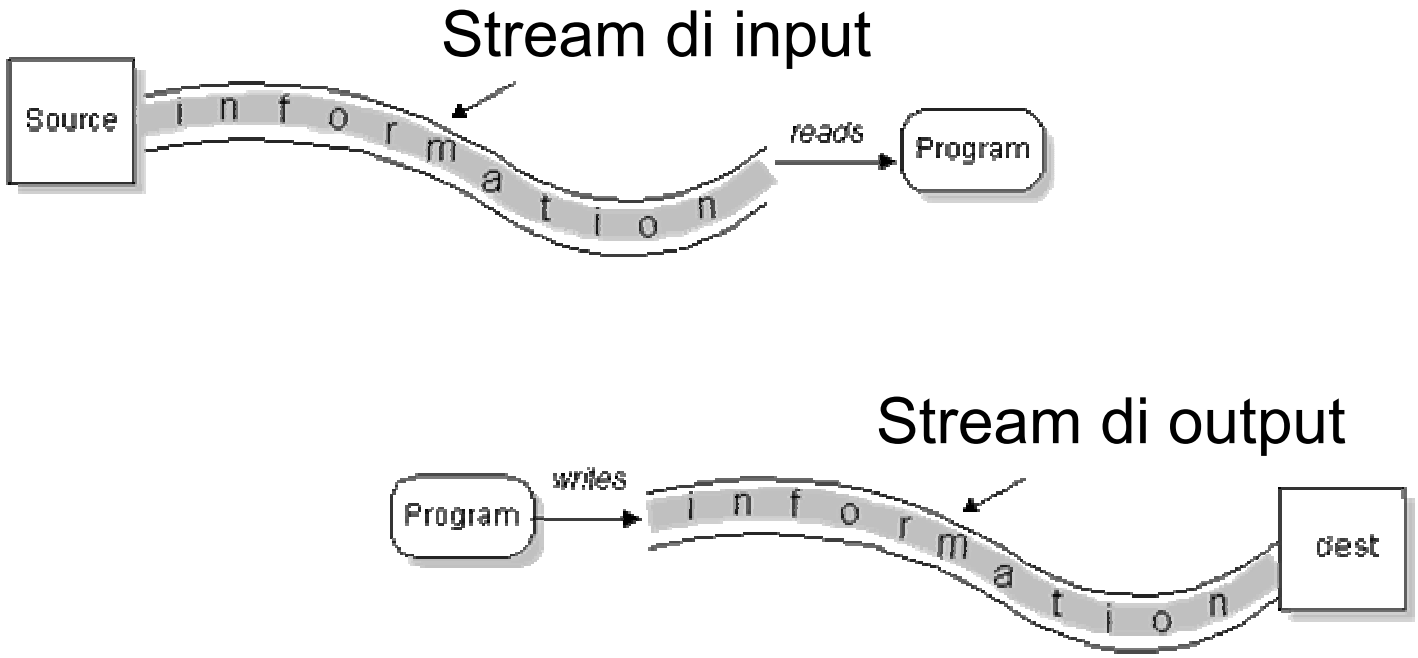
Input Output su Stream e File

In generale ogni programma ha un **flusso** (stream) di dati in input ed uno in output.



Il package **java.io** fornisce le classi, le interfacce e le eccezioni per gestire l'input e l'output dei programmi.

Il package mette a disposizione una serie di classi per trattare i file dal punto di vista del S.O. e un'altra serie per gestire la lettura e la scrittura di dati sui file.

Stream

Uno *stream* e' un'astrazione che produce o consuma informazioni. Uno stream e' collegato a un devide fisico.

Tutti gli stream si comportano allo stesso modo, anche se il device fisico a cui sono collegati e' diverso.

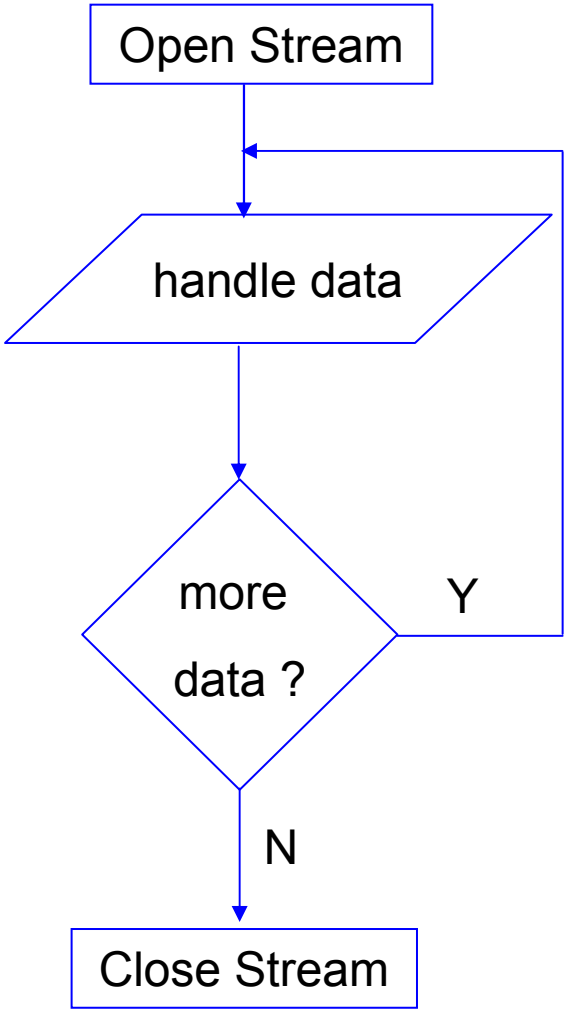
In questo modo, le stesse classe I/O e metodi possono essere applicate ad ogni tipo di device.

Per esempio, gli stessi metodi possono essere usati per scrivere nella console o un su un file di disco.

Stream

Gli Stream rappresentano flussi sequenziali di byte.

Tutti gli Stream vengono gestiti con algoritmi di questo tipo:



java.io

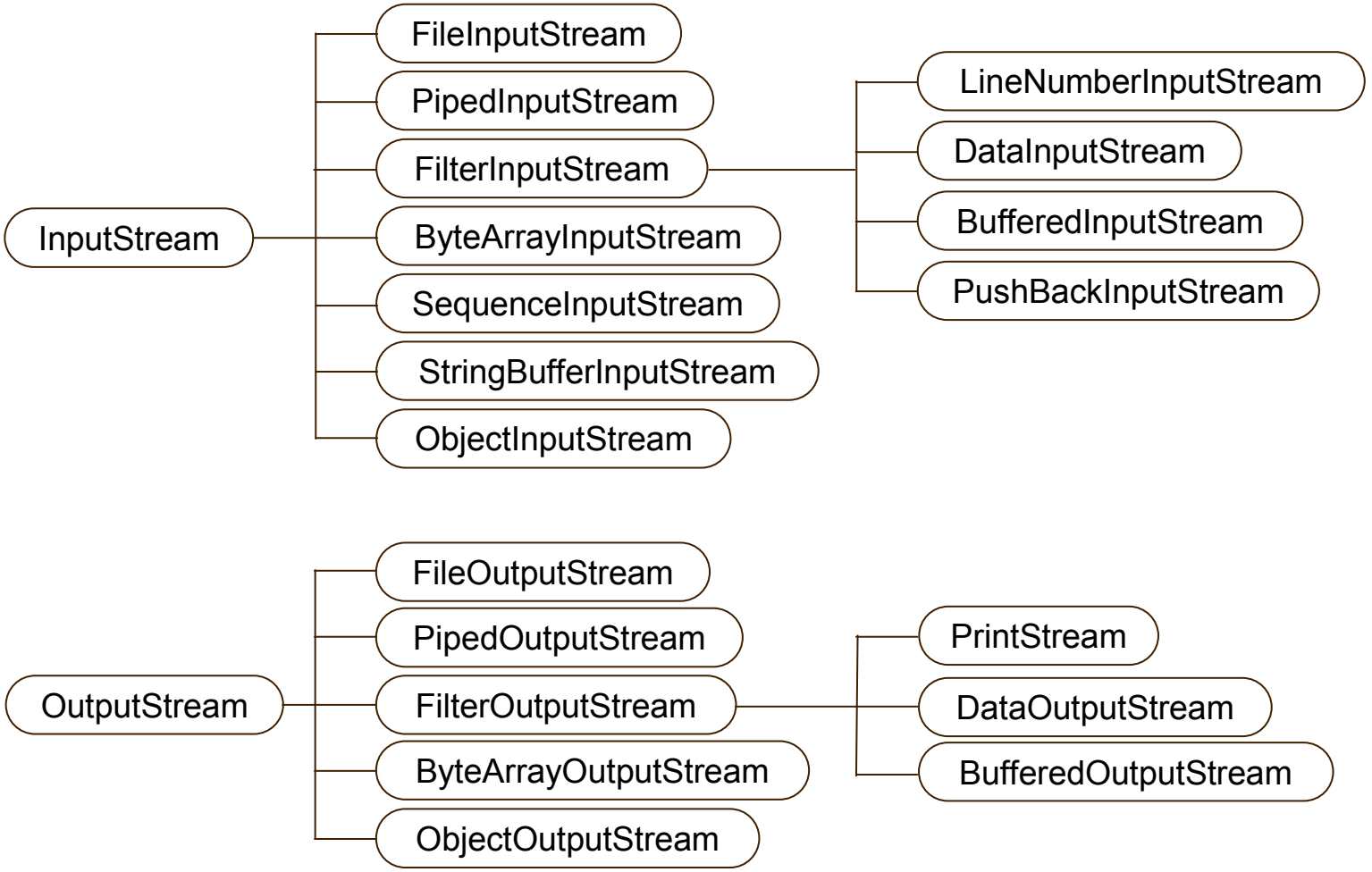
Java è uno dei pochi linguaggi che gestiscono i dati in formato **UNICODE** (16 bit).

`http://www.unicode.org/standard/translations/italian.html`

Il package **java.io** distingue due gerarchie di classi per la gestione degli stream:

- **byte stream** (8 bit - byte)
- **character stream** (16bit - char)

Il formato 8bit-byte è gestito tramite le classi **InputStream** e **OutputStream**



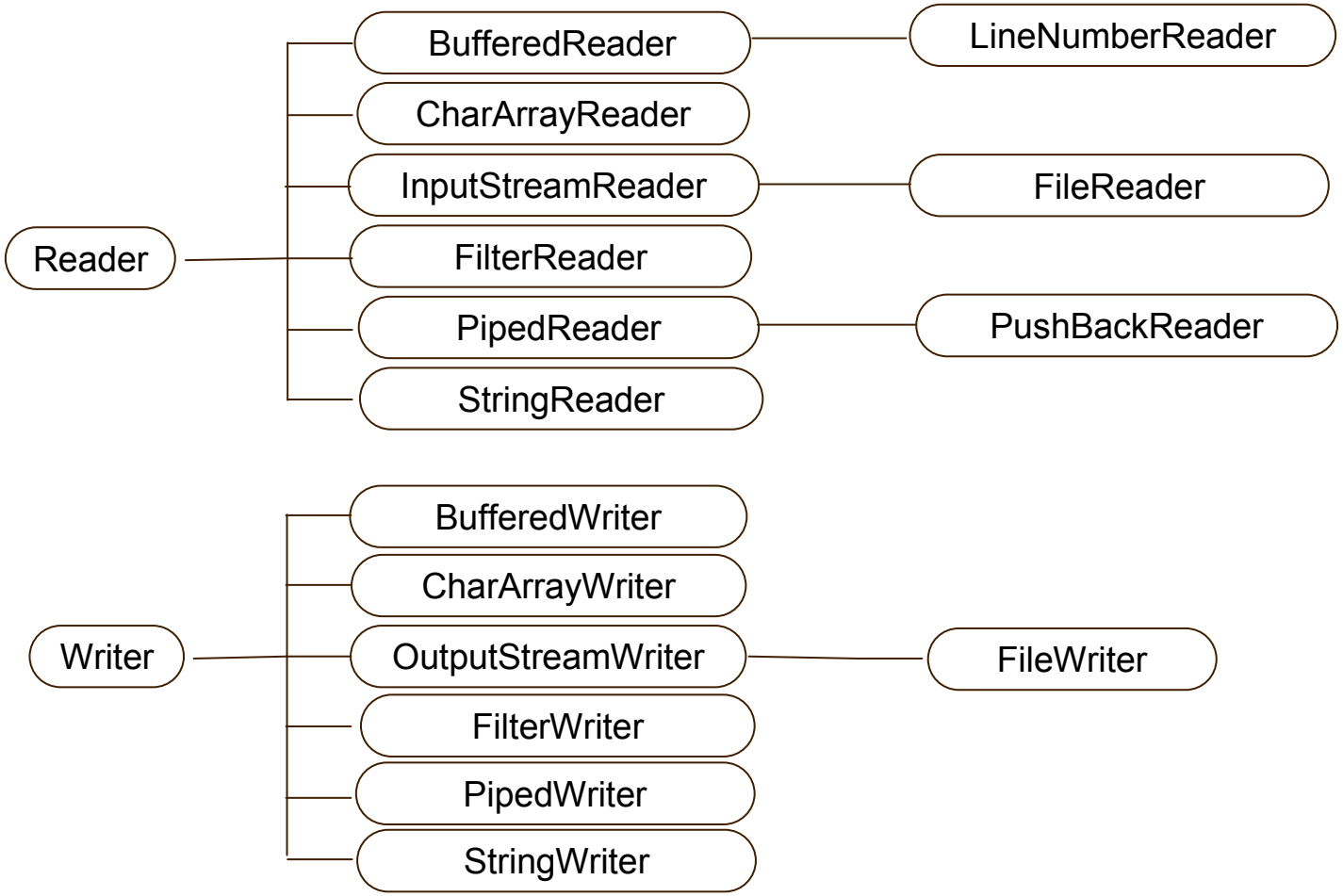
Le classi Byte Stream

BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Legge da un array di byte
ByteArrayOutputStream	Scrive su un array di byte
DataInputStream	Leggere i tipi standard di Java
DataOutputStream	Scrive tipi standard di Java
FileInputStream	Legge da un File
FileOutputStream	Scrive su un File
FilterInputStream	Implementa InputStream
FilterOutputStream	Implementa OutputStream
InputStream	Classe astratta che descrive uno stream in input
ObjectInputStream	Input stream per oggetti
ObjectOutputStream	Output stream per oggetti
OutputStream	Classe astratta che descrive stream in output

Le classi Byte Stream

PipedInputStream	**Input pipe**
PipedOutputStream	**Output pipe**
PrintStream	**Output stream che contiene print( ) e println( )**
PushbackInputStream	**Input stream che permette al byte di tornare nello stream**
RandomAccessFile	**Supporta random access file I/O**
SequenceInputStream	**Input stream che e' la combinazione di due o piu' input stream che saranno letti in modo sequenziale**

Il formato Unicode è gestito tramite le classi **Reader** e **Writer**



Character Stream Classes

BufferedReader

BufferedWriter

CharArrayReader

CharArrayWriter

FileReader

FileWriter

FilterReader

FilterWriter

InputStreamReader

LineNumberReader

OutputStreamWriter

PipedReader

PipedWriter

PrintWriter

PushbackReader

Reader Abstract

StringReader

StringWriter

Writer

Stream

Tipo di I/O	Classe	Descrizione
Memoria	CharArrayReader CharArrayWriter ByteArrayInputStream ByteArrayOutputStream	Lo stream coincide con un array
	StringReader StringWriter	Lo stream è una stringa
Pipe	PipedReader PipedWriter PipedInputStream PipedOutputStream	Lo stream coincide con una pipe. Le pipes sono canali di collegamento tra l'output di un thread e l'input di un altro thread
File	FileReader FileWriter FileInputStream FileOutputStream	Lo stream è un File

Stream

Tipo di I/O	Classe	Descrizione
Data Conversion	DataInputStream DataOutputStream	Convertono i dati da/in un formato indipendente dalla macchina
Buffering	BufferedReader BufferedWriter BufferedInputStream BufferedOutputStream	La lettura (scrittura) è bufferizzata, migliorando le prestazioni.
Filtering	FilterReader FilterWriter FilterInputStream FilterOutputStream	I dati vengono filtrati in base ad un filtro
Conversione Byte - Char	InputStreamReader OutputStreamWriter	

Input e Output su byte stream

InputStream e **OutputStream** consentono la lettura e scrittura di stream di byte.

I metodi definiti da queste due classi astratte sono disponibili a tutte le loro sottoclassi.

Quindi essi formano un insieme minimale di funzioni I/O che tutti gli stream di byte hanno.

I metodi in **InputStream** e **OutputStream** possono generare eccezioni **IOException**.

Input sequenziale

La classe **InputStream** consente di accedere ad uno stream di input in modo astratto.

Questa classe non ha metodi per leggere numeri o stringhe, ma può leggere soltanto singoli **byte**.

I metodi disponibili sono:

```
public void close()
public int available()
public long skip(long n)
public abstract int read()
public int read(byte buffer[])
public int read(byte buffer[], int offset, int length)
```

Input sequenziale di caratteri

La classe **Reader** si comporta analogamente.

I metodi disponibili sono:

```
public void close()
public int available()
public long skip(long n)
public abstract int read()
public int read(char buffer[])
public int read(char buffer[], int offset, int length)
```

Output sequenziale

Analogamente le classi **OutputStream** e **Writer** consentono di accedere ad uno stream di output in modo astratto.

I metodi disponibili per **OutputStream** sono:

```
public void close()

public int flush()

public abstract void write(int)//scrive un byte non c'e'
//bisogno di fare un cast.

public void write(byte buffer[])

public void write(byte buffer[], int offset, int length)
```

Output sequenziale

I metodi disponibili per **Writer** sono:

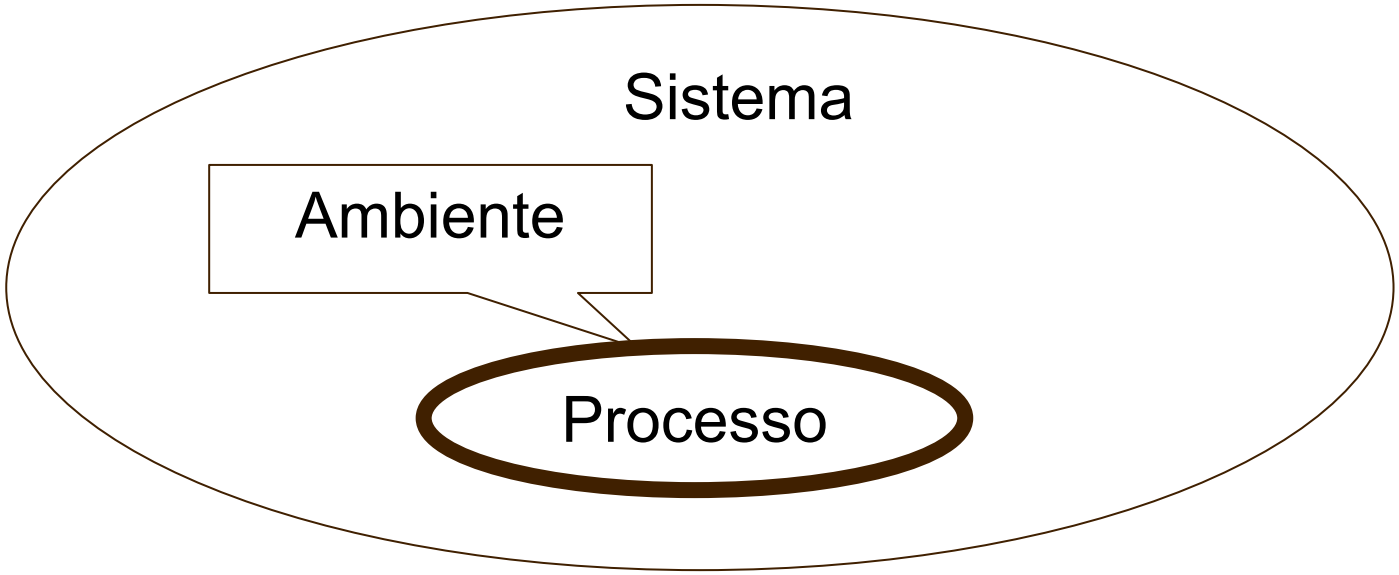
```
public void close()

public int flush()

public abstract void write(int)

public void write(char c[]) //scrive un array di
caratteri

public void write(char c[], int offset, int length)
//scrive lenght caratteri di c[] iniziando da offeset

public void write(String str)

public void write(String str, int offset, int length)
```

Risorse del Sistema

I programmi Java vengono eseguiti all'interno di un ambiente di lavoro (**environment**).

L'ambiente di lavoro corrisponde con la parte del **Sistema** visibile da parte del processo.

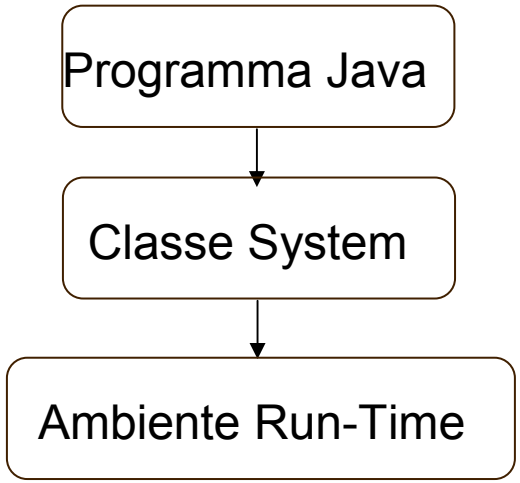
Il processo interagisce con il sistema esclusivamente tramite il suo ambiente di lavoro.



Risorse del Sistema

Il **Sistema** dispone di alcune **risorse**, che vengono messe a disposizione dell'utente tramite l'ambiente.

Java fornisce la classe **System** per accedere a tali risorse.



Stream Predefiniti

Tutti i programmi Java automaticamente importano **java.lang** package.

Questo pacchetto definisce una classe **System**, la quale contiene diversi aspetti dell'ambiente in run-time.

Per esempio essa contiene 3 variabili di stream predefiniti: **in**, **out**, e **err**.

Stream Predefiniti

A differenza delle altre classi, dalla classe **System** non è possibile istanziare oggetti; essa è una **final class** e tutti i suoi costruttori sono privati.

Le final class si usano direttamente come se fossero oggetti.

Tutti i metodi e le variabili della classe System sono **class methods** e **class variables** e sono dichiarati **static e public**. Questo significa che possono essere usate da qualsiasi parte del programma senza riferimento a un oggetto specifico.

La classe System

Per richiamare un **class methods** o accedere ad una **class variables** si usa la Java dot notation.

Ad esempio, per accedere alla variabile di classe **out** di **System** si scrive:

```
System.out
```

Analogamente si procede per accedere ai metodi di **System**:

```
System.getProperty(argument);
```

La classe System

Java, in accordo ai S.O. Unix-like, definisce tre stream per gli standard Input/Output:

System.out : standard output stream (console).

System.in: standard input, (tastiera).

System.err: standard error (console).

Stream Predefiniti

System.in e' un oggetto di tipo **InputStream**;
System.out e **System.err** sono oggetti di tipo **PrintStream**.

public static final InputStream in
public static final PrintStream out
public static final PrintStream err

Questi sono byte streams, anche se sono tipicamente usati per leggere e scrivere caratteri da e nella console.

Essi sono byte e no character streams perche' gli stream predefiniti erano parti delle specifiche originali di Java, che non conteneva ancora character streams.

E' possibile fare il wrap di questi in stream di caratteri.

La classe System

System.out e **System.err** si comportano allo stesso modo. I metodi più comuni sono:

- **print()**
- **println()**
- **write()**

I primi due consentono di scrivere stringhe, mentre il terzo viene utilizzato per l'output di byte.

La classe System

Notare che **print()** e **println()** sono simili:

```
System.out.print("Duke is not a penguin!\n");
```

è equivalente a:

```
System.out.println("Duke is not a penguin!");
```

La classe System

Esempio:

```
public class DataTypePrintTest
{ public static void main(String[] args)
  { String stringData = "Java Mania";
    char[] charArrayData = { 'a', 'b', 'c' };
    int integerData = 4;
    long longData = Long.MIN_VALUE;
    float floatData = Float.MAX_VALUE;
    double doubleData = Math.PI;
    boolean booleanData = true;
```

La classe System

```
        System.out.println(stringData);
        System.out.println(charArrayData);
        System.out.println(integerData);
        System.out.println(longData);
        System.out.println(floatData);
        System.out.println(doubleData);
        System.out.println(booleanData);
    }
}
```

L'output fornito è il seguente:

```
Java Mania
abc
4
-9223372036854775808
3.40282e+38
3.14159
true
```

Risorse di Sistema

La classe System consente di accedere alle seguenti risorse tramite il metodo **getProperty()**:

"file.separator"
"java.class.path"
"java.class.version"
"java.home"
"java.vendor"
"java.vendor.url"
"java.version"
"line.separator"
"os.arch"
"os.name"
"os.version"
"path.separator"
"user.dir"
"user.home"
"user.name"

Permette di determinare il nome del sistema operativo, la versione della Java Virtual Machine, il carattere che determina il carattere di separatore di file (‘/’). Consultate le API per le specifiche.

Input da tastiera

Allo standard input si accede tramite **System.in**.

Diversamente da **System.out**, lo standard input non prevede la gestione dei vari formati dei dati.

Le conversioni devono essere fatte in modo esplicito.

Input da tastiera

System.in e' un'istanza di **InputStream**, quindi si ha accesso automaticamente a tutti i metodi definiti da **InputStream**.

InputStream definisce un solo metodo di input **read( )**, legge bytes.

Ci sono 3 versioni di **read( ):**

int read( ) throws IOException //legge un singolo carattere. Ritorna -1 quando incontra la fine dello stream.

int read(byte *data*[ ]) throws IOException // Legge byte dal input stream e li mette in data fino a quando o l'array e' pieno o si e' raggiunto la fine dello stream, o vi e' stato un errore. Torna il num di bytes letti o -1 se si e' alla fine dello stream.

int read(byte *data*[ ], int *start*, int *max*) throws IOException //Legge input in data iniziando da start. Vengono immagazzinati fino a max byte. Torna il num di bytes letti o -1 se si e' alla fine dello stream.

Tutti i metodi generano **IOException** se si verifica un errore. Quando legge da **System.in**, premendo ENTER si genera una condizione di fine stream.

Esempio di lettura di un array di byte da System.in

```
// Read an array of bytes from the keyboard.
import java.io.*;

class ReadBytes {
    public static void main(String args[])
     throws IOException {
     byte data[] = new byte[10];
     System.out.println("Enter some characters.");
     System.in.read(data);
     System.out.print("You entered: ");
     for(int i=0; i < data.length; i++)
         System.out.print((char) data[i]);
    }
}
```

int read(byte *data*[ ])

Enter some characters:
Read Bytes
You entered:
Read Bytes

Esempio di lettura di un array di byte da System.in

```
// Read an array of bytes from the keyboard.
import java.io.*;

class ReadBytes {
    public static void main(String args[])
     throws IOException {
     byte data[] = new byte[10];
     System.out.println("Enter some characters.");
     System.in.read(data);
     System.out.print("You entered: ");
     for(int i=0; i < data.length; i++)
         System.out.print((char) data[i]+" ");
    }
}
```

int read(byte *data*[ ])

Enter some characters:
Read Bytes
You entered:
R e a d B y t e s

Esempio di lettura di un array di byte da System.in

```
import java.io.*;

class ReadBytes{

   public static void main(String args[])
     throws IOException {
     byte data[] = new byte[10];
     System.out.println("Enter some characters:");
     System.in.read(data,5,5);
     System.out.print("You entered: ");
     for(int i=0; i < 5; i++)
        data[i]='*';
     for(int i=0; i < data.length; i++)
        System.out.print((char) data[i]);
   }
}
```

int read(byte *data*[ ], int *start*, int *max*)

Enter some characters:
Read Bytes
You entered:
*****Read**

Esempio di lettura di un array di byte da System.in

```
import java.io.*;

class ReadBytes{

   public static void main(String args[])
     throws IOException {
     byte data[] = new byte[10];
     System.out.println("Enter some characters:");
   System.in.read(data,5,2);
     System.out.print("You entered: ");
     for(int i=0; i < 5; i++)
        data[i]='*';
     for(int i=0; i < data.length; i++)
        System.out.print((char) data[i]);
   }
}
```

int read(byte *data*[ ], int *start*, int *max*)

Enter some characters:
Read Bytes
You entered:
*****Re**

Esempio di lettura di un array di byte da System.in

```
import java.io.*;

class ReadBytes{

   public static void main(String args[])
     throws IOException {
     byte data[] = new byte[10];
     System.out.println("Enter some characters:");
   System.in.read(data,5,8);
     System.out.print("You entered: ");
     for(int i=0; i < 5; i++)
        data[i]='*';
     for(int i=0; i < data.length; i++)
        System.out.print((char) data[i]);
   }
}
```

int read(byte *data*[ ], int *start*, int *max*)

Enter some characters:
Exception in thread “main” java.lang.IndexOutOfBoundsException

Esempio di lettura di un array di byte da System.in

int read(byte *data*[ ], int *start*, int *max*)

```
import java.io.*;

class ReadBytes{

   public static void main(String args[])
     throws IOException {
     byte data[] = new byte[10];
     System.out.println("Enter some characters:");
  int num= System.in.read(data,5,2);
     System.out.print("You entered: ");
     for(int i=0; i < 5; i++)
        data[i]='*';
     for(int i=0; i < data.length; i++)
        System.out.print((char) data[i]);
System.out.printf("The number of readed chars is: %d",num);
}
}
```

```
Enter some characters:
Read Bytes
You entered:
*****Re
The number of readed chars is: 2
```

Esempio di lettura di un array di byte da System.in

int read()

```
import java.io.*;

class ReadBytes {

    public static void main(String args[])
            throws IOException {
            char data='a';
            while (data !='f')
            {
                    System.out.println("Enter a  character:");
                    data=(char) System.in.read();
                    System.out.print("You entered: ");
                    System.out.println(data);
            }
    }
}
```

```
Enter a character:
read
You entered: r
Enter a character:
You entered: e
Enter a character:
You entered: a
Enter a character:
You entered: d
Enter a character:
You entered:
Enter a character:
You entered:

Enter a character:
f
You entered: f
```

Output da tastiera System.out

System.out e' un byte stream.

Console output e' attuata con **print( )** and **println( ).**

Come avevamo detto questi metodi sono definiti dalla classe **PrintStream** che e' un output stream derivato da **OutputStream**, quindi implementa metodo **write( )**.

void write(int *byteval*) throws IOException

Non si usa spesso **write( )** ma si preferiscono **print( )** and **println( ).**

Si hanno anche i metodi **printf().** Danno un controllo dettagliato sul formato dei dati in output.

Es: Output da tastiera System.out

```
class WriteDemo {
    public static void main(String args[]) {
            Int b;
            b = 'X';
            System.out.write(b);
            System.out.write('\n');
    }
}
```

I File

I file sono sequenze ordinate di byte memorizzati su unità di memorizzazione di massa.

In generale, le funzionalità di accesso ai file prevedono due meccanismi di base:

- accesso **sequenziale** (stream)
- accesso **diretto** (random)

Al primo tipo appartengono anche oggetti in grado di gestire flussi di dati non legati direttamente ai file, come nel caso degli standard I/O.

I File

I file sono suddivisibili in due categorie:

- File di **Testo**
- File **Binari**

I **File di Testo** sono costituiti da caratteri dell'alfabeto, raggruppati in parole e righe. Sono visualmente interpretabili dall'utente.

I **File Binari** sono costituiti da byte di qualunque valore. La struttura interna è interpretabile solo tramite un apposito programma di gestione.

I File

Gli stream di Standard Input e Standard Output prima descritti sono due casi particolari di file.

Il Sistema Operativo tratta tutti gli stream come casi particolari di **File.**

Java fornisce un'astrazione del concetto di File, semplificandone l'utilizzo.

I File

In Java i file, come gli stream, sono gestiti tramite le classi presenti nella libreria standard **java.io**.

La prima operazione da fare consiste nell'aprire (**open**) il file. Questa operazione informa il S.O. che vogliamo accedere a quel determinato file.

Al termine di tutte le operazioni il file deve essere chiuso (**close**) per rilasciare le risorse impegnate e svuotare eventuali buffer.

Leggere e Scrivere un File usando un byte stream

Per creare un byte stream collegato a un file, si usa **FileInputStream** or **FileOutputStream**.

Per aprire un file si crea un oggetto di queste classi specificando il nome del file come argomento del costruttore.

Dopo aver aperto con successo il file allora si puo' leggere o scrivere nel file.

Input sequenziale da file

Come abbiamo detto la classe **FileInputStream** estende la classe **InputStream** per l'accesso sequenziale ai file.

In particolare vengono definiti i seguenti 2 costruttori:

```
public FileInputStream(String filename)
                        throws FileNotFoundException

public FileInputStream(File fp)
                        throws FileNotFoundException
```

I metodi a disposizione sono simili a quelli di **InputStream**.

Output sequenziale da file

La classe **FileOutputStream** estende la classe **OutputStream** per l'accesso sequenziale ai file.

```
public FileOutputStream(String filename)
                                throws IOException
public FileOutputStream(String filename, boolean append)
                                throws IOException
public FileOutputStream(File file)
                                throws IOException
public FileOutputStream(File file, boolean append)
                                throws IOException
```

I metodi a disposizione sono simili a quelli di **OutputStream**.

Leggere e Scrivere Dati binari

Abbiamo scritto e letto byte contenenti ASCII values.

Possiamo leggere e scrivere altri tipi di dati. Per esempio si possono creare file contenenti **int**s, **double**s, o **short**s.

Per leggere o scrivere valori binari di tipi primitivi si usano **DataInputStream** e **DataOutputStream**.

DataOutputStream implementa **DataOutput** interfaccia: essa definisce i metodi che scrivono tutti i tipi primitivi di Java.

Questi dati sono scritti usando il formato binario a noi “non comprensibile”.

Il costruttore: **DataOutputStream(OutputStream *outputStream*)**
outputStream e’ lo stream dove i dati verranno scritti.

Per scrivere in un file, si usa l’oggetto creato da **FileOutputStream** per questo parametro.

Metodi di Output definiti da DataOutputStream

void writeBoolean(boolean val)

void writeByte(int val)

void writeChar(int val)

void writeDouble(double val)

void writeFloat(float val)

void writeInt(int val)

void writeLong(long val)

void writeShort(int val)

Input

DataInputStream implementa **DataInput** interfaccia, la quale da' metodi per leggere i tipi primitivi di Java.

DataInputStream usa un'istanza di **InputStream**

DataInputStream(InputStream *inputStream*)

inputStream e' lo stream che e' legato all'istanza di **DataInputStream** che si sta creando.

Per leggere da un file si usa l'oggetto creato da **FileInputStream** per questo parametro.

Metodi di Input definiti da DataInput Stream

boolean readBoolean( )

byte readByte( )

char readChar( )

double readDouble( )

float readFloat( )

int readInt( )

long readLong( )

short readShort( )

Leggere e Scrivere Dati binari

Es in RWData.java. Il file su cui si scrive “testdata” e’ binario.

L’output del programma RWData.java
Writing 10
Writing 1023.56
Writing true
Writing 90.28

Reading 10
Reading 1023.56
Reading true
Reading 90.28

Stream basati sui caratteri

In cima alla gerarchia dei character stream ci sono le classi **Reader** e **Writer**.

Tutti i metodi generano **IOException** in presenza di errori.

I metodi definiti da queste due classi sono disponibili per tutte le sottoclassi.

Quindi, sono un insieme minimale di funzioni di I/O per gli stream di caratteri.

I metodi della classe Reader

abstract void close( )

int read( )

int read(char buffer[ ])

abstract int read(char buffer[ ],int offset,int numChars)

long skip(long numChars) //salta numChars caratteri //dell'input, e ritorna il numero di caratteri effettivamente //saltati

I metodi della classe Writer

Writer append(char ch) throws IOException //appende ch alla //fine dell'output stream

abstract void close( )

abstract void flush( )

void write(int ch)

void write(char buffer[ ]) //scrive l'array di caratteri sull'output //stream

abstract void write(char buffer[ ],int offset,int numChars)

void write(String str)

void write(String str, int offset,int numChars)//scrive una //sottostinga di str iniziando da offeset lunga numChars nello //stream di output

Input da tastiera

Le classi Reader e Writer serviranno a risolvere tre problemi:

1) vogliamo leggere **singole righe** di input e non generici flussi di byte;
2) In caso di errore è necessario gestire le **eccezioni** generate;
3) Dobbiamo essere in grado di leggere stringhe, e **numeri**, nei vari formati che conosciamo.

Input da console usando uno stream di caratteri

System.in e' un byte stream, quindi bisogna wrap **System.in** in un tipo di **Reader**.

La miglior classe per leggere da console input e' **BufferedReader**, la quale sopporta un buffered input stream.

Non si puo' costruire direttamente un **BufferedReader** da **System.in**. Si deve prima convertire in uno stream di caratteri.

Si usa **InputStreamReader**, che converte bytes in caratteri.

Per ottenere un **InputStreamReader** oggetto collegato a **System.in**, si usa:

InputStreamReader(InputStream *inputStream*)

Input da tastiera

Il primo passo consiste quindi nell'utilizzare la classe **InputStreamReader** che estende la classe Reader. Questa classe costituisce un ponte tra gli stream di byte e gli stream di caratteri.

I costruttori sono:

```
public InputStreamReader(InputStream IS)
Ci sono altri costruttori..consultare le API
```

Input da tastiera

Abbiamo detto che come **InputStream** possiamo utilizzare la classe **System.in**:

```
InputStreamReader reader = new InputStreamReader(System.in);
```

Il costruttore **InputStreamReader()** crea quindi un oggetto **reader** per leggere sequenze di caratteri a partire da **System.in**.

Input da tastiera

L'oggetto **reader** legge singoli caratteri.

La classe **BufferedReader** trasforma un *reader* in un lettore in grado di leggere intere righe.

```
BufferedReader(Reader in);
BufferedReader(Reader in, int size); //specifica la dimensione dell'input buffer.
```

```
InputStreamReader reader = new InputStreamReader(System.in);
BufferedReader console = new BufferedReader(reader);
```

Input da tastiera

Il metodo **readLine** della classe **BufferedReader** consente di leggere una singola riga di testo da tastiera.

```
Public String readLine() throws IOException
```

```
InputStreamReader reader = new InputStreamReader(System.in);
BufferedReader console = new BufferedReader(reader);
System.out.println("Inserisci una riga di testo");
String str = console.readLine();
```

Input da tastiera

La variabile **reader** può essere omessa.

Lo standard input **System.in** viene trasformato direttamente in un oggetto di tipo lettore bufferizzato.

```
BufferedReader console = new BufferedReader (
new InputStreamReader (System.in));

System.out.println("Inserisci una riga di testo");

String str = console.readLine();
```

Input da console usando uno stream di caratteri

Quindi, si usa l'oggetto prodotto da **InputStreamReader**, per costruire un **BufferedReader** nel seguente modo:

BufferedReader(Reader *inputReader*)

inputReader e' lo stream collegato ad un'istanza di **BufferedReader** che si sta creando.

Mettendo tutto insieme: **BufferedReader br = new BufferedReader(new InputStreamReader(System.in));**

br e' uno stream basato sui caratteri collegato alla console attraverso System.in

Leggere caratteri

Caratteri sono letti usando da **System.in** usando **read( )** definito da **BufferedReader**.

Ci sono 3 versioni di read() definiti da **BufferedReader:**

int read( ) throws IOException //legge un semplice Unicode carattere. Torna -1 se la fine dello stream e' raggiunta.

int read(char *data*[ ]) throws IOException // legge caratteri dallo stream di input e li mette in data fino a quando e' pieno o si raggiunge la fine del file o si incontra un errore. Torna il numero di caratteri letti o -1 alla fine dello stream.

int read(char *data*[ ], int *start*, int *max*) throws IOException //legge input in data inziando da start. Sono letti un massimo di max caratteri. Torna il numero di caratteri letti o -1 alla fine dello stream.

Es: Leggere Caratteri da tastiera

```
// Use a BufferedReader to read characters from the console.
import java.io.*;
class ReadChars {
    public static void main(String args[])
    throws IOException
    {
        char c;
        BufferedReader br = new
                BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter characters, period to quit.");
        // read characters
        do {
                c = (char) br.read();
                System.out.println(c);
        } while(c != '.');
    }
}
```

Leggere una stringa da tastiera

Per leggere una stringa da tastiera si usa **readLine( )** membro di **BufferedReader**.

String readLine( ) throws IOException

Torna un oggetto **String** che contiene i caratteri letti. Torna null se si cerca di leggere alla fine dello stream.

ES: Leggere una stringa da tastiera

```
// Read a string from console using a BufferedReader.

import java.io.*;

class ReadLines {
     public static void main(String args[])
     throws IOException
     {
          // create a BufferedReader using System.in

          BufferedReader br = new BufferedReader
                                   (new InputStreamReader(System.in));

          String str;
          System.out.println("Enter lines of text.");
          System.out.println("Enter 'stop' to quit.");

          do {
               str = br.readLine();
               System.out.println(str);
          } while(!str.equals("stop"));
     }
}
```

Questo esempio mostra l'uso di **BufferedReader** e del metodo **readLine( )**.

Il programma legge e mostra linee di testo fino a quando si entra la parola stop.

Output da console usando uno stream di caratteri

Si puo' usare **System.out** per scrivere nella console Java. E' preferibile usare **PrintWriter** stream.

PrintWriter e' una classe character-based.

PrintWriter(OutputStream *outputStream*, boolean *flushOnNewline*)
outputStream e' un oggetto di tipo **OutputStream** e *flushOnNewline* controlla se Java flushes l'output stream ogni volta che un **println( )** e' chiamato.

Se *flushOnNewline e'* **true**, allora il flushing e' automatico. Se **false**, flushing non e' automatico.

PrintWriter supporta **print( )** e **println( )** metodi per tutti i tipi in **Object**.
Cosi', usate questi metodi come in **System.out**.

Per scrivere nella console si usa **PrintWriter**, si specifica **System.out** per output stream e si flush lo stream dopo ogni chiamata a **println( )**.

Per esempio queste linee di codice creano un **PrintWriter** che e' alla console output.

```
PrintWriter pw = new PrintWriter(System.out, true);
```

Output da console usando uno stream di caratteri

```
// Demonstrate PrintWriter.
import java.io.*;
public class PrintWriterDemo {
     public static void main(String args[]) {

          PrintWriter pw = new PrintWriter(System.out, true);
          int i = 10;
          double d = 123.65;

          pw.println("Using a PrintWriter.");
          pw.println(i);
          pw.println(d);
          pw.println(i + " + " + d + " is " + (i+d));
     }
}
```

The output from this program is:

```
Using a PrintWriter.
10
123.65
10 + 123.65 is 133.65
```

File I/O usando uno stream di caratteri

Per costruire file I/O basati su caratteri si usano le classi **FileReader** e **FileWriter**.

Input da File

La classe **FileReader** estende la classe **InputStreamReader**. Viene usata per leggere in modo sequenziale da un file.

I costruttori sono:

```
public FileReader(String filename)
                throws FileNotFoundException


public FileReader(File fp)
```

Input da File

Normalmente si utilizza il seguente costruttore, al quale si passa il nome del file:

```
FileReader reader = new FileReader(“file.txt”);
```

Se il file non esiste viene generata un’eccezione **FileNotFoundException**

Input da File

Il metodo **read** restituisce un carattere del file per ogni sua invocazione.

Raggiunto il termine del file viene restituito il valore **-1**

```
Public int read() throws IOException
```

```
FileReader reader = new FileReader("file.txt");

while(true)
{ int i = reader.read();
  if (i == -1) break;
  char c= (char) i;
  …
}
```

Esempio 2

Il seguente programma mostra come leggere un file di testo. L'output è mandato sul video (standard output)

```
import java.io.*;

public class esempio02
{ public static void main(String[] args)
  { int n;
    boolean esci=false;
    ConsoleReader console = new ConsoleReader();//QUESTA
       //CLASSE LA vedremo alla fine della lezione
    while(!esci) {
       System.out.print("Inserisci il nome del file da
       leggere :");
       System.out.print(" ('FINE' per uscire) :");
       String nomefile = console.readLine();
       if ( nomefile.compareToIgnoreCase("FINE")==0)
              esci=true;
```

Esempio 2

```
    else
    {try
     { FileReader filetxt = new FileReader(nomefile);
       while ((n=filetxt.read()) !=-1)
         System.out.print( (char)n);
       filetxt.close();
        esci=true;
     }
     catch(FileNotFoundException e)
     { System.out.println(e);
       }
     catch(IOException e)
     { System.out.println(e);
       System.exit(1);
     }
    }
   }
  }
}
```

Output su File

La classe **FileWriter** estende la classe **OutputStreamWriter**. Viene usata per scrivere in modo sequenziale su un file.

I costruttori sono:

```
public FileWriter(String filename)
                  throws IOException
public FileWriter(String filename, boolean append)
                  throws IOException
public FileWriter(File file)
                  throws IOException
public FileWriter(File file, boolean append)
                  throws IOException
```

Esempio 3

Il seguente programma legge un file di testo e lo ricopia in un altro.

```
import java.io.*;

public class esempio03
{ public static void main(String[] args)
  { int n;
    boolean esci=false
    ConsoleReader console = new ConsoleReader();
    while(!esci)
 { System.out.print("Inserisci il nome del file di input :");
   System.out.print(“ (\”FINE\” per uscire) :”);
   String FileIn=console.readLine();
   System.out.print("Inserisci il nome del file di output:");
   String FileOut=console.readLine();

   if(nomefile.compareToIgnoreCase(“FINE”)==0) esci=true;
```

Esempio 3

```
    else
   { try
     {  FileReader filein = new FileReader(FileIn);
        FileWriter fileout = new FileWriter(FileOut);
        while ( (n=filein.read()) !=-1 )
        fileout.write( (char)n);
        filein.close();
        fileout.close();
        esci=true; }
     catch(FileNotFoundException e)
     { System.out.println(e); }
     catch(IOException e)
     { System.out.println(e);
       System.exit(1);}
      }
    }
  }
}
```

La classe FileWriter

FileWriter crea un **Writer** che si puo' usare per scrivere un file.

FileWriter(String *fileName*) throws IOException

FileWriter(String *fileName,* boolean *append*) throws IOException

fileName: path name del file.

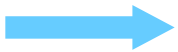
Se *append e'* **true**, allora l' output e' scritto alla fine del file. Altrimenti il file viene sovrascritto.

FileWriter e' derivato da **OutputStreamWriter** e **Writer**. Cosi', ha accesso ai metodi definiti in queste classi.

Es: FileWriter

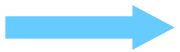
Legge linee di testo da input da tastiera e le scrive in un file “test.txt.” Il testo e’ letto fino a quando l’utente non scrive stop.

```
/* A simple key-to-disk utility that
demonstrates a FileWriter. */

import java.io.*;
class KtoD {
    public static void main(String args[])
    throws IOException {
        String str;
        FileWriter fw;

        BufferedReader br = new BufferedReader(
                            new InputStreamReader(System.in));
```



FileWriter

```
        try {
                fw = new FileWriter("test.txt");
        }
        catch(IOException exc) {
                System.out.println("Cannot open file.");
                return ;
        }
        System.out.println("Enter text ('stop' to quit).");

        do {
                System.out.print(": ");
                str = br.readLine();
                if(str.compareTo("stop") == 0) break;
                str = str + "\r\n"; // add newline
                fw.write(str);
        } while(str.compareTo("stop") != 0);

        fw.close();
    }
}
```



La classe FileReader

La classe **FileReader** crea un **Reader** per leggere il contenuto di un file.

FileReader(String *fileName*) throws FileNotFoundException

fileName: path name del file. Genera un **FileNotFoundException** se il file non esiste. **FileReader** e' derivato da **InputStreamReader** e **Reader**. Quindi, ha accesso a tutti i metodi definiti da queste classi.

ES:FileReader

Legge un file di testo “test.txt” e lo mostra in uno stream .

```
/* A simple disk-to-screen utility that demonstrates a FileReader. */

import java.io.*;
class DtoS {
    public static void main(String args[]) throws Exception {
            FileReader fr = new FileReader("test.txt");
            BufferedReader br = new BufferedReader(fr);

            String s;
            while((s = br.readLine()) != null) {
            System.out.println(s);
            }
            fr.close();
    }
}
```

FileReader e’ incorporato in un BufferedReader. Questo permette di usare readLine( ).

Riassumendo—Stream Filtrati

FileInputStream e FileOutputStream mettono a disposizione stream di input e di output collegati a un file di byte sul disco.

FileInputStream fin=new FileInputStream(“employee.dat”);

Si puo’ anche usare un oggetto File

File f = new File("employee.dat");
FileInputStream fin = new FileInputStream(f);

Per leggere o scrivere dati corrispondenti ai tipi primitivi di Java si usano le classi DataInputStream e DataOutputStream.

FileInputStream non ha metodi per gestire tipi numerici, DataInputStream non ha un metodo per acquisire i dati da un file.

Riassumendo—Stream Filtrati

Alcuni stream servono per ricavare i byte dai file e da sorgenti differenti, altri stream, per esempio DataInputStream o PrintWriter possono assemblare i byte in dati piu' utili.

Il programmatore deve combinare questi due elementi in quello che viene chiamato **stream filtrato**, impostando uno stream esistente nel costruttore dell'altro stream.

ES: FileInputStream fin = new FileInputStream("employee.dat");

DataInputStream din = new DataInputStream(fin);

double s = din.readDouble();

Riassumendo—Stream Filtrati

Gli stream non hanno un **buffer**. Ogni chiamata di read contatta il sistema operativo per chiedere un altro byte.

Per costruire uno stream bufferizzato si devono combinare piu' elementi usando uno **stream filtrato:**

Es: **DataInputStream din = new DataInputStream(new BufferedInputStream(new FileInputStream("employee.dat")));**

Riassumendo—Stream Filtrati

Quando si legge l'input a volte e' necessario leggere il byte successivo per vedere se si tratta di un valore corrispondente a quello desiderato. In java si usa **PushbackInputStream**

PushbackInputStream pbin = new PushbackInputStream(new BufferedInputStream(new FileInputStream("employee.dat")));

Per leggere il byte successivo: int b = pbin.read();

Per rifiutarlo se non corrisponde al byte voluto:

```
if (b != '<') pbin.unread(b);
```

Per leggere il byte successivo e tipi numerici:

```
DataInputStream din = new DataInputStream( new
PushbackInputStream( new BufferedInputStream( new
FileInputStream("employee.dat"))));
```

Riassumendo—Stream Filtrati

La possibilita' di combinare le classi di filtro per costruire sequenze di stream veramente utili consente tuttavia di avere una grande flessibilita'.

Per esempio si possono leggere i numeri da un file ZIP compresso usando la sequenza di stream:

```
ZipInputStream zin = new ZipInputStream(new
FileInputStream("employee.zip"));
DataInputStream din = new DataInputStream(zin);
```

Stream di File ZIP

Le classi che gestiscono file ZIP si trovano in java.util.zip

Anche se non fanno parte di Java.io, le classi GZIP e ZIP sono sottoclassi di java.io.FilterInputStream e java.io.FilterOutputStream.

Ogni file ZIP ha un'intestazione che contiene informazioni quali il nome del file e il metodo di compressione utilizzato.

Stream di File Input ZIP

- Per leggere un file ZIP si utilizza uno ZipInputStream combinato con un FilterInputStream.
- Si cercano le vari voci nell'archivio con il metodo **getNextEntry** il quale restituisce un oggetto **ZipEntry** che corrisponde alle voci.
- Il metodo **read** di **ZipInputStream** viene modificato per restituire -1 alla fine della voce corrente.
- Si chiama **closeEntry** per leggere la voce successiva.

ES: Stream di File Input ZIP

Leggere un file zip

```
ZipInputStream zin = new ZipInputStream(new
                          FileInputStream(zipname));
ZipEntry entry;
while ((entry = zin.getNextEntry()) != null)
{
   analyze entry;
   read the contents of zin;
   zin.closeEntry();
}
zin.close();
```

ES: Stream di File Input ZIP

Leggere il contenuto di una voce zip

```
BufferedReader in = new BufferedReader(new
      InputStreamReader(zin));
String s;
while ((s = in.readLine()) != null)
   do something with s;
```

Stream di File Output Zip

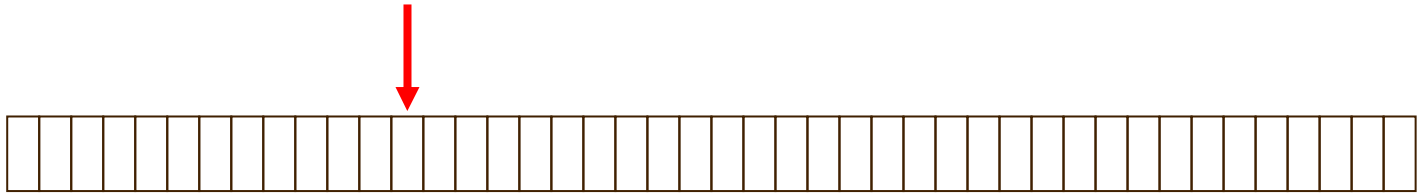
- Analogamente si costruisce uno stream legato ad un file di output ZipOutputStream
  - Per ogni voce che si vuole creare nel file zip si crea un oggetto zipEntry.
  - Si chiama putNextEntry per iniziare a scrivere un nuovo file.
  - Si inviano i dati del file allo stream ZIP.
  - Una volta terminato di scrivere si chiama closeEntry.

```
FileOutputStream fout = new FileOutputStream("test.zip");
ZipOutputStream zout = new ZipOutputStream(fout);
for all files
{
   ZipEntry ze = new ZipEntry(filename);
   zout.putNextEntry(ze);
   send data to zout;
   zout.closeEntry();
}
zout.close();
```

Accesso Random ai file

La classe **RandomAccessFile** consente di accedere direttamente in un qualunque punto di un File.

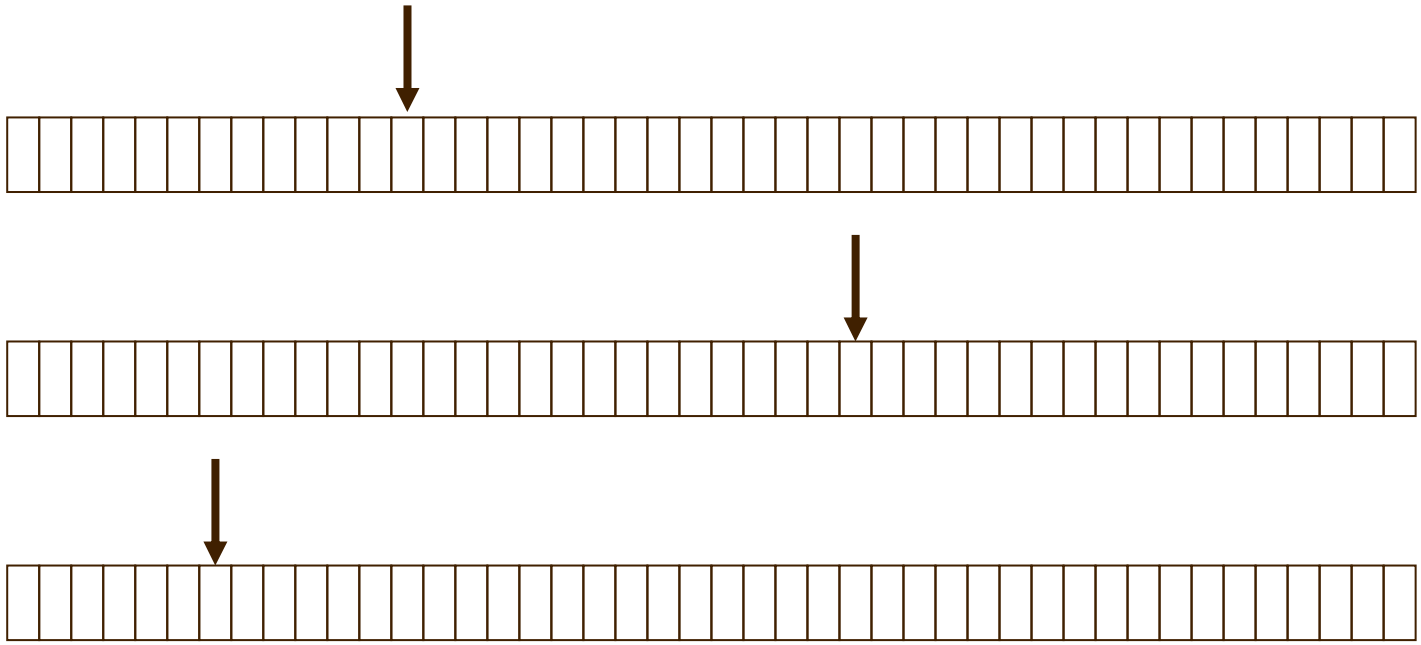
Il file viene visto come sequenza di byte, con un indice (file pointer) che identifica la posizione per la successiva operazione di I/O.



Dopo una operazione di I/O, la posizione del file pointer viene aggiornata.

Accesso Random ai file

Il File **pointer** può essere spostato dall'utente in una qualsiasi posizione all'interno del file.



Accesso Random ai file

RandomAccessFile non e' derivata da **InputStream** o **OutputStream**. Essa implementa l'interfaccia **DataInput** and **DataOutput**, la quale definisce i metodi di I/O.

I costruttori sono:

```
public RandomAccessFile(String name, String mode)
                                throws FileNotFoundException
public RandomAccessFile(File fp, String mode)
                                throws FileNotFoundException
```

Accesso Random ai file

I modi possibili sono:

- "**r**": aperto solo in lettura;
- "**rw**": aperto in lettura e scrittura;

Se il modo è errato viene lanciato l'eccezione

`IllegalArgumentException`

Accesso Random ai file

I metodi:

void close() //Chiude il radom access file e rilascia ogni

//risorsa associata ad esso

long length() //Ritorna la lunghezza del file

long getFilePointer() //Ritorna l'offset attuale del file

void seek(long pos) //Set il file-pointer offset, misurando

//dall'inizio del file al quale la prossima lettura o

//scrittura deve avvenire

int skipBytes(int n) //Salta n bytes di input

Accesso Random ai file

I metodi di lettura:

```
int read()//Legge un byte dal file

int read(byte b[]) //Legge b.lenght byte da questo file  e li mette in un array di byte

int read(byte b[], int off, int len) //Legge len byte dal file e li mette in un array iniziando da off

boolean readBoolean()

byte readByte()

char readChar() //Legge un Unicode carattere

double readDouble()

float readFloat()

int readInt()

String readLine() //Legge la prossima linea di testo dal file

short readShort() // Legge un 16-bit con segno numero
```

Accesso Random ai file

I metodi di scrittura:

```
void write(byte b[]) //Scrive b.lenght bytes dall'array
//al file iniziando dalla posizione corrente del file
//pointer.

void write(byte b[], int off, int len) //Scrive len byte
//dall'array iniziando da offset di questo file

void write(int b) // Scrive il byte b nel file
```

Accesso Random ai file

I metodi di scrittura:

```
void writeBoolean(boolean v)
void writeByte(int v)
void writeBytes(String s) //scrive la stringa in un file
                          //come sequenza di bytes
void writeChar(int v)
void writeDouble(double v)
void writeFloat(float v)
void writeInt(int v)
void writeLong(long v)
void writeShort(int v)
```

Es: Accesso Random ai file

RandomAccessDemo.java

Il programma scrive 6 double in un file e poi li legge in un ordine non sequenziale

L'output:

First value is 19.4

Second value is 10.1

Fourth value is 33.0

Here is every other value:

19.4 123.54 87.9

Accesso Random ai file

Tutti i metodi precedenti possono lanciare l'eccezione `IOException`.

Inoltre nei S.O multiutente è necessario gestire l'eccezione `SecurityException`, generata da accessi non consentiti per motivi di protezione.

StringTokenized e testo delimitato

Quando si legge una riga di input, si acquisisce una singola stringa lunga, che si vuole suddividere in stringhe individuali (**token**).

La classe da utilizzare StringTokenizer class in java.util.

Bisogna specificare uno o piu' delimitatore (es. '|'), per default " \t\n\r", (space, tab, newline, e carriage return)

StringTokenizer tokenizer = new StringTokenizer(line, "|");

StringTokenizer tokenizer = new StringTokenizer(line, "|,;");

```
while (tokenizer.hasMoreTokens())
{
   String token = tokenizer.nextToken();
   process token
}
```

StringTokenized e testo delimitato

Un’alternativa a StringTokenizer e’ il metodo di suddivisione **split** della classe String.

line.split("[|,;]") ritorna un array String[] con tutti i token presenti nella stringa usando come delimitatori presenti tra le parentesi quadre.

Stream di Oggetti

Se si desidera salvare file che contengono dei dati strutturati si utilizza il meccanismo di **serializzazione** degli oggetti che ne automatizza quasi completamente la gestione.

La classe che si desidera leggere/scrivere come uno stream di oggetti deve implementare l'interfaccia `Serializable`:

```
Class xxxxx implements Serializable { … }
```

L'interfaccia `Serializable` non dispone di alcun metodo, per cui non serve modificare la classe in alcun modo.

Scrittura di oggetti

Per salvare i dati di un oggetto è necessario aprire un oggetto `ObjectOutputStream`:

```
ObjectOutputStream out = new
ObjectOutputStream (new
FileOutputStream(nomefile, true/false);
```

Il valore del parametro booleano indica o meno la condizione di *append.*

Per salvare un oggetto si utilizza il metodo `WriteObject:`

```
xxxxx p1 = new xxxxx(…);
Out.WriteObject(p1);
```

Lettura di Oggetti

Per rileggere gli oggetti bisogna innanzitutto avere un oggetto **ObjectInputStream**:

```
ObjectInputStream in = new ObjectInputStream
(new FileInputStream(nomefile);
```

Per recuperare gli oggetti nello stesso ordine in cui sono stati scritti si utilizza il metodo `ReadObject()`:

```
xxxxx  p1 = (xxxxx) in.ReadObject()
```

La chiusura di uno stream sia di input che di output si realizza mediante il metodo `close()`

Esempio

Lettura/Scrittura di un file strutturato: “Data la classe ***Person***, contenente informazioni anagrafiche degli studenti di questo corso, realizzare un programma che generi un file strutturato contenente 100 records.

Prevedere una fase di lettura e di stampa a video del contenuto del file.

La gestione dei File

Abbiamo visto come leggere e scrivere i dati in un file.

Le classi Stream riguardano appunto il contenuto dei file.

La classe **File** incapsula le funzionalita' necessarie per lavorare con il file system del computer dell'utente.

La classe **File** ha a che fare con la memorizzazione del file sul disco.

La classe FILE

La classe File e gli attributi di directory e file, consentono di accedere alle directory, verificare e acquisire le caratteristiche dei file presenti, cambiarne gli attributi, cancellare e rinominare file.

Il metodo costruttore `File(String nome)`

- fornisce un oggetto file con il nome assegnato posto nella directory corrente.
- Una chiamata di questo costruttore non crea un file con il nome assegnato, se il file non esiste
- Se si vuole creare un nuovo file si deve usare createNewFile della classe File.

Il metodo costruttore `File(String pathname, String name)`

- fornisce un oggetto file con il nome assegnato posto nella directory specificata

La classe File

Il metodo costruttore `File(File dir, String name)`

- fornisce un oggetto file con il nome assegnato posto nella directory specificata
- Se dir e' null allora lo crea nella directory corrente

La classe FILE

`boolean exists()`	Verifica l'esistenza del file con cui è invocata.
`boolean canRead()`	Verifica se il file è di sola lettura.
`boolean canWrite()`	Verifica se il file ha i permessi di lettura e scrittura.
`boolean isDirectory()`	Verifica se si tratta di Directory
`boolean isFile()`	Verifica se si tratta di File
`boolean isHidden()`	Verifica se il File è nascosto.
`String getAbsolutePath()`	Restituisce una stringa con il path completo del file.
`String getName()`	Restituisce una stringa con il nome del File.
`String getParent()`	Restituisce una stringa con il solo path del file.
`String[] list()`	Restituisce un array di stringhe con tutti i file e le sottodirectory se invocato con un nome di directory.

La classe FILE

Metodi di settaggio

`boolean createNewFile()`	Crea un file vuoto. Il boolean è true se l'operazione ha avuto successo.
`boolean delete()`	Cancella il file.
`void deleteOnExit()`	Setta il file per la sua cancellazione all'uscita dal programma.
`boolean mkdir()`	Crea una directory
`boolean renameTo(File)`	Rinomina il file invocante con il nome del parametro.
`boolean setReadOnly()`	Setta il file in sola lettura. Il boolean è true se l'operazione è riuscita.

Esempio File01

Il programma acquisisce due stringhe corrispondenti al nome di una Directory e ad una estensione e stampa la lista dei file della Directory che hanno l'estensione desiderata

```
import java.io.*;
class esempio_File01 {
public static void main(String arg[]) {
    String nome="", est="";
    boolean c=false;
    if (arg.length>=1) nome=arg[0];
    if (arg.length>=2) est=arg[1];
    File f= new File(nome);
    if (!f.exists())
     System.out.println("Non esiste la directory "+nome);
```

Esempio File01

```
File a[]=f.listFiles();

    for (int i=0; i<a.length;i++) {
        String s=a[i].toString();
        s=s.substring(s.length()-4);
        boolean b=(s.equalsIgnoreCase(est));
        if  ((a[i].isFile()) && b) {
            c=true;
            System.out.println(a[i].getName());
            }
        }
    if (!c) System.out.println("Nessun file trovato....");
   }
}
```

La classe ConsoleReader

```
import java.io.*;
public class ConsoleReader
{ private BufferedReader reader;
  public ConsoleReader()
  {reader=newBufferedReader(new InputStreamReader(System.in));
  }

  public String readLine()
  { String inputLine="";
     try
     { inputLine = reader.readLine();
     }
     catch(IOException e)
     { System.out.println(e);
       System.exit(1);
     }
     return inputLine;
  }
```

La classe ConsoleReader

```
    public int readInt ()
  { String inputString = readLine();
     int n = Integer.parseInt(inputString);
     return n;
  }

  public double readDouble ()
  { String inputString = readLine();
     double x = Double.parseDouble(inputString);
     return x;
  }
}
```

Esempio 1

Il seguente programma mostra come utilizzare la classe appena descritta:

```
public class esempio01
{ public static void main(String[] args)
  { int i;
    ConsoleReader console = new ConsoleReader();
    System.out.print("inserisci un numero intero :");
    int n=console.readInt();
    System.out.print("inserisci un numero reale  :");
    double x=console.readDouble();
    for(i=1;i<=n;i++)
      System.out.println(x*i);
  }
}
```

Input e Output—La classe Scanner

Per visualizzare un messaggio sullo stream di output basta chiamare il metodo System.out.println

Prima di JDK 5.0 non esisteva un metodo altrettanto semplice per leggere l'input.

Per Leggere l'input dalla console si deve creare uno **Scanner** collegato allo "stream di input standart" System.in.

Scanner in = new Scanner(System.in);

Input—La classe Scanner

Per leggere una linea di input:

```
System.out.print("What is your name? ");
String name = in.nextLine();
```

Per leggere una singola parola

```
String firstName = in.next();
```

Per leggere un intero

```
System.out.print("How old are you? ");
int age = in.nextInt();
```

Formattare l’output

System.out.printf("Hello, %s. Next year, you'll be %d", name, age);

d **decimale**

x **esadecimale**

b **boolean**

c **carattere**

s **String**

Ambienti di Sviluppo e Debugging

Il Debugger

Man mano che la struttura di un programma si fa più complessa (utilizzo di molteplici **classi** e **gerarchie**) sono sempre maggiori le probabilità di commettere degli **errori**.

Gli errori di **sintassi** vengono rilevati direttamente dal compilatore.

Scovare gli errori a ***run-time*** (manualmente) può diventare un'impresa ardua.

Il Debugger

La maniera più semplice per “esplorare” il comportamento di un programma durante la sua esecuzione è quella di stampare sul terminale i valori delle variabili “**critiche**”.

Uno strumento di lavoro che consente di “investigare” un programma in modo semplice e pulito è il ***debugger***: Uno strumento software che permette di individuare i "**bugs**".

Il ***debugger*** a sua volta manda in esecuzione appositi programmi in maniera da sorvegliarne il funzionamento fermandone per esempio l'esecuzione in un dato punto e analizzandone il contenuto rispetto a *variabili*, *classi*, *threads*, ecc.

Esempi di debugger

JDB – Debugger a linea di comando. E' distribuito insieme all'SDK JAVA;

JSWAT – Debugger grafico, rilasciato sotto licenza GNU – Sito web e documentazione:

`http://www.bluemarsh.com/java/jswat/`

JDB

Il debugger Java. E' necessario compilare con **javac -g** . Per farlo partire digitare: **jdb Nomeclass -** Le principali istruzioni sono:

Per inserire un ***breakpoint*** all'inizio del main:

```
stop in Nomeclasse.main
```

Per inserire un ***breakpoint*** all'inizio di un metodo:

```
stop in Nomeclasse.nomemetodo
```

Per far partire l'esecuzione:

```
run
```

Per far continuare l'esecuzione:

```
cont
```

Per listare le istruzioni vicino a quella in esecuzione:

```
list
```

JDB

Per listare l'istruzione n e seguenti:

```
list n
```

Per creare un breakpoint alla riga n:

```
stop at Nomeclasse:n
```

Per stampare una variabile:

```
print variabile
```

Per stampare tutte le variabili locali:

```
locals
```

Per eseguire la prossima istruzione:

```
step
```

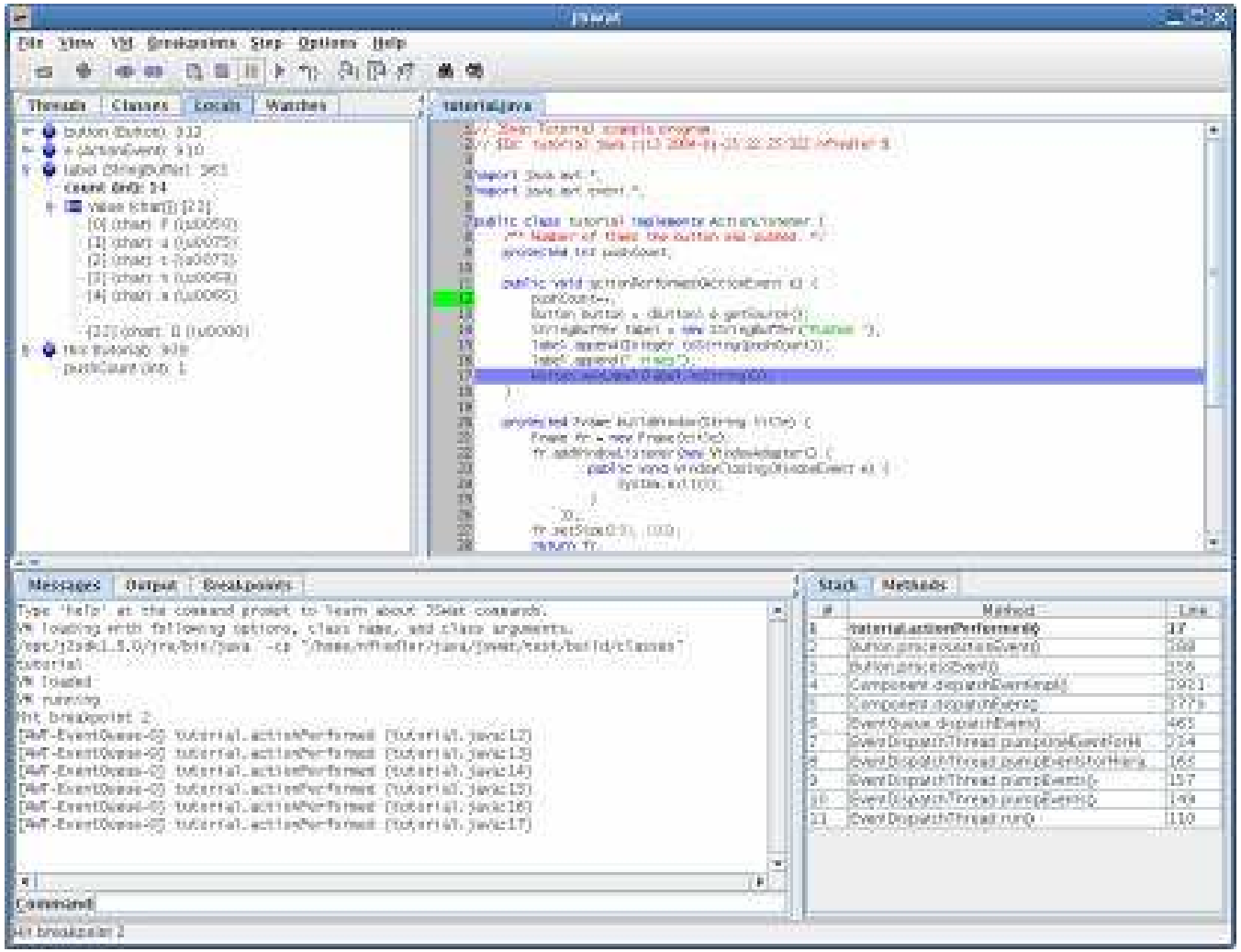
Per eseguire la prossima istruzione senza saltare all'interno di un metodo:

```
next
```

Altri comandi….

`threads [threadgroup]`	elenca i thread
`thread` *`thread_id`*	imposta al thread desiderato
`suspend [`*`thread_id`* `(s)]`	sospende il/i thread
`resume [`*`thread_id`* `(s)]`	riprende thread sospesi
`dump` *`nome/i`*	stampa tutte le informazioni dell'oggetto
`clear` *`classe:linea`*	elimina il breakpoint alla linea
`!!`	ripete l'ultimo comando (unix-like)
`help anche ?`	fornisce la sintassi dei comandi del debugger
`exit`	per uscire

JSWAT


JSwat
File View VM Breakpoints Step Options Help
Threads Classes Locals Watches
tutorial.java
Messages Output Breakpoints
Stack Methods
Command

IDE –Ambienti di Sviluppo

Spesso risulta utile affidarsi ad un ambiente di sviluppo completo, in grado di offrire oltre al debugging integrato anche una serie di servizi aggiuntivi quali per esempio:

- editing avanzato;
- supporto multi-classe;
- *wizard* per interfacce grafiche;
- generazione automatica della documentazione;
- **…**

In generale tali software prendono il nome di IDE (**Integrated Development Environment**).

IDE –Ambienti di Sviluppo

NetBeans – *open source*;

`http://www.netbeans.org/index_it.html`

Jbuilder ($$)

`http://www.borland.com/jbuilder/`

JCreator ($$) - `http://www.jcreator.com/index.htm`

Eclipse (free) - `http://www.eclipse.org/`

L’ambiente di sviluppo

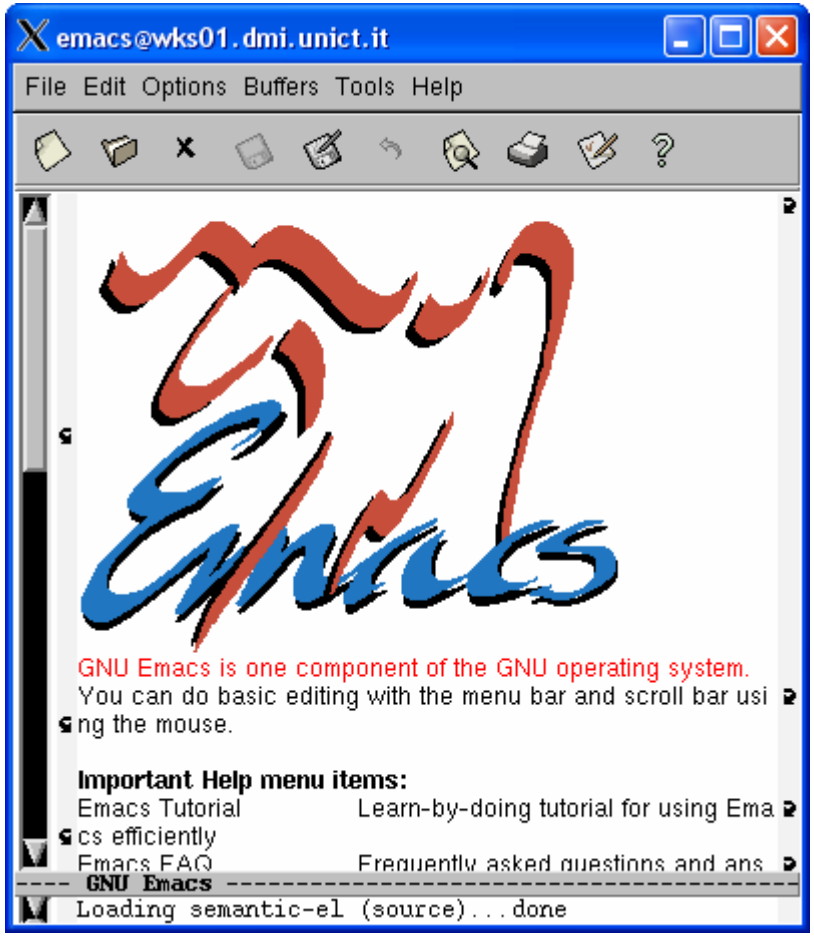
Un editor:

In ambiente Linux troverete diversi ambienti testo quali Vi o Emacs o Scite (XEmacs).

Emacs e’ un ambiente integrato mediante il quale si può anche effettuare la compilazione ed il debugging del programma.

Emacs

Da shell digitare il comando
`emacs [nome file]`
se questo è seguito dal simbolo **&**
Il sistema manda l'esecuzione del comando in background ritornando il controllo del prompt all'utente



Emacs

Emacs, sviluppato originariamente da Richard Stallman nel 1976, è un editor molto popolare fra i programmatori.

In Emacs distinguiamo due tipologie di comandi generali:

CONTROL “CTRL-x” comando

ESC-x meta-comando

(alternativamente ALT-x)

Alcuni comandi Emacs

Undo	`CTRL-x u` **oppure** `CTRL-_`
Salva il file	`CTRL-x CTRL-s`
Salva con nome diverso	`CTRL-x CTRL-w` ***nome***
Apre un nuovo file	`CTRL-x CTRL-f` ***nome***
Inserisce un file	`CTRL-x i` ***nome***
Passa ad un altro buffer	`CTRL-x b`
Chiude un buffer	`CTRL-x k`
Divide la finestra in due	`CTRL-x 2`
Passa da una metà all'altra	`CTRL-x o`
Riunifica la finestra	`CTRL-x 1`
Refresh della finestra	`CTRL-l`
Quit da `emacs`	`CTRL-x CTRL-c`
Cursore a fine riga	`CTRL-e`
Cursore a inizio riga	`CTRL-a`
Cursore giù una pagina	`CTRL-v`
Cursore su una pagina	`ESC v`

Inizio del buffer	`ESC <`
Fine del buffer	`ESC >`
Vai alla linea...	`ESC x goto-line` ***numero***
Cerca testo	`CTRL-s` ***testo***
Sostituisce testo	`ESC %` ***testo1 testo2***
Marca inizio di un blocco	`CTRL-SPACE`
Marca fine blocco e taglia	`CTRL-w`
Marca fine blocco e copia	`ALT-w`
Incolla blocco	`CTRL-y`
Pagina di aiuto	`CTRL-h CTRL-h`
Significato di un tasto	`CTRL-h k` ***tasto***
Significato di tutti i tasti	`CTRL-h b`
Apre una shell dentro `emacs`	`ESC x shell`
Aiuto psicologico	`ESC x doctor`

Alcuni comandi Emacs

Undo	CTRL-x u oppure CTRL-_
Salva il file	CTRL-x CTRL-s
Salva con nome diverso	CTRL-x CTRL-w *nome*
Apre un nuovo file	CTRL-x CTRL-f *nome*
Inserisce un file	CTRL-x i *nome*
Passa ad un altro buffer	CTRL-x b
Chiude un buffer	CTRL-x k
Divide la finestra in due	CTRL-x 2
Passa da una metà all'altra	CTRL-x o
Riunifica la finestra	CTRL-x 1
Refresh della finestra	CTRL-l
Quit da emacs	CTRL-x CTRL-c
Cursore a fine riga	CTRL-e
Cursore a inizio riga	CTRL-a
Cursore giù una pagina	CTRL-v
Cursore su una pagina	ESC v

Inizio del buffer	ESC <
Fine del buffer	ESC >
Vai alla linea...	ESC x goto-line *numero*
Cerca testo	CTRL-s *testo*
Sostituisce testo	ESC % *testo1 testo2*
Marca inizio di un blocco	CTRL-SPACE
Marca fine blocco e taglia	CTRL-w
Marca fine blocco e copia	ALT-w
Incolla blocco	CTRL-y
Pagina di aiuto	CTRL-h CTRL-h
Significato di un tasto	CTRL-h k *tasto*
Significato di tutti i tasti	CTRL-h b
Apre una shell dentro emacs	ESC x shell
Aiuto psicologico	ESC x doctor

EMACS

emacs@WKS01

File Edit Options Buffers Tools Java Help

```
/**
 * La classe HelloWorld implementa
 * un'applicazione che semplicemente
 * visualizza "Hello World!" nello
 * standard output.
 */

class HelloWorld {
        public static void prova(String[] args) {
                System.out.println("Hello World!");
        }
}
```

-1\-- **HelloWorld.java** (Java Abbrev)--L1--All----------

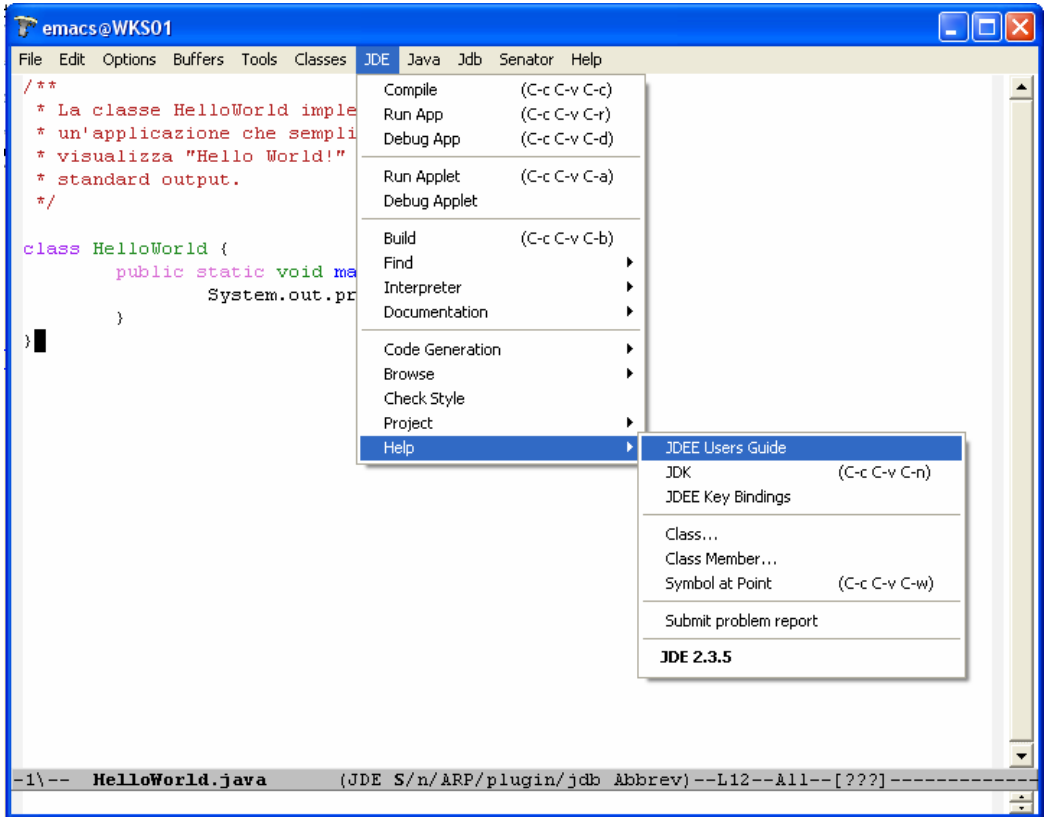
Font-Lock mode enabled

Hello World

Usando la combinazione di tasti “ALT-x” si
Accede ai meta-comandi Emacs.

Digitando: font-lock-mode
Si abilita la colorazione del testo

JDE



JDEE (Java Development Environment for Emacs) è un pacchetto Emacs scritto in linguaggio Lisp che consente ad Emacs di interaggire con tool per lo sviluppo di applicazioni Java. Come risultato JDE rende Emacs un ambiente di sviluppo integrato (integrated development environment (IDE)).

Eclipse

Java - ApplicationServer.java - Eclipse SDK

File Edit Source Refactor Navigate Search Project Run Window Help

Java

Package Explorer

satellite
src
(default package)
ApplicationServer.java
attribDialog.java
BrowserControl.java
ChooseMenuItem.java
CoordGeo.java
elencoStradeDialog.java
ExampleFileFilter.java
grafoStradale.java
graphEdge.java
graphPointNode.java
Heap.java
IdentifyResult.java
imageCanvas.java
impostDialog.java
infoFrame.java
JnlpProxy.java
Latitudine.java
layerDialog.java
LettoreNMEA.java
Longitudine.java
myFrame.java
mySimulazione.java

*(default package).ucd

ApplicationServer... | CoordGeo.java

```
// University of Catania

import java.net.Socket;

public class ApplicationServer implements
    private  int port = 4333;
    private  int MAXCONNECTS = 100;
    private int connects = 0;
    Hashtable utenti;
    Thread thread;
    ServerSocket server_socket;

    public ApplicationServer() {
```

Outline

import declarations
ApplicationServer
port : int
MAXCONNECTS : ir
connects : int
utenti : Hashtable
thread : Thread
server_socket : Se
ApplicationServer(
setHashtable(Hash
start()
run()
serviceClientReque
utentiMobili()

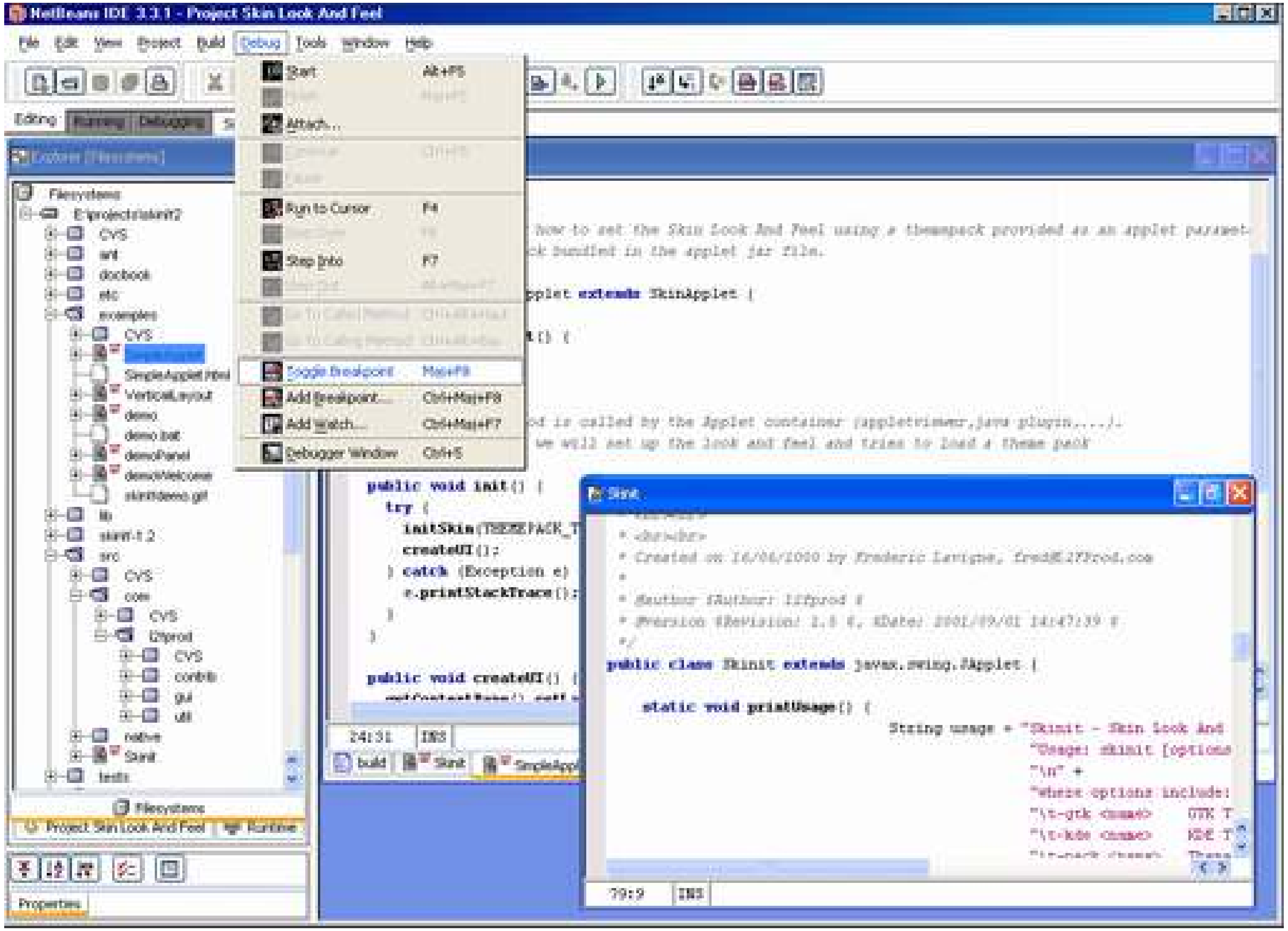
Problems | Javadoc | Declaration | Console

0 errors, 0 warnings, 0 infos

Description	Resource	In Folder	Location

Writable | Smart Insert | 6 : 1

NetBeans


NetBeans IDE 3.3.1 - Project Skin Look And Feel
File Edit View Project Build Debug Tools Window Help
Start Alt+F5
Attach...
Run to Cursor F4
Step Into F7
Toggle Breakpoint
Add Breakpoint... Ctrl+Maj+F8
Add Watch... Ctrl+Maj+F7
Debugger Window Ctrl+5
Editing
Filesystems
CVS
ant
docbook
etc
examples
SimpleApplet.html
VerticalLayout
demo
demo.bat
demoPanel
demoWelcome
skinitdemo.gif
lib
src
com
l2fprod
contrib
gui
util
native
Skinit
tests
Project Skin Look And Feel
Runtime
Properties
public void init() {
try {
createUI();
} catch (Exception e) {
e.printStackTrace();
public void createUI() {
24:31
INS
build
public class Skinit extends javax.swing.JApplet {
static void printUsage() {
String usage = "Skinit - Skin Look And
79:9

Ancora esempi ed esercizi

...

Esempio 4

Il seguente programma è simile all'esempio 3, ma gestisce l'eccezione di **FileNotFound**.

```
import java.io.*;

public class esempio04
{ public static void main(String[] args)
  { ConsoleReader console = new ConsoleReader();
    FileReader filein=null;
    boolean continua = true;
```

Esempio 4

```
    do
    { continua=true;
      try
      {  System.out.print("Inserisci  il  nome  del  file  di
input :");
        String FileIn=console.readLine();
        filein = new FileReader(FileIn);
      }
      catch(FileNotFoundException e)
      { System.out.println("Il file non esiste");
        continua=false;
      }
    } while(!continua);
```

Esempio 4

```
    try
    { int n;
      System.out.print("Inserisci il nome del file di output
:");
      String FileOut=console.readLine();
      FileWriter fileout = new FileWriter(FileOut);
      while ((n=filein.read()) !=-1)
        fileout.write( (char)n);
      filein.close();
      fileout.close();
    }
    catch(IOException e)
    { System.out.println(e);
      System.exit(1);
    }
  }
}
```

Esempio 5

Il seguente programma ricopia un file in un altro in ordine inverso.

```
import java.io.*;
// legge un file carattere per carattere e lo trascrive
// in ordine inverso nel file di destinazione


public class esempio05
{ public static void main(String[] args)
  { ConsoleReader console = new ConsoleReader();
    RandomAccessFile filein=null;
    RandomAccessFile fileout=null;
    boolean continua;
```

Esempio 5

```
do
   { continua=false;
     System.out.print("Inserisci ilnomedel file di input:");
     String FileIn=console.readLine();
     try
     { filein = new RandomAccessFile(FileIn, "r");
     }
     catch(FileNotFoundException e)
      { System.out.println("Il file non esiste");
        continua=true;
      }
   } while(continua);
```

Esempio 5

```
System.out.print("Inserisci il nome del file di output :");
    String FileOut=console.readLine();
    try
    { fileout = new RandomAccessFile(FileOut, "rw");
    }
    catch(IOException e)
    {
      System.out.println("Errore nell'apertura del file di
  output");
      System.out.println(e);
       System.exit(1);
    }
```

Esempio 5

```
try
     { long pos;
       int n;
        for ( pos =filein.length();pos>0; --pos)
                  {
                   filein.seek(pos-1);
                   n = filein.read();
                   fileout.writeByte((char) n);
                  }
       filein.close();
        fileout.close();
     }
     catch(IOException e)
     { System.out.println(e);
       System.exit(1);
     }
   }
}
```

Esercizi

Creazione di un file - Esercizio 1: "*Realizzare un programma che generi in un ciclo 150 numeri da 1000 a 1149 e dopo averli trasformati in stringa li scriva su una singola riga del file.*"

Appendere righe a un file - Esercizio 2: *"Realizzare un programma che aggiunga al precedente file di testo altri 50 numeri da 1150 a 1199 e li scriva in coda al file esistente."*

In entrambi i casi prevedere un metodo stampa() a video per la verifica;

File Dati – Esercizio 3: "*Realizzare un programma che generi in un ciclo 45 numeri da 10000 a 10044 e li scriva su un file di dati.*"