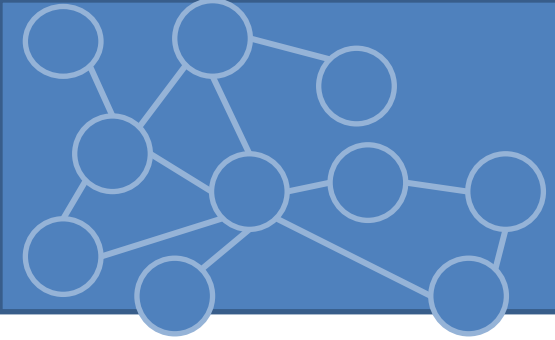
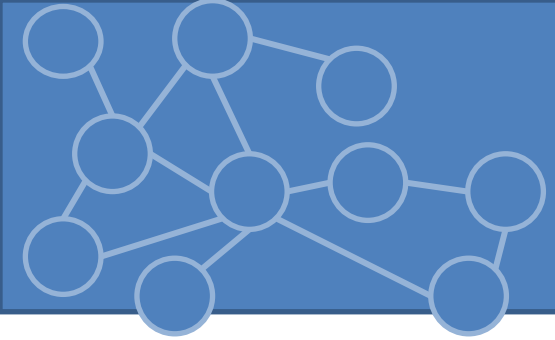


Laboratorio Reti di Calcolatori

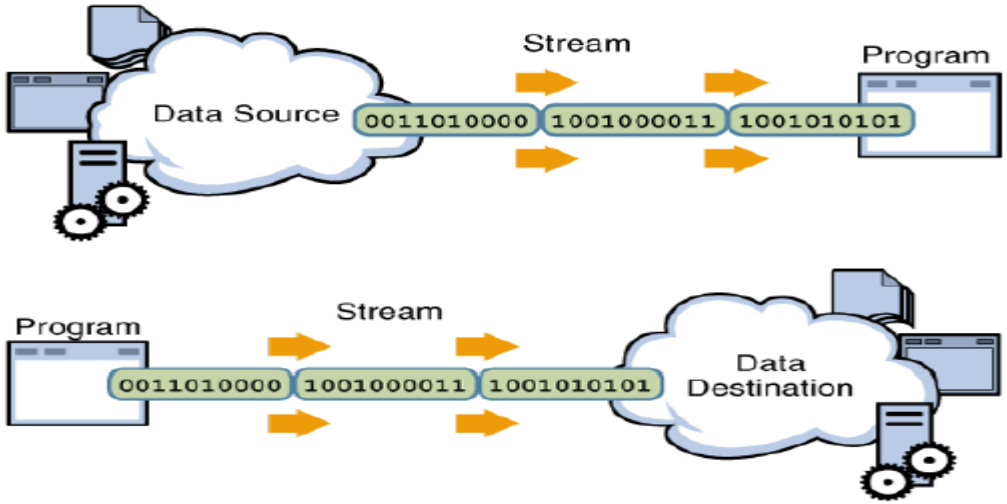
Laurea Triennale in Comunicazione Digitale

Anno Accademico 2012/2013

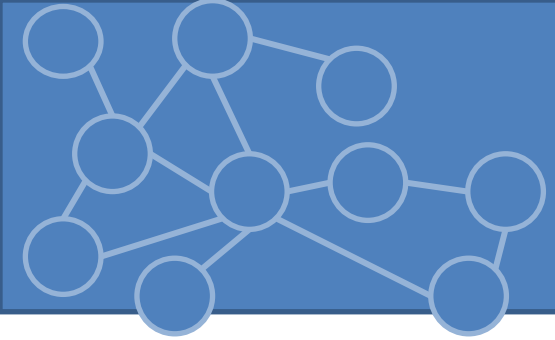


Streams

- Maggior parte della programmazione di rete è eseguire operazione di input e output.
- Sposto byte da un sistema all'altro => leggere dati da un server è molto simile a leggere i byte di un file
- I/O in Java è costruito sul concetto di STREAM
  - Input streams: leggere dati
  - Output streams: scrivo dati



- Diverse classi scrivono e leggono da particolari sorgenti di dati (file, programma, periferica, network socket).
- Tutte le classi di output e le relative di input usano gli stessi metodi per scrivere / leggere dati
- Streams sono SINCRONI: quando invoco metodi di in e out aspetta che l'operazione venga eseguita. Dalla 1.4 vengono supportati I/O non bloccanti (channels e buffer)



OutputStream

- Classe base di output (livello byte) è *java.io.OutputStream*

public **abstract** void **write(int b)** throws IOException

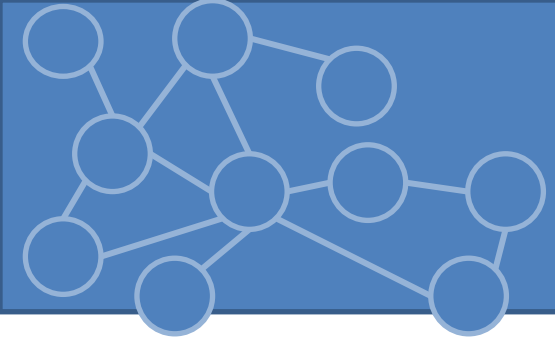
public void **write(byte[] data)** throws IOException

public void **write(byte[] data, int offset, int length)** throws IOException

public void **flush( )** throws IOException

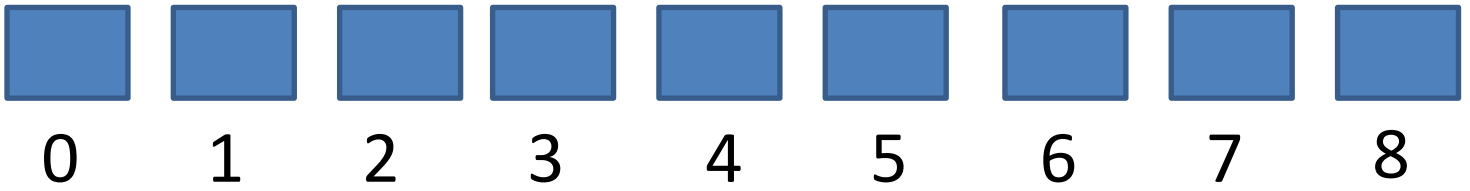
public void **close( )** throws IOException

- *FileOutputStream, TelnetOutputStream, ByteArrayOutputStream (classi concrete)*



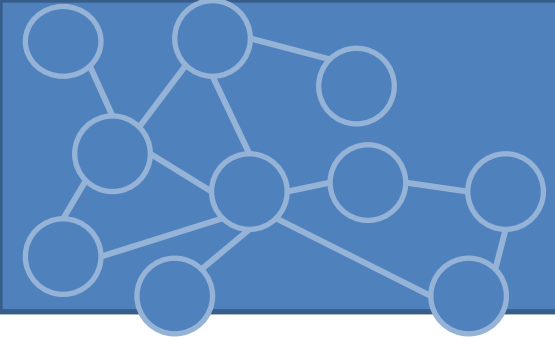
- ***write(int b)*** riceve un intero tra 0 e 255 e scrive il byte corrispondente nello stream di output.
  - Dichiarato abstract perché sottoclassi devono adattarsi allo stream che modellano
  - Se b > 255, viene scritto solo il byte meno significativo (effetto del casting di un int in un byte)
  - Scrivo un byte alla volta nello stream => spesso inefficiente

- ***write(byte[] data, int offset, int length)***

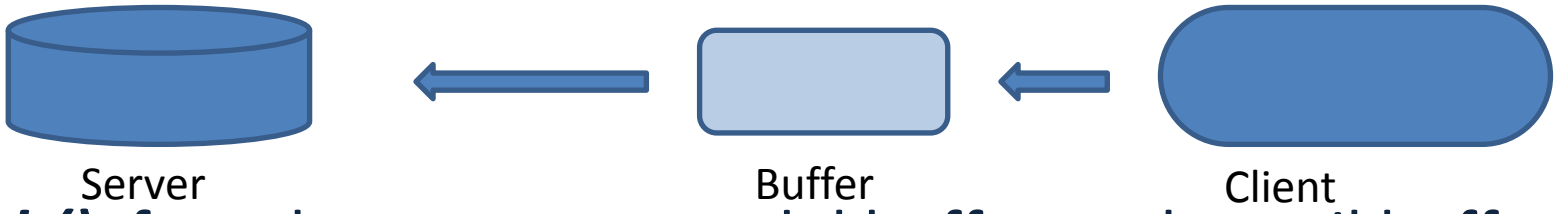


Da data[offset] a data[offset+length-1]

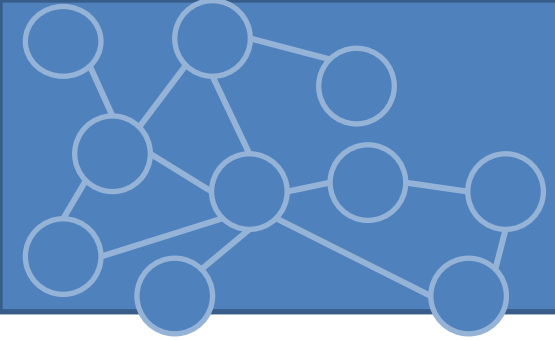
Invoca il metodo *write(int b)*, un numero di volte specificato da length



- Streams possono essere bufferizzati (codice Java o network hardware)
  - Collego lo stream a un *BufferedOutputStream* o *BufferedWriter*
- E' importante svuotare il buffer = flush
  - Es: HTTP Keep-Alive



- ***flush()***: forza la svuotamento del buffer anche se il buffer non è ancora pieno
  - Flush uno stream prima di chiuderlo
- ***close()***: chiude lo stream rilasciando le risorse associate allo stream (file handler o porte)

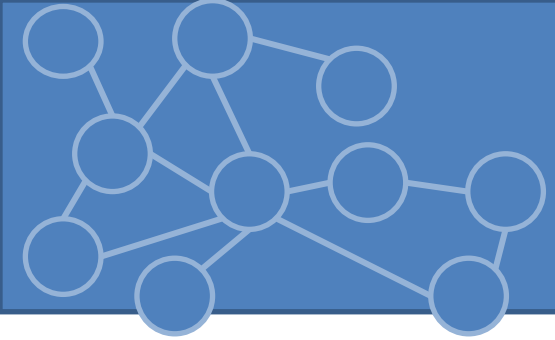


InputStream

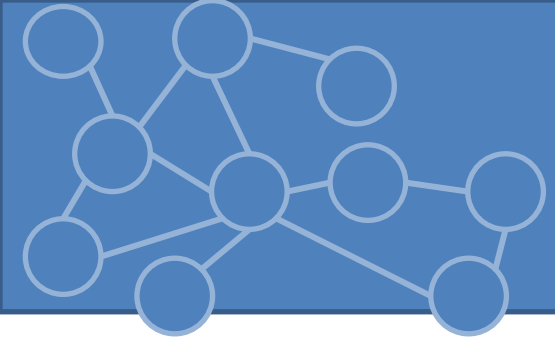
- Classe base di input (livello byte) è *java.io.InputStream*

public **abstract** int read( ) throws IOException
public int read(byte[] input) throws IOException
public int read(byte[] input, int offset, int length) throws IOException
public long skip(long n) throws IOException
public int available( ) throws IOException
public void close( ) throws IOException

- Es: *FileInputStream, TelnetInputStream, ByteArrayInputStream*
- È una classe astratta



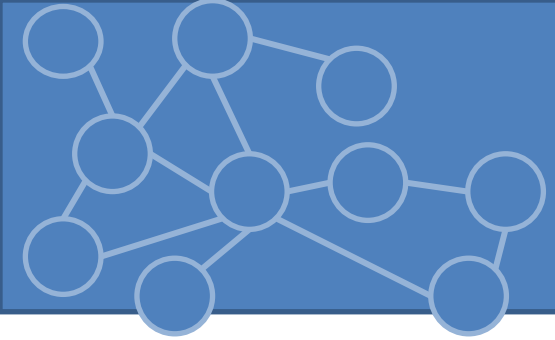
- ***read()***: legge un singolo byte e lo restituisce come un intero compreso tra 0 e 255.
  - La conclusione dello stream viene indicata restituendo un -1
  - Attende e sospende l'esecuzione del codice che segue il metodo fino a che è disponibile un byte di dati
  - I/O può essere lento => assegno un thread per I/O



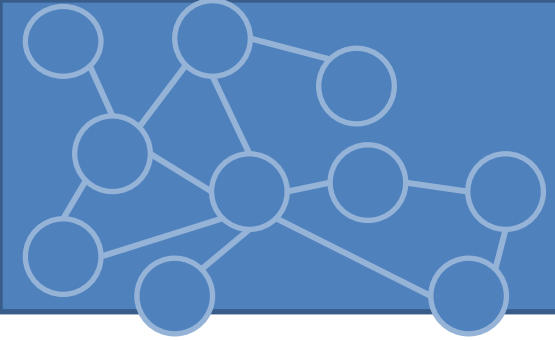
- ***read(byte[] input)*** e ***read(byte[] input, int offset, int length)***
  - Tentano di riempire l'array di byte input rispettando i vincoli imposti dai parametri
  - Restituiscono il numero di byte letti effettivamente

```
int byteLetti = 0;
int lenghtBuffer = 1024;
byte[] input = new byte[lenghtBuffer];
while (byteLetti < lengthBuffer) {
  byteLetti += in.read(input, byteLetti, lengthBuffer - byteLetti);}
```

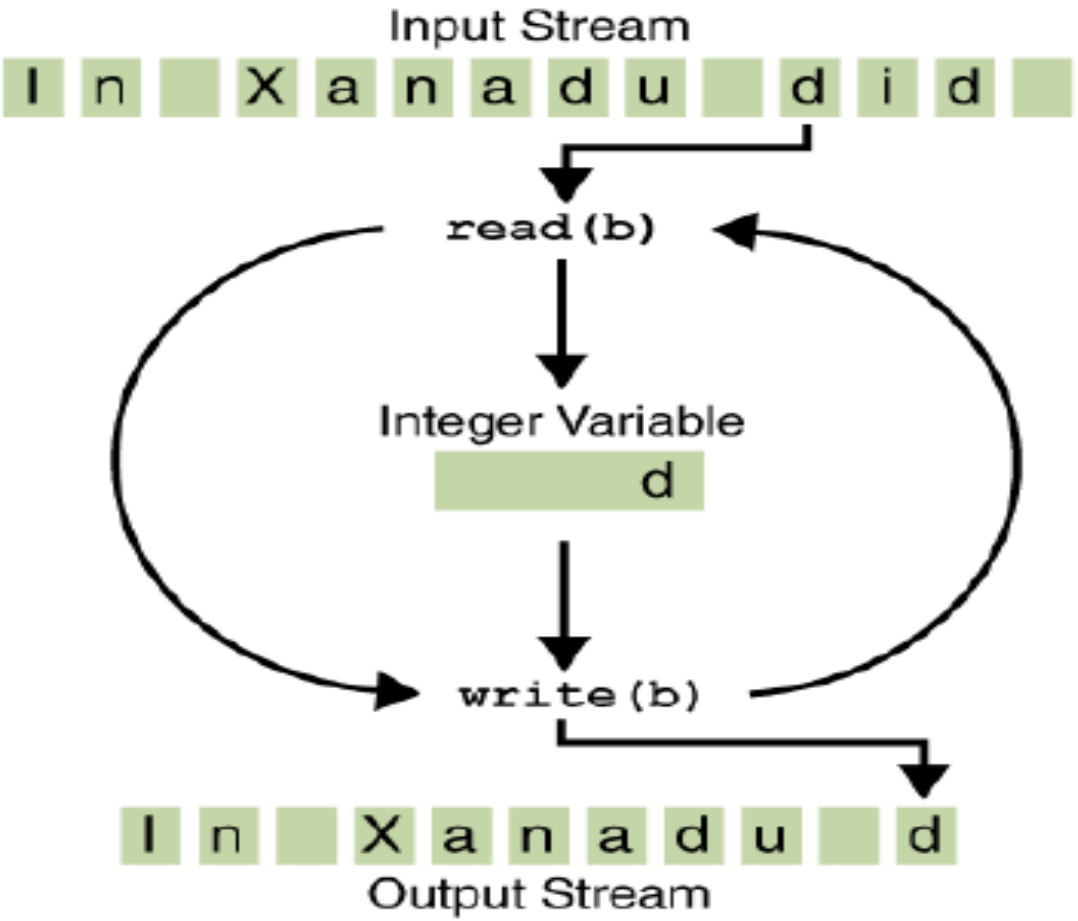
- Se stream finisce mentre ci sono ancora dati nel buffer, vengono restituiti bytes finchè il buffer non è vuoto

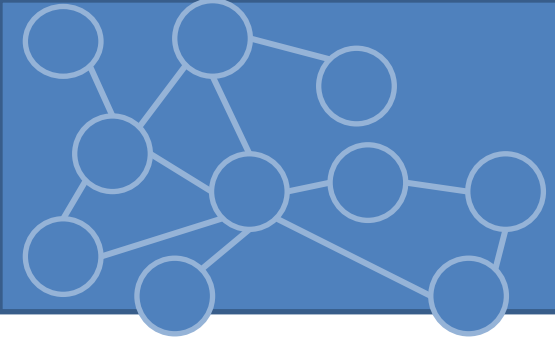


- ***available()***: restituisce una **stima** (numero minimo di byte che posso leggere) di quanti possono essere letti senza bloccarsi.
- ***skip(long)***: andare oltre nello stream senza leggere i dati
  - Poco utile con network stream, più utile con file perché c'è accesso casuale e posso riposizionare il puntatore
- ***close()***: rilascia ogni risorsa associata allo stream
  - IOException se tento ulteriori letture



- ***mark(int readLimit)***: per rileggere i dati posso marcare la posizione e posso riportare lo stream alla posizione marcata usando il metodo ***reset()***. readLimit indica il numero di byte che posso leggere dalla posizione marcata.
  - Un solo mark alla volta per un dato stream
- ***markSupported()***: indica se un dato stream supporta il marking

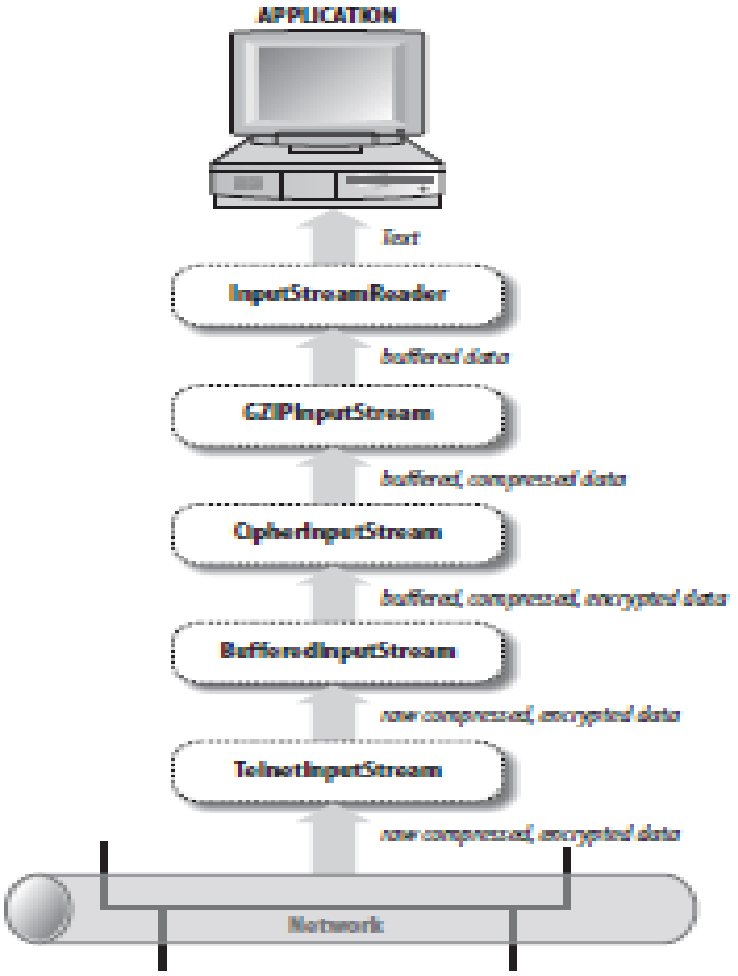

Input Stream
I n X a n a d u d i d
read(b)
Integer Variable
d
write(b)
I n X a n a d u d
Output Stream

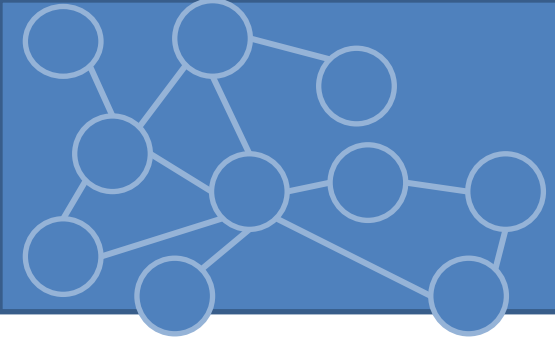


Filter Streams

- *InputStream* e *OutputStream* sono stream di basso livello. Il significato dei bytes viene lasciato al programmatore.
  - 32-bit big-endian integer, ASCII 7-bit, Latin-1, UTF-8, zip format
- Varie classi filtro a cui «attaccare» uno stream di byte
  - Filter streams: usano byte stream
  - Reader e Writer: testo usando differenti codifiche

- Reader messi in cima a filtri sui byte o altri reader
- Catena classica
- Filtering è interno: non vengono esposti nuovi metodi
- La concatenazione avviene attraverso i costruttori





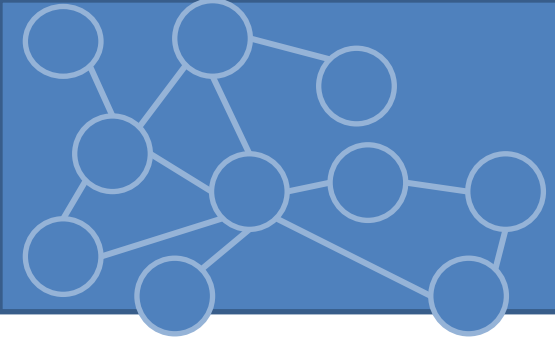
```
FileInputStream fis = new FileInputStream(‘’data.txt”);
BufferedInputStream bis = new BufferedInputStream(fin)
```

- Posso leggere sia da fis sia da bis, tuttavia chiamate alternate alla *read()* può violare contratti impliciti
- Uso l’ultimo filtro della catena per leggere perdendo il riferimento

```
InputStream is = new FileInputStream(‘’data.txt”);
is = new BufferedInputStream(is);
```

- Se voglio usare altri metodi non definiti nella super classe costruisco lo stream direttamente

```
DataOutputStream dos = new DataOutputStream(new
BufferedOutputStream(new FileOutputStream(‘’data.txt”)))
```

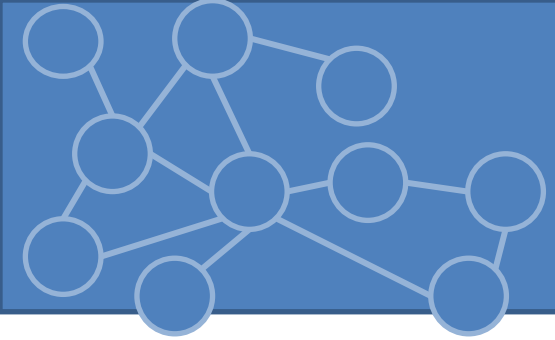


Buffered Streams

- ***BufferedOutputStream*** memorizza i dati in un buffer fino a che il buffer è pieno o viene invocato flush(). In questi casi scrive il contenuto del buffer nello stream sottostante
  - Utile nelle reti per limitare overhead dei protocolli
- ***BufferedInputStream***: quando viene invocata read prima legge da buffer, quando svuotato legge da stream sottostante
  - Efficiente per lettura da file

public BufferedInputStream(InputStream in)
public BufferedInputStream(InputStream in, int bufferSize)
public BufferedOutputStream(OutputStream out)
public BufferedOutputStream(OutputStream out, int bufferSize)

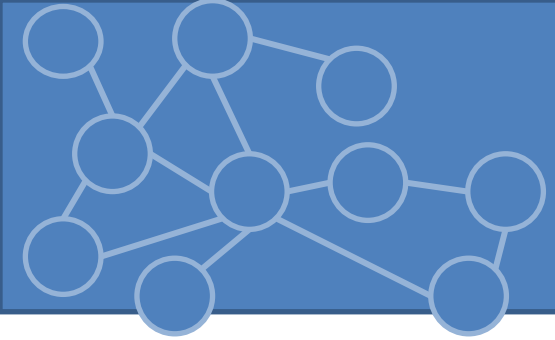
  - 2048 byte per lettura, 512 byte per scrittura
- *flush()* è obbligatorio quando voglio inviare i dati



PrintStream

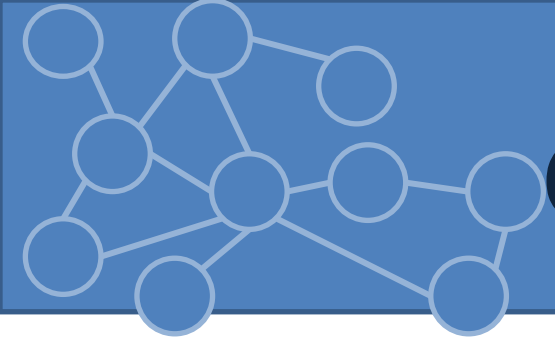
public PrintStream(OutputStream out)
public PrintStream(OutputStream out, boolean autoFlush)

- Se autoflush è settato, buffer svuotato appena viene aggiunto un array di byte, un linefeed o invocato il metodo println()
- 9 metodi di print e 10 di println che convertono argomento in una stringa e lo scrivono nello stream sottostante. I println aggiungono un separatore di linea che dipende dal SO (\r\n Windows, \n Unix, \r MacOS9)
- Difetti:
  - Dipende da piattaforma per quanto riguarda il separatore
  - Assume la codifica del testo della piattaforma in cui viene creato l'oggetto + non ci sono meccanismi per modificare la codifica.
  - Prende tutte le eccezioni -> eccezioni sono ordine del giorno per programmi di networking data instabilità dei collegamenti => unico meccanismo è flag di errore



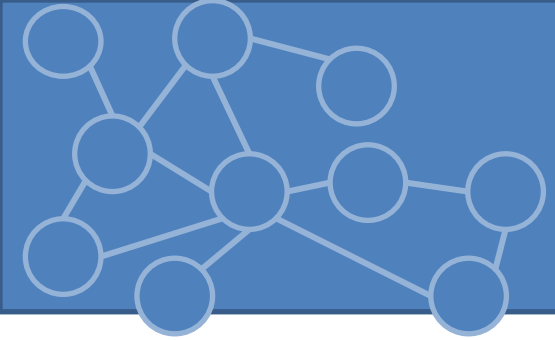
Data Streams

- ***DataInputStream*** e ***DataOutputStream*** forniscono metodi per leggere e scrivere i tipi primitivi di Java => scambio di dati tra programmi Java attraverso rete, file, pipe
  - Il formato degli interi è lo stesso del time protocol
  - Dati scritti con formato big-endian
- ***readFully(byte[] input)***: legge dati da input stream fino a che il numero richiesto di byte è stato letto altrimenti solleva IOException. Utile se campo Content-Lenght di un header HTTP è settato
- ***readLine()***: legge una linea di testo fino a un terminatore di linea.
- NB: viene riconosciuto solo \n o \r\n. Se riconosce un \r aspetta se dopo c'è un \n. Ma se \r è l'ultimo carattere dello stream rimane in attesa dell'ultimo carattere (capita con stream che arrivano da MacOS9). Il problema si presenta con connessioni persistenti



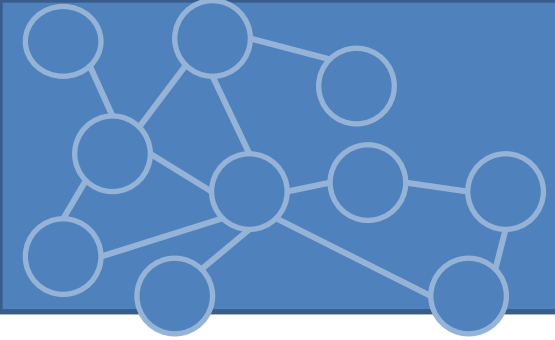
Compressing streams

- Nel **package java.util.zip** ci sono filter streams che comprimono gli streams in zip,gzip
  - Utili dato che HTTP 1.1 supporta il trasferimento di file compressi
- *DeflaterOutputStream, InflaterInputStream, GZIPOutputStream, GZIPInputStream, ZipOutputStream, ZipInputStream* che utilizzano essenzialmente lo stesso algoritmo + zip streams possono contenere più file
- Per comprimere attacco il filtro per la compressione allo stream che voglio comprimere. Poi scrivo e leggo normalmente
- Zip streams sono più complessi perchè possono contenere più elementi. Ogni file è un oggetto ZipEntry

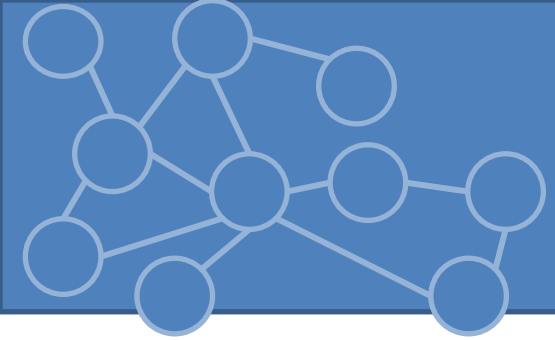


Readers e Writers

- Problema: non tutto il testo è codificato in ASCII o nel codice nativo della piattaforma di sviluppo
  - HTTP e altri protocolli usano una varietà di codifiche (Cirillico, Giapponese, Cinese)
  - Quando la codifica va oltre ASCII l'assunzione che 1 byte = 1 carattere cade!!
- Due classi astratte definiscono le API base per leggere e scrivere caratteri (Unicode)
  - ***java.io.Reader*** (lettura)
  - ***Java.io.Writer*** (scrittura)



- Le classi concrete più importanti sono
  - ***InputStreamReader***: traduce byte sottostanti in caratteri Unicode secondo la codifica specificata
  - ***OutputStreamWriter***: traduce caratteri in byte usando una codifica specificata e li scrive nello stream sottostante

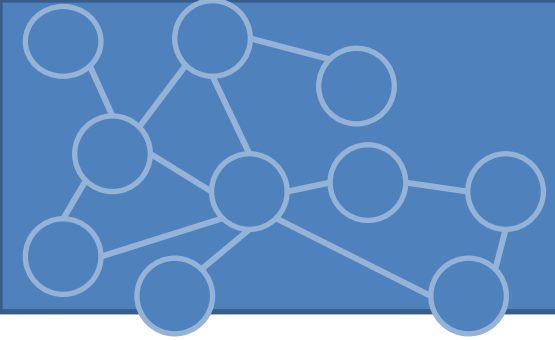


Writers

- La classe astratta ***Writer*** ha due costruttori protetti => mai usata direttamente

```
public abstract void write(char[] text, int offset, int length) throws IOException
public void write(int c) throws IOException
public void write(char[] text) throws IOException
public void write(String s) throws IOException
public void write(String s, int offset, int length) throws IOException
public abstract void flush( ) throws IOException
public abstract void close( ) throws IOException
```

- ***write(char[],int,int)*** è il metodo base su cui altri metodi implementati
- Il numero di byte che scrivo dipende da codifica
- Writers possono essere bufferizzati direttamente (chaining) o indirettamente (raw stream bufferizzato)



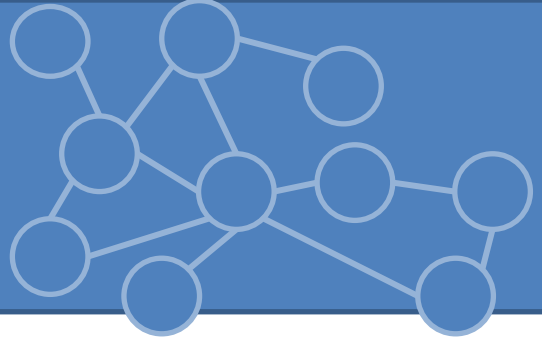
OutputStreamWriter

- Riceve caratteri da un programma Java e li converte secondo una codifica specificata

  public OutputStreamWriter(OutputStream out, String encoding) throws UnsupportedEncodingException

  public OutputStreamWriter(OutputStream out)

- Codifiche valide disponibili http://docs.oracle.com/javase/6/docs/technotes/guides/intl/encoding.doc.html
  - La codifica di default è quella della piattaforma su cui sviluppo
  - Posso reperire la codifica usando metodo *getEncoding()*



Readers

- ***Reader*** è una classe astratta speculare a *Writer*

public **abstract** int read(char[] text, int offset, int length) throws IOException

public int read( ) throws IOException -> int tra -1 e 65535

public int read(char[] text) throws IOException

public long skip(long n) throws IOException

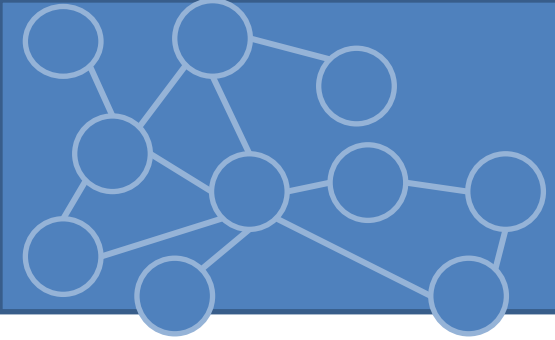
public boolean ready( )

public boolean markSupported( )

public void mark(int readAheadLimit) throws IOException

public void reset( ) throws IOException

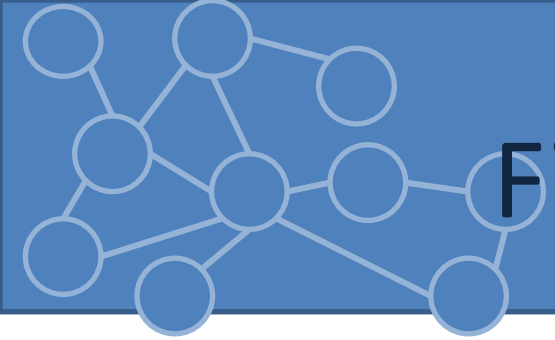
public **abstract** void close( ) throws IOException



InputStreamReader

- Legge i byte e li converte in caratteri secondo un’opportuna codifica

```
public InputStreamReader(InputStream in)
public InputStreamReader(InputStream in, String
encoding) throws UnsupportedEncodingException
```

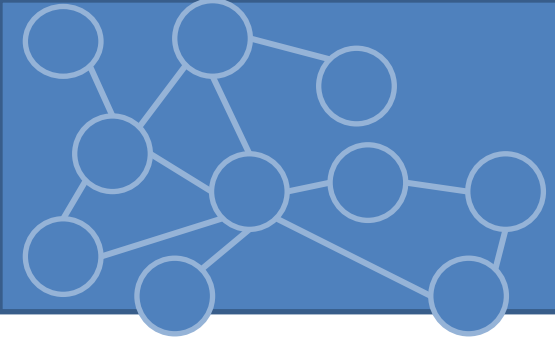


Filter Readers e Writers

- *InputStreamReader* e *OutputStreamWriter* portano uno stream di byte in uno stream di caratteri.
- Possono essere agganciati a filtri usando le sottoclassi di *FilterReader* e *FilterWriter*
  - ***BufferedReader***
  - ***BufferedWriter***
  - ***LineNumberReader***
  - ***PushbackReader***
  - ***PrintWriter***

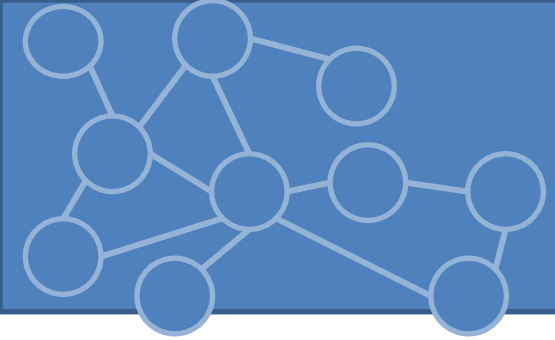
BufferedReader e BufferedWriter

- Usano un array interno di char come buffer (8192 caratteri di default)
- In lettura il testo è preso dal buffer, quando viene svuotato e riempito da altri caratteri provenienti dallo stream sorgente anche se non sono necessari (efficienza)
- In scrittura il testo viene posto in un buffer. Il testo viene inviato al relativo stream quando il buffer è pieno o viene eseguito un *flush()*
- *readLine()* non dipende più dalla piattaforma ma persiste il problema dell'attesa.

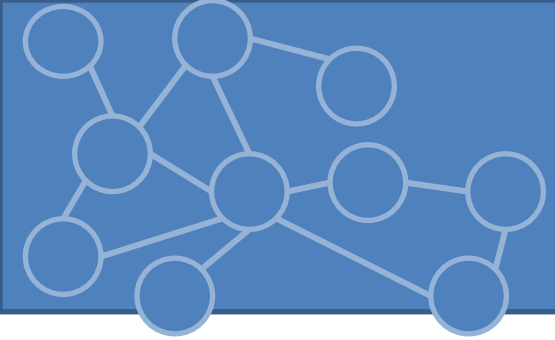


Esercizi

1. Si modifichi la classe *Output2* utilizzando la write(byte[]).
2. Si modifichi la classe *Output2* in modo tale per ogni linea alterni i primi 72 numeri pari e i primi 72 dispari.
3. Si crei una classe CopyByte che dispone di un metodo *static void copy(OutputStream os, InputStream is)* che implementa il ciclo visto nella slide. Il metodo legge byte a byte da **is** e scrive in **out.**
4. Si modifichi il codice in modo tale che la lettura avvenga byte a byte ma la scrittura sia bufferizzata. Il particolare il buffer viene svuotato quando è pieno o quando viene raggiunta la fine dello stream in input.



5. Si crei un metodo *static void copyBuffered(OutputStream os, Inputstream is)* che utilizza solo i metodi write(byte[],int,int) e e read(byte[],int, int)
6. Si testi la funzionalità dei metodi precedenti copiando diversi tipi di file (scaricati o creati)
7. Si testino le prestazioni dei metodi copy e copyBuffered usando un file di notevoli dimensioni
8. Si scriva il main della classe Pappagallo che ripete quanto scritto sulla console, utilizzando la classe *BufferedInputStream*.
9. Si scriva una classe che decomprima il file charGenerator.gz creato dalla classe Compressione vista a lezione.



10. Si modifichi il codice della soluzione dell'esercizio 4 della seconda lezione, sostituendo la classe *Scanner* con un *BufferedReader*
11. Si implementi il main della classe *ContaRighe*, che conta le righe di un file di testo.
12. Si legga il file Cirillico.txt utilizzando diverse codifiche per i caratteri e lo si riscriva utilizzando una codifica diversa da quello originale