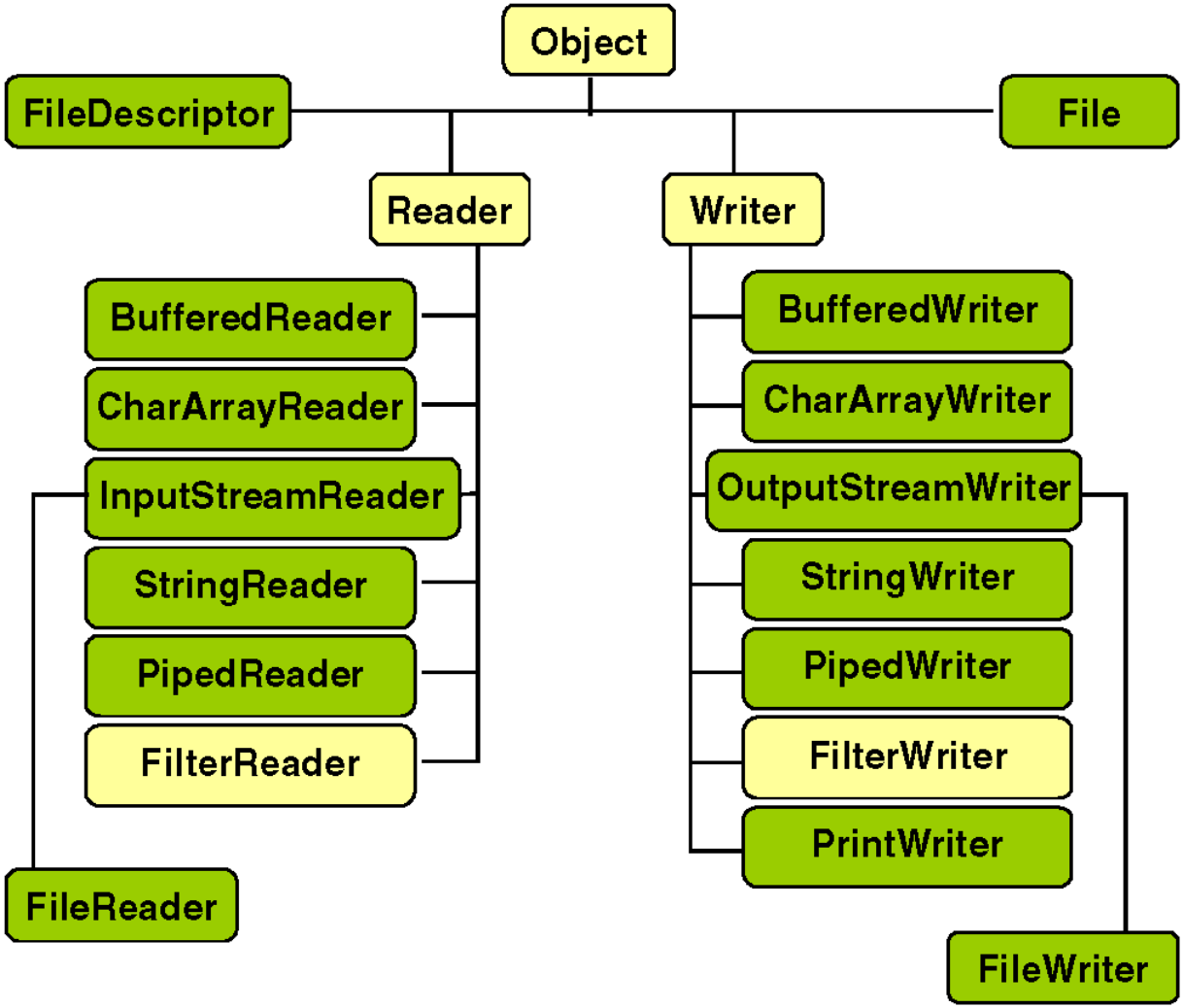
Classi stream di byte

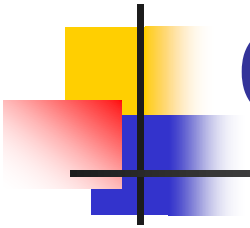


Classi stream di caratteri



Classi stream di caratteri





Classe Inputstream (byte)

- La classe astratta InputStream definisce il concetto generale di "flusso di input" operante sui byte
- Dichiara i metodi che permettono di leggere i byte a partire da una specifica sorgente
- InputStream è una classe astratta, quindi i metodi dovranno essere realmente definiti dalle classi derivate, in modo specifico a seconda della sorgente dati.
- Il metodo costruttore apre lo stream

Classe Inputstream (byte)

public abstract int read() throws IOException

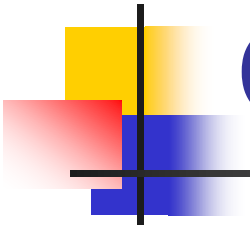
- Legge un singolo byte di dati e lo restituisce sotto forma di intero compreso tra 0 e 255. Se non è disponibile alcun byte restituisce il valore -1. Viene utilizzato un int e non un byte perchè i valori sono 257 (compreso il -1); chiaramente, solo I 16 bit meno significativi sono quelli utilizzati per restituire il valore 0-255

public int read(byte[] b, int offset, int count) throws IOException

- Legge una sequenza di byte e lo memorizza in una parte di un array di byte a partire dall'elemento di posizione *offset* e in numero massimo di *count*

public int read(byte[] b) throws IOException

- Come il metodo precedente con offset=0 e count=buff.length
- I metodi read con parametri sono un overload del primo, e utilizzano il codice del primo per operare, quindi è sufficiente che sia implementato tale primo metodo dalle sottoclassi per avere anche disponibili gli altri



Classe Inputstream (byte)

public long skip(long count) throws IOException

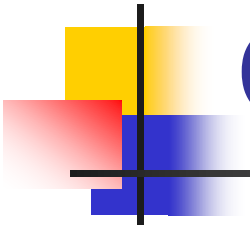
- Avanza il flusso saltando di un numero di byte pari a count. Si arresta se raggiunge la fine del flusso. Restituisce il numero effettivo di byte. Se count è negativo non avanza affatto

public int available() throws IOException

- Restituisce il numero di byte che possono essere letti senza arrestarsi

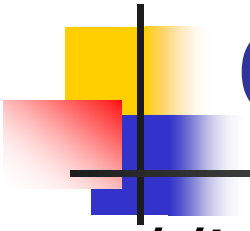
public void close() throws IOException

- Chiude il flusso. La chiusura di un flusso già chiuso non ha alcun effetto. Al suo interno dovrebbe anche rilasciare tutte le risorse associate al flusso (ad esempio descrittori di file). La liberazione di tali risorse non è automatica; potrebbe essere fatta all'interno di un eventuale metodo finalize(), ma la garbage collection è asincrona, per cui è opportuno inserire tutto il codice di rilascio risorse dentro tale metodo. L'implementazione di default di tale metodo non fa nulla.



Classe Outputstream (byte)

- La classe astratta OutputStream definisce il concetto generale di "flusso di output" operante sui byte
- Dichiara i metodi che permettono di scrivere dei byte a partire da una specifica sorgente
- OutputStream è una classe astratta, quindi i metodi dovranno essere realmente definiti dalle classi derivate, in modo specifico a seconda della sorgente dati.
- Il metodo costruttore apre lo stream



Classe Outputstream (byte)

public abstract void write(int b) throws IOException

- Scrive b sotto forma di byte. Del valore int b vengono presi in considerazione solo gli otto bit meno significativi; viene considerato int perchè solitamente l'argomento è il risultato di un'operazione aritmetica

public void write(byte[] b, int offset, int count) throws IOException

- Scrive una parte dell'array di byte buf a partire da buf[offset], scrivendo un numero di byte pari a count

int write(byte[] b) throws IOException

- Come sopra ma con offset=0 e count=buff.length

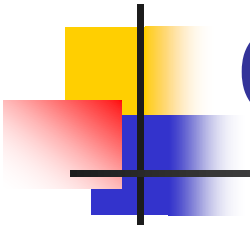
Classe Outputstream (byte)

public void flush() throws IOException

- Effettua lo svuotamento del flusso; se il flusso aveva depositato in un buffer i dati, li scrive immediatamente per svuotare il flusso; se l'uscita del flusso è un altro flusso, anche questo viene svuotato, il tutto seguendo la catena dei flussi. Se il flusso non è fornito di buffer, il metodo non ha effetto (cosa che coincide con l'implementazione di default)

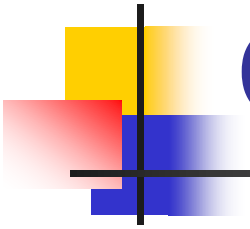
void close() throws IOException

- Chiude il flusso. Dovrebbe rilasciare le risorse (l'implementazione di default non fa nulla).



Classi per stream di caratteri

- Le classi per l'I/O da stream di caratteri (Reader e Writer) sono più efficienti di quelle a byte
  - hanno nomi analoghi e struttura analoga
  - Convertono correttamente la codifica UNICODE di Java in quella locale
    - specifica della piattaforma in uso (Windows, DOS, ASCII)…
    - …e della lingua in uso (essenziale per l'internazionalizzazione).



Classe Reader (caratteri)

- La classe Reader è simile alla classe InputStream. I metodi in Reader sono soggetti alla interpretazione dei caratteri.

public int read(char b[]) throws IOException

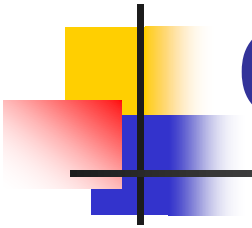
- Legge un carattere e lo restituisce come un intero compreso tra 0 e 65535. Oltre la fine del flusso restituisce -1

public abstract int read(char [] buf, int offset, int count) throws IOException

- Legge una sequenza di caratteri e lo memorizza in una parte di un array di char a partire dall'elemento di posizione offset e in numero massimo di count

public abstract int read(char buf[]) throws IOException

- Come sopra ma con offset=0 e count=buff.length



Classe Reader (caratteri)

public abstract int read(java.nio.CharBuffer buf[]) throws IOException

- Tenta di leggere quanti più caratteri possibili senza causare un overflow. Restituisce il numero di caratteri effettivamente letti

public long skip(long count) throws IOException

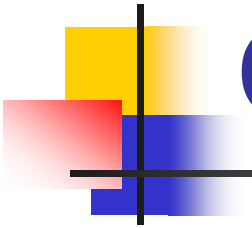
- Avanza il flusso di count caratteri. Si arresta se raggiunge la fine del flusso. Restituisce il numero effettivo di byte.

public boolean ready() throws IOException

- Restituisce true se il flusso è pronto per essere letto

public void close() throws IOException

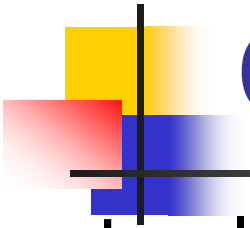
- Chiude il flusso. Dovrebbe rilasciare tutte le risorse.



Classe Reader (caratteri)

Esistono differenze fra Reader e InputStream:

- In Reader, il metodo fondamentale per la lettura utilizza un array di caratteri, e gli altri metodi sono definiti in termini di questo; per Inputstream, il metodo di base usa il singolo byte
- In Reader, il metodo close va solitamente ridefinito perchè l'implementazione di base vuota non è sufficiente
- InputStream permette di sapere quanti byte è possibile leggere, mentre Reader consente solo di sapere se esistono ancora dati



Classe Writer (caratteri)

- La classe Writer è simile alla classe OutputStream.

public void write (char b) throws IOException
- Scrive b sotto forma di carattere (16 bit meno significativi)

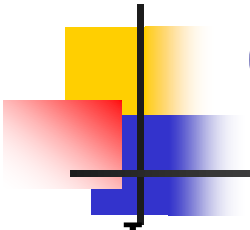
public abstract int write(char []buf, int offset, int count) throws IOException
- Scrive un array di caratteri da buf[offset] per count caratteri

public abstract int write(char buf[]) throws IOException
- Come sopra ma con offset=0 e count=buff.length

public abstract void flush() throws IOException
- Effetua lo svuotamento del flusso

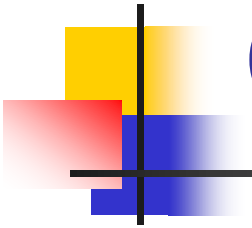
public void close() throws IOException
- Chiude il flusso.

- Valgono per la Writer le stesse considerazioni fatte per la Reader, tra cui il fatto che la versione fondamentale di write è quella relativa all'array di char, le altre utilizzano questa, che è quindi quella da implementare



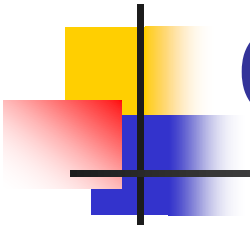
Conversione flussi

- I flussi di conversione InputStreamReader e OutputStreamWriter, estensioni rispettivamente di Reader e Writer (flussi di caratteri) permettono di convertire tra loro flussi di caratteri e byte utilizzando una codifica specifica o la codifica di default definita per il sistema locale
- Costruttori fondamentali:
  public InputStreamReader(InputStream in)
  public OutputStreamWriter (OutputStream in)
- I metodi read di InputStreamReader leggono i byte dall'InputStream assegnato, e li convertono in caratteri con la codifica appropriata; analogamente, i metodi write di OutputStreamReader converte i caratteri a disposizione in byte e li manda all'OutputStream assegnato
- In entrambe le classi, la chiusura del flusso di conversione comporta la chiusura del flusso di byte associato



Conversione flussi

- System.in, System.out e System.err sono i tre flussi standard associati all'input, output e standard error
- Storicamente, sono stati progettati insieme alle prime versioni di Java, quando esistevano solo flussi di byte, e tali sono anche se dovrebbero essere flussi di caratteri, il che può risultare talora scomodo o apparentemente incoerente



Classi dei flussi - Varianti

- Flussi **Filter** sono classi astratte che rappresentano tutti quei flussi ai quali è applicata una operazione di filtraggio; possono essere concatenati per produrre filtraggi sofisticati
- Flussi **Buffered** aggiungono ai flussi usuali la presenza di un buffer in modo che read e write non richiedano di accedere al file system ad ogni invocazione
- Flussi **Piped** vengono progettati come coppie in modo che all'interno della coppia possono essere canalizzate (piped) le informazioni

Classi dei flussi - Varianti

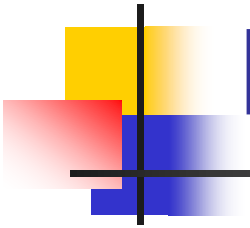
- I **Flussi di memoria** permettono di utilizzare le informazioni in memoria come sorgenti o destinazioni di un flusso:
  - ByteArray
  - CharArray
  - String
- **Flussi solo input o output** (senza controparte)
  - Print (forniscono i metodi print e println)
  - LineNumberReader (lettore con buffer per contare i numeri di riga)
  - SequenceInputStream (converte una sequenza di oggetti InputStream in uno solo)
- **Flussi per la costruzione di parser**
  - Flussi Pushback (per tornare indietro in fase di lettura (lookahead, utile per lo scanning))
  - StreamTokenizer (suddivide un Reader in un flusso di token, utile per il parsing)

Classi dei flussi Sincronizzazione e concorrenza

- La presenza di diverse classi che operano sui flussi richiede un meccanismo (possibilmente uniforme) di trattamento dei problemi di sincronizzazione e concorrenza
- **Le classi che operano sui flussi di byte** si sincronizzano su tale flusso, garantendo comunque l'atomicità delle operazioni, ad esempio in caso di scrittura simultanea da parte di due thread, il flusso risultante evita l'inframezzarsi dei byte scritti dai due thread, garantendo che si trovi prima l'intera sequenza di un thread e dopo l'intera sequenza dell'altro. Nell'operazione di lettura di N byte, i due thread accederebbero a due porzioni di N byte poste consecutivamente nel flusso
- **Le classi che operano sui flussi di caratteri** si sincronizzano su un capo protetto di nome lock che per default è un riferimento al flusso stesso, e che potrebbe anche essere un altro flusso (passabile come parametro dei metodi costruttori di Reader e Writer), ad esempio la classe StringWriter, sottoclasse di Writer, scrive i caratteri utilizzando un oggetto StringBuffer, a cui lock viene riferito; in generale, lock deve quindi essere riferito all'oggetto effettivamente usato come flusso

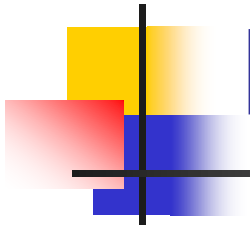
Classi dei flussi Sincronizzazione e concorrenza

- Nel caso si utilizzino **flussi incapsulati**, in accordo con l'approccio "a cipolla", la sincronizzazione dipende dal flusso più interno; questo potrebbe comportare problemi nel caso le classi incapsulanti richiedano operazioni atomiche al loro livello, operazioni che potrebbe non essere garantito che lo siano perché ad esempio potrebbero corrispondere a più operazioni del flusso interno, ciascuna atomica ma non atomiche nel loro insieme
- Le operazioni di input in genere sono **bloccanti** (si resta in attesa finchè i dati non diventano disponibili), e spesso lo sono anche quelle di output (la scrittura su un socket potrebbe dovere attendere).
- Se queste operazioni, come solitamente si fa, sono implementate da un thread, si potrebbe implementare anche la parte di codice relativa ad una richiesta di interruzione invocata sul thread stesso, ad esempio per liberare la risorsa o per dare informazioni sullo stato di avanzamento prima della richiesta di interruzione. Le situazioni che si dovrebbero fronteggiare sono tuttavia spesso complicate, pertanto **si preferisce in genere mantenere completamente bloccanti** (non interrompibili) le operazioni di I/O; java.nio offre strumenti aggiuntivi che permettono di affrontare il problema.



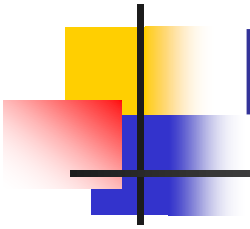
Flussi filter

- FilterInputStream FilterOutputStream sono i filtri per i flussi di byte
- FilterReader FilterWriter sono i filtri per i flussi di caratteri
- Ogni flusso di tipo filter è collegato con un altro flusso a cui delega le operazioni di I/O; per collegare tale flusso sono messi a disposizione costruttori che accettano in ingresso un flusso, proprio quello al quale saranno poi connessi
- E' possibile connettere un qualunque numero di flussi Filter di tipo byte o carattere, e la sorgente originale non deve necessariamente essere un flusso Filter; questo vale anche per i flussi di output
- Non tutte le classi di tipo Filter modificano realmente i dati, ad esempio una classe che introduce una bufferizzazione applica un filtro al flusso, ma solo di natura semantica (comportamentale)



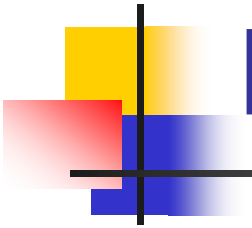
Flussi filter – Esempio

```
import java.io.*;
public class Convertitore extends FilterReader     {
        public Convertitore (Reader in) {super (in);}
        public int read() throws IOException
          {
          int c = super.read();
          return (c == -1 ? c : Character.toUpperCase((char)c));
          }

        public int read(char buf[], int offset, int count)throws IOException
          {
          int nread = super.read(buf,offset,count);
          int last = offset + nread;
          for (int i = offset; i<last; i++)
                                buf[i]=Character.toUpperCase(buf[i]);
          return nread;
          }
```



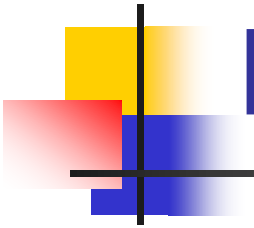
Flussi filter – Esempio

```
public static void main (String[] args)throws IOException
        {       StringReader src = new StringReader(args[0]);
                FilterReader f = new Convertitore(src);
                int c= f.read();
                while (c != -1)
                {
                        System.out.print((char)c);
                        c= f.read();
                }
        }

}
```



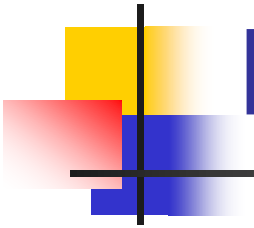
Flussi buffered

- Java introduce stream che utilizzano un buffer per ridurre il numero di accessi al flusso più interno, velocizzando quindi le operazioni di input e output (ad esempio nel caso di accesso ai file).
- Il flusso più interno è fornito anche in questo come parametro al costruttore; esiste anche un costruttore che permette di specificare anche la dimensione del buffer
- Quando si invoca il metodo read su un flusso bufferizzato vuoto, questo invoca il corrispondente read sul flusso interno, riempie il buffer e da quel momento le successive chiamate alla read attingono dal buffer per restituire i dati; solo quando il buffer si svuota si reinvoca la read sottostante
- La scrittura funziona in maniera analoga: la write del flusso bufferizzato riempie il buffer e solo quando è pieno produce un'invocazione sul write "reale" (del flusso interno)



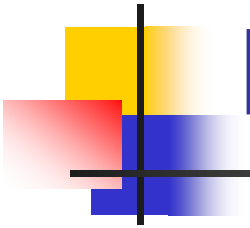
Flussi buffered

- Il flusso più interno fornito come parametro al costruttore non può essere restituito dall'oggetto contenitore, pertanto se occorre utilizzarlo è necessario **conservarne un riferimento**
- Quando tuttavia si mantiene un riferimento ad un flusso interno ad un altro (ad esempio bufferizzato), prima di utilizzare il flusso interno è opportuno **invocare un flush** sul flusso esterno, in modo che eventuali operazioni non ancora propagate dal flusso contenitore a quello interno possano avere luogo (ad esempio lo svuotamento del buffer) rendendo così consistente l'accesso al flusso interno



Flussi piped

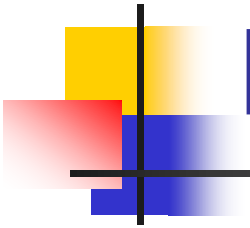
- I flussi Piped vengono utilizzati come coppie di flussi di input/output accoppiati. La pipe mantiene al suo interno un buffer della capacità definita in fase di implementazione che permette alle velocità di lettura e scrittura di essere differenti
- I flussi Piped forniscono un meccanismo per comunicare tra due thread differenti, associati alle due controparti; l'uso dei thread è in realtà l'unico modo sicuro di utilizzare le pipe
- Ogni pipe tiene traccia dei thread lettori e scrittori più recenti, e in una coppia di thread la pipe verifica l'esistenza della controparte prima di arrestare il thread corrente; se la controparte ha terminato l'esecuzione, viene sollevata una IOException
- Input: PipeInputStream e PipeReader
- Output: PipeOutputStream e PipeWriter



Flussi di memoria

- Mediante i flussi **ByteArray** è possibile utilizzare un array di byte come sorgente o destinazione di un flusso di byte
- La ByteArrayInputStream consente di effettuare l'input da un array di byte; la lettura non può arrestarsi (la sorgente è in memoria) e viene effettuata prelevando i dati dall'array (non viene effettuata una qualche copia interna), pertanto occorre preservare i dati per tutta la durata della lettura
- Per l'output si utilizza la ByteArrayOutputStream, che permette di memorizzare dati in un array di byte di dimensione crescente
- I flussi **CharArray** sono simili ai ByteArray ma operano su char, anche questi senza interruzioni
- **StringReader** legge senza interruzioni da una stringa (specificata come parametro nel costruttore)
- **StringWriter** permette di scrivere dati entro un buffer, e di recuperarne il contenuto come oggetto String o StringBuffer

Flussi monodirezionali Print

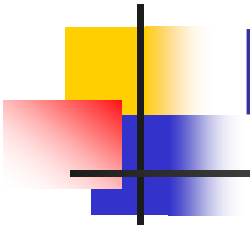


- I **flussi Print** permettono di semplificare la scrittura in forma leggibile agli umani dei valori di tipi primitivi e oggetti .
- La PrintStream opera su byte, la PrintWriter su char; la classe rilevante è la PrintWriter, ma per motivi storici System.out e System.err sono PrintStream; è opportuno quindi utilizzare solo la PrintWriter
- Forniscono entrambe i metodi print e println per Object, char, int, float, boolean, char[], long, double e String; tali metodi sono da utilizzarsi in sostituzione dei metodi write dei flussi quando si voglia ottenere un formato leggibile dagli umani, ad esempio data una variabile float f, l'istruzione out.print(f) invocata sul PrintStream out, se dovesse essere realizzata tramite write diventerebbe out.write(String.valueOf(f).getBytes())
- I flussi di stampa non sollevano IOException durante la scrittura; per controllare eventuali errori, occorre invocare il metodo checkError, che effettua il flush del flusso e ne controlla lo stato

Flussi monodirezionali LineNumberReader e SequenceInputStream

- Il **flusso LineNumberReader** tiene traccia dei numeri di riga durante la lettura del testo. Il testo solitamente proviene da un flusso passato come parametro al costruttore di un oggetto LineNumberReader, sempre secondo l'approccio "a cipolla"
- Una riga si ritiene conclusa quando si incontra il linefeed (\n) il carriage return (\r) o ambedue di seguito (\r\n); in corrispondenza di tali caratteri, si conteggiano le linee, potendo poi conoscerne il numero con il metodo getLineNumber; esiste anche il setLineNumber per potere controllare il conteggio
- La **classe SequenceInputStream** crea un unico flusso di input leggendo e concatenando uno o piu flussi di byte InputStream
- La gestione dei flussi da concatenare avviene tramite un oggetto di tipo Enumeration

Flussi per Parser Pushback



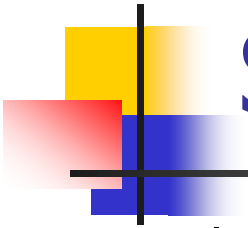
- I flussi Pushback permettono di "riportare indietro" la lettura di caratteri o byte quando ci si è spinti troppo avanti (lookahead)
- il flusso è bufferizzato (la dimensione viene specificata con il costruttore), quindi si possono riportare indietro un numero di byte o caratteri pari allo spazio libero corrente del buffer (ecomunque non superiore alla dimensione max); se la richiesta è di più byte o caratteri viene sollevata un'IOException
- I dati sono reinseriti nel buffer, e questo potrebbe alterarne l'ordine, ad esempio se sono letti i caratteri '1', '2', '3', e poi si invoca per tre volte il metodo unread('1'), unread('2'), unread('3'), le successive letture produrranno '3', '2', '1', in accordo all'ordine di inserimento nel buffer

Flussi per Parser Pushback

Esempio che stampa la lunghezza della più lunga sequenza di byte uguali:

```
import java.io.*;
class Cont {
  public static void main (String s[]) throws IOException
{       PushbackInputStream in = new PushbackInputStream(System.in);
        int max = 0;     int maxB = -1;  int b;
        do {
                int count;
                int b1 = in.read();
                for (count = 1; (b = in.read()) == b1; count ++) continue;
                if (count > max)
                {       max = count;
                        maxB = b1;
                }
               in.unread(b);
        } while (b!= -1);
        System.out.println(max + "byte di" + maxB);}}
```

Flussi per Parser StreamTokenizer



- La classe StreamTokenizer consente di leggere uno stream di ingresso (incapsulato e fornito come parametro nel costruttore) e identifica i token presenti nella stringa; Una classe più generale è java.util.scanner
- La classe prevede un insieme di metodi per impostare i criteri di scansione e tokenizzazione; una volta scelti, il metodo nextToken applicato al flusso consente di ottenere il token corrente
- I metodi per i criteri permettono di specificare quali caratteri:
  - sono da considerarsi come parte delle parole; di default sono gli intervalli 'a', …, 'z', 'A', …, 'Z', e i valori non ASCII 7bit dal carattere 128+32 a 255 (ISO8859)
  - rappresentano spazi bianchi (separatori di token); di default sono dal codice ascii 0 al 32 (' ')
  - sono ordinari, ossia da ignorare (vengono letti così come sono)
  - rappresentano l'identificatore di un commento; di default è '/'
  - rappresentano i delimitatori delle costanti string; default ' " '
- i metodi hanno effetto cumulativo (si possono aggiungere intervalli e/o singoli caratteri); resetSyntax() consente di resettare i criteri sinora impostati

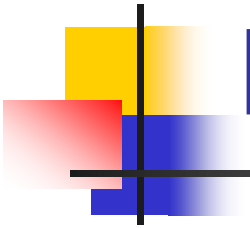
Flussi per Parser StreamTokenizer

- Esistono quattro tipi di token, ciascuno identificato da una costante della classe:
  TT_WORD, se il token è una parola.
  TT_NUMBER, se il token è un numero.
  TT_EOL, se lettura fine linea.
  TT_EOF, se si incontra la fine del file.
- Esistono anche i campi della classe:
  int ttype, che contiene il tipo del token corrente
  double nval, che contiene il valore del token corrente se è un numero.
  String sval, che contiene il valore del token corrente se è una stringa.
- Il metodo principale infine è nextToken:
  public int nextToken() throws IOException
  - Il token sucessivo viene trovato nella stringa di input del StreamTokenizer.
  - Il tipo del token successivo viene ritornato nel campo ttype Se ttype == TT_WORD, il token è memorizzato in sval;
  - se ttype == TT_NUMBER, il token è memorizzato in nval.

Flussi per Parser StreamTokenizer

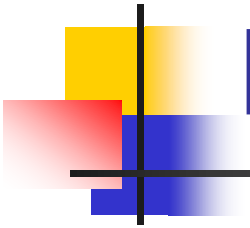
Esempio che restituisce la somma dei valori numerici trovati nel flusso:

```
import java.io.*;
class Prova2 {
  public static void main (String s[]) throws IOException
{       double x = sumStream(new FileReader("pippo.txt");
        System.out.println(x);
}
public static double sumStream (Reader source) throws IOException
{       StreamTokenizer in = newStreamTokenizer(source);
        double result = 0.0;
        while (in.nextToken() != StreamTokenizer.TT_EOF) {
                if (in.ttype == StreamTokenizer.TT_NUMBER)
                        result = result + in.nval;
        }
        return result;} }
```



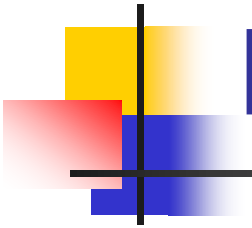
Flussi DataInput e DataOutput

- Gli stream di dati (DataInputStream e DataOutputStream) leggono e scrivono tipi primitivi in modo "machine-independent"
- Ciò consente di scrivere un file in un computer e leggerlo in un altro con caratteristiche diverse (sistema operativo, struttura del file diversa).
- Rappresentano una implementazione dell'interfaccia DataInput e DataOutput rispettivamente; la classe RandomAccessFile implementa entrambe le interfacce
- Esistono metodi specifici per la lettura e scrittura dei tipi primitivi:
  int readByte() throws IOException
  int readShort() throws IOException
  int readInt() throws IOException
  int readLong() throws IOException
  float readFloat() throws IOException
  double readDouble() throws IOException
  char readChar() throws IOException
  boolean readBoolean() throws IOException
  String readUTF() throws IOException
  Esistono tutti i corrispondenti write



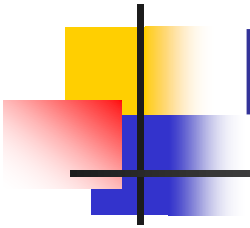
Flussi per i file

- FileInputStream, FileOutputStream FileReader e FileWriter permettono di trattare un file come un flusso; presentano tre costruttori
  - Uno con argomento una String, il nome di un file
  - Un oggetto File
  - Un FileDescriptor
- I flussi di byte non operano con i canali, tuttavia sono in grado di fornire un oggetto FileChannel del package java.nio.Channel
- Oltre gli oggetti File, che rappresentano sostanzialmente flussi collegati a file (il ruolo del file è secondario), esistono:
  - la classe **FileDescriptor**, usata per avere un nuovo flusso a partire dallo stesso file; occorre tuttavia prestare attenzione a più flussi che insistono sullo stesso file
  - La classe **File**, distinta dai flussi, che fornisce operazioni per la manipolazione del nome del file: path, esistenza del file, permessi, creazione, cancellazione (anche di directory) ecc.
  - La classe **RandomAccessFile**, che rispetto ai semplici flussi offre un "cursore" per il posizionamento all'interno del file, più i relativi metodi per la sua gestione (seek, skip ecc.)



Flussi per i file

- FileInputStream è la classe derivata che rappresenta il concetto di sorgente di byte agganciata a un file
  - il nome del file da aprire è passato come parametro al costruttore di FileInputStream
  - in alternativa si può passare al costruttore un oggetto File (o un FileDescriptor) costruito in precedenza



Flussi per i file – Lettura byte

- Per aprire un file binario in lettura si crea un oggetto di classe FileInputStream, specificando il nome del file all'atto della creazione.
- Per leggere dal file si usa poi il metodo read() che permette di leggere uno o più byte
  - restituisce il byte letto come intero fra 0 e 255
  - se lo stream è finito, restituisce -1
  - se non ci sono byte, ma lo stream non è finito, rimane in attesa dell'arrivo di un byte.
- Poiché è possibile che le operazioni su stream falliscano per varie cause, tutte le operazioni possono lanciare eccezioni, quindi necessità di try / catch

Flussi per i file – Lettura byte

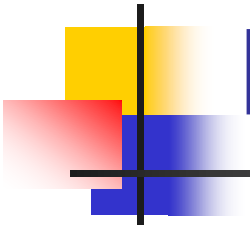
```
import java.io.*;
public class LetturaDaFileBinario {
        public static void main(String args[]){
                FileInputStream is = null;
        try {
                is = new FileInputStream(args[0]);
                }
        catch(FileNotFoundException e){
                System.out.println("File non trovato");
                System.exit(1);
                }
// ... lettura ...
        }
}
```

Flussi per i file – Lettura byte

La fase di lettura:

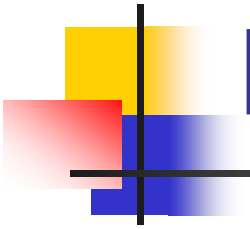
quando lo stream termina, **read()** restituisce -1

```
...
try {
        int x = is.read();
        int n = 0;
        while (x>=0) {
                System.out.print(" " + x); n++;
                x = is.read();
        }
        System.out.println("\nTotale byte: " + n);
    } catch(IOException ex){
                System.out.println("Errore di input");
                System.exit(2);
        }
```



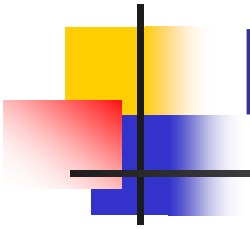
Flussi per i file – Lettura byte

Esempio d'uso:
C:\temp>java LetturaDaFileBinario question.gif
Il risultato:
71 73 70 56 57 97 32 0 32 0 161 0 0 0 0 0 255 255 255
.....
Totale byte: 190



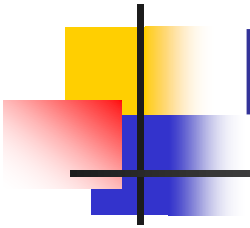
Flussi per i file – Lettura byte

- Lettura di dati da file binario
  - Per leggere da un file binario occorre un FileInputStream, che però consente solo di leggere un byte o un array di byte
  - Volendo leggere dei float, int, double, boolean, … è molto più pratico un DataInputStream, che ha metodi idonei
  - Si incapsula FileInputStream dentro un DataInputStream



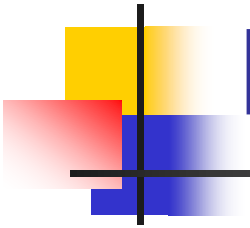
Flussi per i file – Lettura byte

```
import java.io.*;
        public class Esempio2 {
                public static void main(String args[]){
                        FileInputStream fin = null;
                try {fin = new FileInputStream("Prova.dat");
                }
                catch(FileNotFoundException e){
                        System.out.println("File non trovato");
                        System.exit(3);
                }
// continua...
```



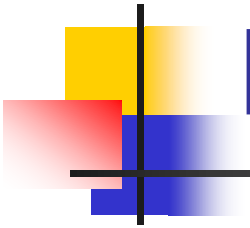
Flussi per i file – Lettura byte

```
DataInputStream is = new DataInputStream(fin);
float f2; char c2; boolean b2; double d2;
int i2;
try {
        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);
}
}
```



Flussi per i file – Scrittura byte

- FileOutputStream è la classe derivata che rappresenta il concetto di dispositivo di uscita agganciato a un file
  - il nome del file da aprire è passato come parametro al costruttore di FileOutputStream
  - in alternativa si può passare al costruttore un oggetto File (o un FileDescriptor) costruito in precedenza FileOutputStream



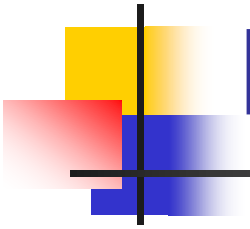
Flussi per i file – Scrittura byte

- Per aprire un file binario in scrittura si crea un oggetto di classe FileOutputStream, specificando il nome del file all'atto della creazione
  - un secondo parametro opzionale, di tipo boolean, permette di chiedere l'apertura in modo append
  - Per scrivere sul file si usa il metodo write() che permette di scrivere uno o più byte
    - scrive l'intero (0 - 255) passatogli come parametro
    - non restituisce nulla
- Poiché è possibile che le operazioni su stream falliscano per varie cause, tutte le operazioni possono lanciare eccezioni, quindi necessità di try / catch

Flussi per i file – Scrittura byte

```
import java.io.*;
public class ScritturaSuFileBinario {
        public static void main(String args[]){
                FileOutputStream os = null;
        try {   os = new FileOutputStream(args[0]);
        }
        catch(FileNotFoundException e){
                System.out.println("Imposs. aprire file");
                System.exit(1);
        }
// ... scrittura ...
        }
}
```

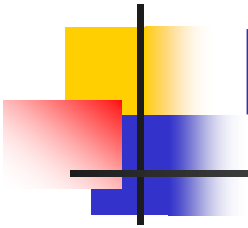
Per aprirlo in modalità append:
FileOutputStream(args[0],true)



Flussi per i file – Scrittura byte

Esempio: scrittura di alcuni byte a scelta

```
...
try {
        for (int x=0; x<10; x+=3) {
        System.out.println("Scrittura di " + x);
        os.write(x);
        }
} catch(IOException ex){
        System.out.println("Errore di output");
        System.exit(2);
        }
...
```



Flussi per i file – Scrittura byte

Esempio d'uso:

C:\temp>java ScritturaSuFileBinario prova.dat

Il risultato:

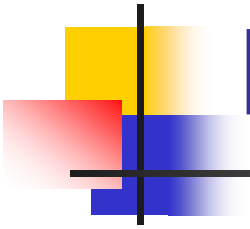
Scrittura di 0
Scrittura di 3
Scrittura di 6
Scrittura di 9

Controllo:

C:\temp>dir prova.dat
16/01/01 prova.dat 4 byte

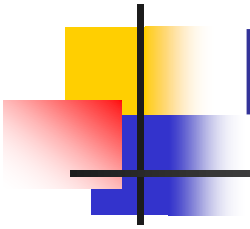
Esperimenti

Provare a rileggere il file con il programma precedente
Aggiungere altri byte riaprendo il file in modo append



Flussi per i file – Scrittura byte

- Scrittura di dati su 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 ha metodi idonei
  - Si incapsula FileOutputStream dentro un DataOutputStream

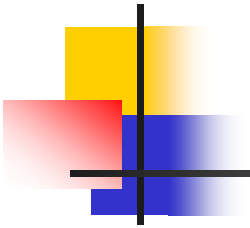


Flussi per i file – Scrittura byte

```
import java.io.*;
public class Esempio1 {
        public static void main(String args[]){
                FileOutputStream fs = null;
        try {
                fs = new FileOutputStream("Prova.dat");
                }
        catch(IOException e){
                System.out.println("Apertura fallita");
                System.exit(1);
        }
// continua...
```

Flussi per i file – Scrittura byte

```
DataOutputStream os = new DataOutputStream(fs);
float f1 = 3.1415F; char c1 = 'X';
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);
        }
}
}
```



Random Access Files

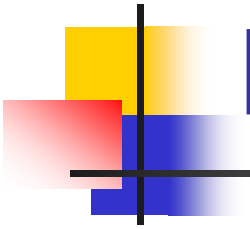
- La classe RandomAccessFiles rappresenta un file con un cursore utilizzato per l'accesso diretto
- RandomAccessFiles, non è sottoclasse di nessuna InputStream, OutputStream o Reader o Writer, in quanto implementa funzionalità sia di input che di output; implementa anche le interfacce DataInput e DataOutput per i tipi primitivi, oltre che i metodi read e write delle classi dei flussi; un oggetto flusso non è tuttavia sostituibile con uno RandomAccessFile
- I metodi sono:

void seek(long pos) throws IOException;
  - offset dall'inizio del RandomAccessFile.

long getFilePointer() throws IOException;
  - Ritorna il corrente offset, in bytes, dall'inizio del file.

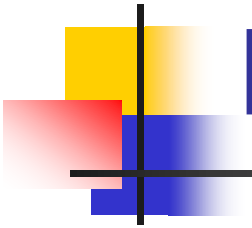
long length() throws IOException
  - Ritorna la lunghezza del file.

Void setLength(long newLength) throws IOException
  - Consente di variare la dimensione del file, riempendolo di byte se più corto o troncandolo se più lungo



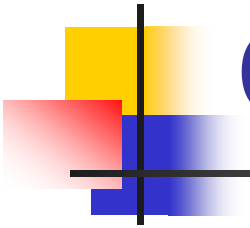
Flussi per i file – Caratteri

- Cosa cambia rispetto agli stream binari ?
  - Il file di testo si apre costruendo un oggetto FileReader o FileWriter, rispettivamente
  - read()e write() leggono/scrivono un int che rappresenta un carattere UNICODE
    - ricorda: un carattere UNICODE è lungo due byte
    - read() restituisce -1 in caso di fine stream
- Occorre dunque un cast esplicito per convertire il carattere UNICODE in int e viceversa



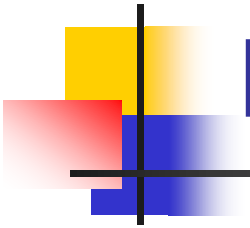
Flussi per i file – Caratteri

- Gli stream di byte sono più antichi e di livello più basso rispetto agli stream di caratteri
  - Un carattere UNICODE è una sequenza di due byte
  - Gli stream di byte esistono da Java 1.0, quelli di caratteri da Java 1.1 ERGO, le classi esistenti fin da Java 1.0 usano stream di byte anche quando sarebbe meglio usare stream di caratteri ma i caratteri potrebbero non essere sempre trattati in modo coerente (in applicazioni che operano con diverse codifiche, magari su piattaforme diverse, o via rete) .
- Alcuni canali di comunicazione sono stream di byte ma a volte si devono usare per inviare / ricevere caratteri (caso tipico: stream di rete - si inviano GIF ma anche testo)



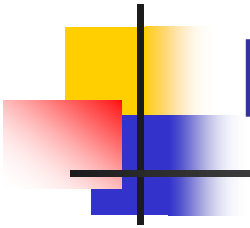
CONSEGUENZA

- Occorre spesso poter reinterpretare uno stream di byte come reader / writer quando opportuno (cioè quando trasmette caratteri)
- Esistono due classi "incapsulanti" progettate proprio per questo scopo:
  - InputStreamReader che reinterpreta un InputStream come un Reader
  - OutputStreamWriter che reinterpreta un OutputStream come un Writer



Flussi per i file – Lettura caratteri

```
import java.io.*;
                public class LetturaDaFileDiTesto {
                        public static void main(String args[]){
                                FileReader r = null;
                try {
                        r = new FileReader(args[0]);
                }
                catch(FileNotFoundException e){
                        System.out.println("File non trovato");
                        System.exit(1);
                }
// ... lettura ...
        }
}
```

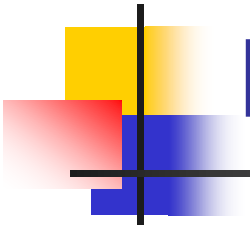


Flussi per i file – Lettura caratteri

Cast esplicito da **int** a **char** - Ma solo se è stato davvero letto un carattere (cioè se non è stato letto -1)

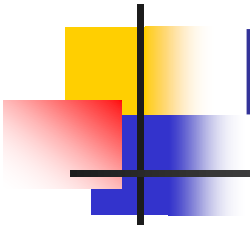
La fase di lettura:

```
...
try {
                int n=0, x = r.read();
                while (x>=0) {
                        char ch = (char) x;
                        System.out.print(" " + ch); n++;
                        x = r.read();
                }
                System.out.println("\nTotale caratteri: " + n);
} catch(IOException ex){
                System.out.println("Errore di input");
                System.exit(2);
}
```



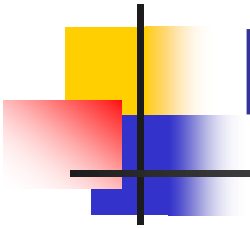
Flussi per i file – Lettura caratteri

Esempio d'uso:
C:\temp>java LetturaDaFileDiTesto prova.txt
Il risultato:
N e l m e z z o d e l c a m m i n d i n o s t r a v i t a
Totale caratteri: 35
Analogo esercizio può essere svolto per la scrittura su file di testo.



Flussi per i file – I/O console

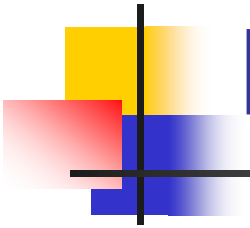
- Video e tastiera sono rappresentati dai due oggetti statici System.in e System.out
  - Poiché esistono fin da Java 1.0 (quando Reader e Writer non esistevano), essi sono formalmente degli stream di byte…
    - System.in è formalmente un InputStream
    - System.out è formalmente un OutputStream
  - ma in realtà sono stream di caratteri.
- Per assicurare che i caratteri UNICODE siano ben interpretati occorre quindi incapsularli rispettivamente in un Reader e in un Writer e specificare il character encoding opportuno.



Flussi per i file – I/O console

- System.in può essere "interpretato come un Reader" incapsulandolo dentro a un Input-StreamReader
- System.out può essere "interpretato come un Writer" incapsulandolo dentro a un OutputStreamWriter
- Tipicamente:

```
InputStreamReader tastiera =
    new InputStreamReader(System.in);
OutputStreamWriter video =
    new OutputStreamWriter(System.out);
```

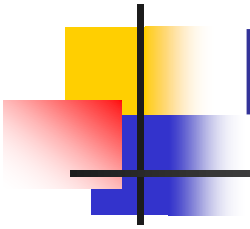


Flussi per i file – I/O console

```
import java.io.*;
public class ConsoleInput {
   public static void main(String[] args) {
      try {
        // crea un oggetto ins e lo collega alla console.
        InputStreamReader inStream = new

InputStreamReader(System.in);
        BufferedReader ins = new
BufferedReader(inStream);

        // Legge il nome del fle di output.
        System.out.print("Enter output file name: ");
        String outFileName = ins.readLine();
```

Flussi per i file – I/O console

```
   // crea un oggetto outs -- output file.
FileWriter outStream = new FileWriter(outFileName);
PrintWriter outs = new PrintWriter(outStream);

// legge il numero di linee del file.
System.out.println("How many data lines?");
int numLines = Integer.parseInt(ins.readLine());

  // legge le righe dalla console e le scrive sul file di output

System.out.println("Type " + numLines + " lines:");
for (int lineCount = 0; lineCount < numLines;
                                        lineCount++)  {
    String dataLine = ins.readLine();
    outs.println(dataLine);
}
```



Flussi per i file – I/O console

```
// scrive un messaggio sulla console e chiude il file
        System.out.println(numLines + "righe inserite nel file"+
outFileName);
      ins.close();
      outs.close();
    }
    catch (IOException ex) {
      System.out.println("i/o error: " + ex.getMessage());
      ex.printStackTrace();
    }
    catch (NumberFormatException ex) {
      System.out.println(ex.getMessage());
      ex.printStackTrace();
    } }      }
```

Flussi per i file – I/O console

Per cancellare lo schermo (clrscr):

- *System.out.print((char)27 + "[2J");*

it doesn't work unless "ansi.sys" is loaded and very few WinXP user's have this.

- *Runtime.exec("cls");* this however makes your application platform-dependent.
- Printing out the form-feed character will clear the screen: *System.out.print("\f");*
- JCurses is an implementation of nCurses (una libreria di funzioni software che gestisce il display di un'applicazione su un terminale a caratteri) which determines the console type being used, and wraps escape sequences with a nice API, independently of the type of console being used.

  http://sourceforge.net/projects/javacurses/

  However, you should consider the more standard Java/Swing and make a nice GUI interface to your application.
- Clearning the screen, cursor positioning etc are inherently operating system-specific, and cannot be implemented in a standard way across platforms. Even the 'standard' C and C++ languages which, like Java, aim for platform independance, do not implement such functionality.