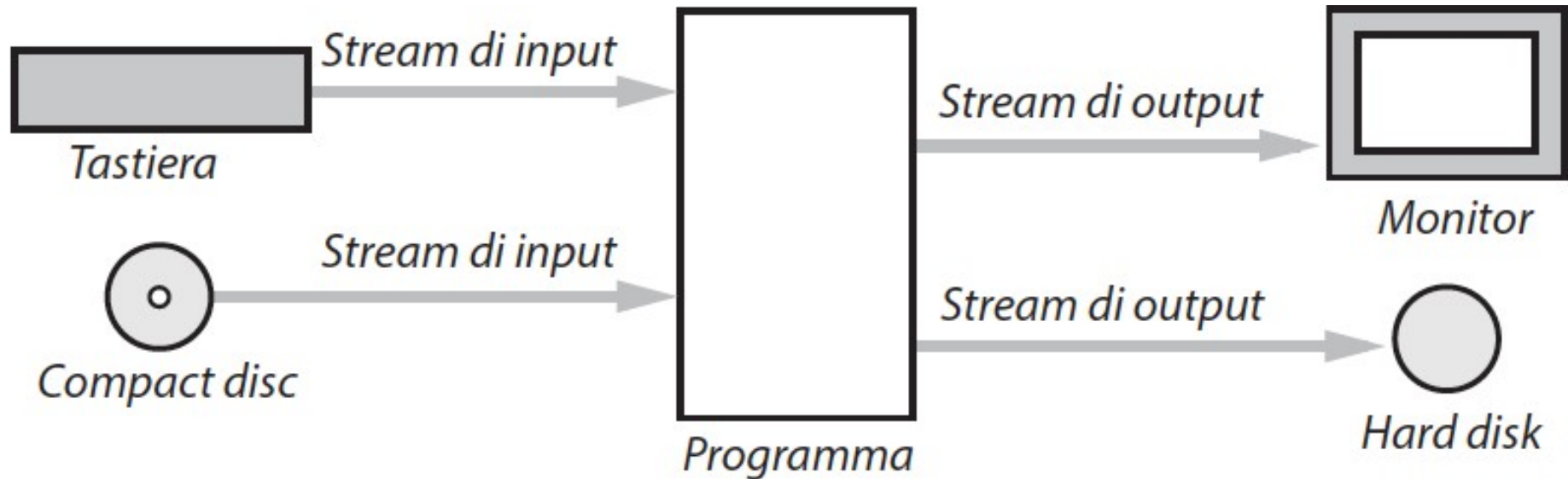

I/O: Files and Streams

Dal materiale di Stefano Ferretti
s.ferretti@unibo.it

Stream di input/output



I/O da tastiera

- Abbiamo visto finora come **leggere** valori di input da tastiera usando 2 classi:
 - **Scanner**
 - `BufferedReader`
- Leggere da tastiera è utile ma talvolta scomodo
 - Ad es. per inserire **manualmente** migliaia di elementi
 - Ogni volta che rilancio l'applicazione, devo **reinserire** da capo tutti gli elementi
 - Magari gli stessi *identici* elementi...

I/O da file

- *Alternativa*: posso leggere i dati da un **file testuale** che li elenca *uno per riga*
 - ...o in qualche altro modo
 - Se voglio modificare gli elementi, modifico *solamente* il file di input
 - Specifico *nell'applicazione* Java il file da leggere
 - L'applicazione *non cambia* se cambio dati di input
 - **Separazione** logica applicazione / dati input

Leggere da file

- L'approccio generale per **leggere** gli elementi di un file in modo sequenziale è:

Apri il file in **lettura** // **Eccezione** se file non esiste

Leggi un “data item” // Byte, linea, record

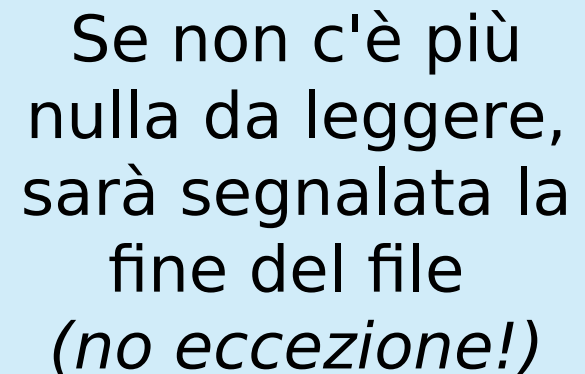
WHILE not EndOfFile DO

Processa il data item

Leggi il *prossimo* data item

END WHILE

Chiudi file



Se non c'è più
nulla da leggere,
sarà segnalata la
fine del file
(no eccezione!)

Scrivere su file

- L'approccio generale per **scrivere** su file è:

Apri il file in **scrittura** // **No errore** se il file non esiste

WHILE ci sono dati da scrivere **DO**

Scrivi il prossimo dato

END WHILE

Chiudi il file

- Quando si apre un file in scrittura, di default:
 - Se il file esiste, lo **sovrascrive**
 - Se non esiste, lo **crea**

File testuali e binari

■ File **testuali**

- Sono "*human readable*"
- I byte in questi file sono interpretati come sequenze di codici, ogni codice rappresenta un carattere
 - ASCII o altra codifica (Es. UTF)
- La fine di una linea è indicata da caratteri speciali
- Es. un sorgente **.java** è un file testuale

■ File **binari**

- Sequenze di byte non interpretabili come codici di caratteri
- Es. il **bytecode** Java (file **.class**) generato dalla compilazione di un sorgente è un file binario

File in Java

- Quali classi Java usare per manipolare files?
- File testuali:
 - classi **Scanner** e **File** per **leggere** da file (**JDK 1.5+**)
 - classe **FileReader** per **leggere** da file (**prima di JDK 1.5**)
 - classe **FileWriter** per **scrivere** su file
- File binari:
 - classe **FileInputStream** per leggere da file
 - classe **FileOutputStream** per scrivere su file
- Queste (e altre) classi sono definite in **java.io**
 - Vanno importate per essere utilizzate

Scanner per file

- Possiamo usare **Scanner** per leggere da file
- Invece di usare **System.in** come parametro del costruttore, usiamo un oggetto di tipo **File** inizializzato col **nome** del file da leggere
 - Una volta creato lo scanner, si possono usare i suoi metodi come se si leggesse da tastiera
- Ex: leggere da un file **pipipo.txt**

```
Scanner scan =  
    new Scanner(new File("pipipo.txt"));  
int n = scan.nextInt();
```

Leggere da file

- Quando si vuole leggere da un file che **non esiste** viene sollevata una **FileNotFoundException**
 - Es. typo o si specifica un path sbagliato...
- **FileNotFoundException** è un'eccezione **checked**: non si può ignorare. Due possibilità:
 - **Gestire** l'eccezione con apposito blocco **try-catch**
 - **Rilanciare** l'eccezione semplicemente annotando con **throws** il metodo che eventualmente la lancia

```
public void mioMetodo(...parametri...)
    throws FileNotFoundException { ...
```

Esempio

```
import java.io.File;
import java.util.Scanner;
...

Scanner scf = new Scanner(new File("info.txt"));
// verifico che ci sia una linea da leggere
// nel file col metodo hasNextLine
while (scf.hasNextLine()) {
    String riga = scf.nextLine(); // leggo la riga
    System.out.println("RIGA: " + riga);
}
// alla fine chiudo lo Scanner
scf.close();
```

Esempio

- Editate un file info.txt:
Here is a small text file
that you will use to
test java.util.scanner.
- E testatelo:

```
RIGA: Here is a small text file  
RIGA: that you will use to  
RIGA: test java.util.scanner.  
> |
```

Esempio completo

```
import java.io.*;
import java.util.Scanner;

public class letturaFileThrows {

    public static void main(String[] args)
        throws FileNotFoundException {
        // Non gestisce l'eccezione ma la rilancia... A chi?
        Scanner scf = new Scanner(new File("info.txt"));
        while (scf.hasNextLine()) {
            String riga = scf.nextLine();
            System.out.println("RIGA: "+riga);
        }
        scf.close();
    }
}
```

Esempio completo (meglio)

```
import java.io.*;
import java.util.Scanner;

public class letturaFile {
    public static void main(String[] args){
        try{ // In questo caso l'eccezione viene gestita
            Scanner scf = new Scanner(new File("info.txt"));
            while (scf.hasNextLine()) {
                String riga = scf.nextLine();
                System.out.println("RIGA: "+riga);
            }
            scf.close();
        }
        catch (FileNotFoundException e) {
            System.out.println("Attenzione! Il file non esiste");
        }
    }
}
```

Altro esempio

```
// Stampa gli int scritti nel file dati1.txt
import java.io.*;
import java.util.Scanner;

...
Scanner scf = new Scanner(new File("dati1.txt"));

// Finchè ci sono numeri nel file leggo un intero
while (scf.hasNextInt())
    System.out.println(scf.nextInt() + " ");
scf.close(); // alla fine chiudo lo Scanner

// Attenzione: se il dato da leggere non è int,
// hasNextInt ritorna false e il ciclo termina.
```

Altro esempio

```
// Stampa i long scritti nel file dati1.txt
import java.io.*;
import java.util.Scanner;

...
Scanner scf = new Scanner(new File("dati1.txt"));

// Finchè ci sono numeri nel file leggo un intero
while (scf.hasNextLong())
    System.out.println(scf.nextLong() + " ");
scf.close(); // alla fine chiudo lo Scanner

// Attenzione: se il dato da leggere non è long,
// hasNextInt ritorna false e il ciclo termina.
```


Controllo file

- Se si apre in lettura un file che non esiste, viene sollevata **FileNotFoundException**
 - ▢ Magari esiste in qualche altra cartella, oppure typo nel nome del file...
- La classe **File** fornisce alcuni metodi per **controllare** se è possibile leggere da un file:
 - ▢ **exists()**
 - ▢ **canRead()**
- NOTA: la classe **File** è una “*rappresentazione astratta*” di un file
Se **x** non è un nome di file valido:
 - ▢ **f = new File(x)** non solleva alcuna eccezione
 - ▢ **new Scanner(f)** solleva **FileNotFoundException**

Controllo file

```
...  
// Non solleva eccezioni anche se myFile non è ok!  
File f = new File("myFile.dat");  
  
if (!f.exists()) {  
    System.out.println("File non esistente!");  
    return 1;  
}  
if (!f.canRead()) {  
    System.out.println("File non leggibile!");  
    return 2;  
}  
  
Scanner scan = new Scanner(f);  
...
```

Esercizio (DoppiaT.java)

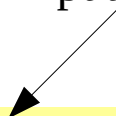
- Scrivere un programma Java che chieda all'utente il nome di un file di testo e lo controlli con **exists** e **canRead**
 - Il nome va reinserito fino a che entrambi questi controlli passano
- Quindi leggere ogni riga di tale file e stampare solo quelle che contengono una doppia "T" consecutiva
 - maiuscola o minuscola non importa
- Es. Se Il file contiene le righe:
Informatica
FATT0
Alice
MaTtEo
- Il programma dovrà stampare **FATTO** e **MaTtEo**

Metodo next()

- Il metodo **next ()** di Scanner restituisce un **token**, cioè un “*pezzo di dati*” separato da **delimitatori**
 - In questo caso un “pezzo di stringa”, ma il token può anche essere un numero (nextInt, nextFloat...), un Boolean, ...
- Di default, i delimitatori sono i **whitespaces**
 - Non solo carattere di spazio, ma anche ‘\t’, ‘\n’, ...
 - [https://docs.oracle.com/javase/7/docs/api/java/lang/Character.html#isWhitespace\(char\)](https://docs.oracle.com/javase/7/docs/api/java/lang/Character.html#isWhitespace(char))
- next () scorre lo stream di input *sequenzialmente*, **scartando** eventuali delimitatori, fino a che trova un token **valido**
 - Se lo trova, lo **ritorna** e si posiziona sul **delimitatore successivo**
- nextLine(), nextInt(), nextDouble(), ... funzionano esattamente allo stesso modo

Metodo next()

La sorgente di
input è una stringa



```
Scanner scan = new Scanner("    ciao \tmondo\n");
String str = scan.next();
// scan inizialmente è posizionato all'inizio di str
// (posizione 0). Quindi scorre str ignorando i 3 spazi
// iniziali, ritorna il token "ciao" e si posiziona sullo
// spazio dopo "ciao" (posizione 7)
str = scan.next();
// scan ignora spazio e \t, ritorna il token "mondo" e si
// posiziona subito dopo (sul carattere \n in pos. 14)
str = scan.next();
// scan ignora \n e raggiunge la fine del buffer senza
// aver trovato un nuovo token: viene lanciata l'eccezione
// NoSuchElementException, evitabile invocando il metodo
// in.hasNext() che ritorna false in questo caso
```

Metodo nextInt()

```
Scanner scan = new Scanner(System.in);  
while (true) {  
    try {  
        System.out.println("Dammi un intero:");  
        return scan.nextInt();  
    }  
    catch (InputMismatchException exc) {  
        System.out.println("Numero non corretto,  
                             riprova.");  
    }  
}
```

In teoria, se inserisco un intero non valido il programma mi chiede l'inserimento di un altro intero

Metodo nextInt()

Dammi un intero:

ciao

Numero inserito non corretto, riprova.

Dammi un intero:

Numero inserito non corretto, riprova.

Dammi un intero:

Numero inserito non corretto, riprova.

Dammi un intero:

Numero inserito non corretto, riprova.

...

*In pratica, se il primo valore inserito non è un intero otterrò un **loop infinito**... Perché?*

Metodo nextInt()

```
Scanner scan = new Scanner(System.in);
while (true) {
    try {
        System.out.println("Dammi un intero:");
        return scan.nextInt();
    }
    catch (InputMismatchException exc) {
        System.out.println("Numero non corretto,
                            riprova.");
    }
}
```

Se **nextInt** fallisce, la sua esecuzione termina ma quanto letto **rimane** nel buffer e **scan** “non avanza”. Nell’iterazione seguente **nextInt** rileggerà lo stesso buffer... e così via!

Metodo nextInt()

```
Scanner scan = new Scanner(System.in);
while (true) {
    try {
        System.out.println("Dammi un intero:");
        return scan.nextInt();
    }
    catch (InputMismatchException exc) {
        System.out.println("Numero non corretto,
                            riprova.");
        scan.nextLine();
    }
}
```

Una possibile soluzione è forzare una **nextLine()** nel blocco **catch** per “resettare” il buffer leggendo la linea corrente

- In alternativa, utilizzare **nextLine()** per leggere riga per riga e **Integer.parseInt** al posto di **nextInt()**

Scanner e stringhe

- Quando creo un oggetto Scanner posso anche usare una stringa come argomento

```
import java.io.*;
import java.util.Scanner;

...
String prova = "metto una stringa di prova";

Scanner scan = new Scanner(prova);
...
```

Scanner e delimitatori

- Si possono specificare delimitatori *ad hoc* col metodo **useDelimiter**
 - Es. Parole chiave e espressioni complicate

```
// Il delimitatore in questo caso è la stringa  
// "fish" preceduta e seguita da zero o più  
// whitespace, identificati con \s*  
scan.useDelimiter("\\s*fish\\s*");
```

Esempio

```
import java.io.*;
import java.util.Scanner;

String input = "1_fish_2_fish__red_fish_blue_fish";
// Ovviamente, funziona anche se input viene immesso
// da file o da tastiera
Scanner s = new
    Scanner(input).useDelimiter("\\s*fish\\s*");

System.out.println(s.nextInt());
System.out.println(s.nextInt());
System.out.println(s.next());
System.out.println(s.next());
s.close();
```

1
2
red
blue

Esempio

- Editate un file **Employee.data**
Joe, 38, true
Kay, 27, true
Lou, 33, false
- Leggiamo il file una riga per volta e poi facciamo il **parsing** della stringa con una funzione apposita

Esempio

```
import java.util.Scanner;
import java.io.File;

...
void parseLine(String line) { ... } // vedi prossimo lucido

... void main(){
    // ometto il try {} catch per gestire eccezioni
    // creo l'oggetto Scanner
    Scanner scan = new Scanner(new File("Employee.data"));
    // leggo tutte le righe del file
    while (scan.hasNextLine()) {
        // per ogni riga che leggo chiamo
        // la funzione parseLine()
        parseLine(scan.nextLine());
    }
    scan.close();
}
```

Esempio

```
// funzione parseLine
void parseLine(String line) {
    // creo uno Scanner per la riga inserita
    Scanner lineScanner = new Scanner(line);

    // setto che il delimitatore è una "," con spazi prima o dopo
    lineScanner.useDelimiter("\\s*,\\s*");

    // prendo il primo Token (una stringa)
    String name = lineScanner.next();

    // secondo Token (un int)
    int age = lineScanner.nextInt();

    // terzo Token (un boolean)
    boolean isCertified = lineScanner.nextBoolean();

    System.out.println("It is " + isCertified + " that " + name +
        ", age " + age + ", is certified.");
}
```

Output

```
// funzione parseLine
void parseLine(String line) {
    // creo uno Scanner per la riga inserita
    Scanner lineScanner = new Scanner(line);

    // setto che il delimitatore è una "," con spazi
    lineScanner.useDelimiter("\\s*,\\s*");

    // prendo il primo Token (una stringa)
    String name = lineScanner.next();

    // secondo Token (un int)
    int age = lineScanner.nextInt();

    // terzo Token (un boolean)
    boolean isCertified = lineScanner.nextBoolean();

    System.out.println("It is " + isCertified + " that " + name +
        ", age " + age + ", is certified.");
}
```

It is true that Joe, 38, is certified
It is true that Kay, 27, is certified
It is false that Lou, 33, is certified

Creare Oggetti

- I dati letti da un file possono essere usati nel modo più opportuno
- **Es:** ho una classe `Employee` con costruttore del tipo:
`Employee(String name, int age, boolean isCertified)`
posso creare un nuovo oggetto con i dati letti in ogni riga

- **Es:** posso anche creare **collezioni** di tali oggetti
`ArrayList<Employee>`
`employees = new ArrayList<Employee>();`
`...`
`Employee myEmpl =`
`new Employee(name, age, isCertified);`
`employees.add(myEmpl);`

StringTokenizer

- Per fare il parsing una stringa di input si può usare la classe **StringTokenizer**

```
String delims = ",";
String str = "one,two,,three,four,,five";

StringTokenizer st = new StringTokenizer(str, delims);
while (st.hasMoreElements())
    System.out.println("Element: " + st.nextElement());
```

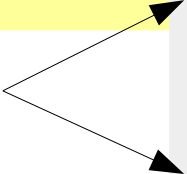
```
Element: one
Element: two
Element: three
Element: four
Element: five
```

Split

- Il metodo **split** di **String** ritorna un array di tokens
 - Anche tokens “vuoti”

```
String delims = ",";
String str = "one,two,,three,four,,five";
String[] tokens = str.split(delims);
for (int i = 0; i < tokens.length; i++)
    System.out.println("Token: "+tokens[i]);
```

Attenzione!



```
Token: one
Token: two
Token:
Token: three
Token: four
Token:
Token: five
```

Esercizio (Prodotti.java)

- Leggere un file testuale in formato **CSV** (*Comma Separated Value*) contenente una lista di prodotti venduti del tipo:

Codice, Quantità, Prezzo, Descrizione

4039, 50, 0.99, ARANCIATA

, 5, 9.50, T-SHIRT

1949, 30, 110.00, LIBRO JAVA

5199, 25, 1.50, BISCOTTI

- La prima riga (header) è sempre uguale e va ignorata
- Lo stesso prodotto non può comparire in righe diverse
- Esistono prodotti il cui campo codice non è presente
- Stampare a video:
 - Il numero totale, il ricavo totale e medio dei prodotti venduti
 - il **nome** del prodotto più venduto e di quello meno venduto
- *Hint*: usare **Integer.parseInt** e **Double.parseDouble** per convertire da string a numero

Scrivere su File

Scrivere su file

- Un file non è solamente una sorgente di dati di input
- Si può ovviamente **scrivere** su un file
 - I dati scritti su file sono **persistenti**
 - Attenzione alle **sovrascritture**!
- Vediamo come scrivere su un file di **testo**

PrintWriter

- Prima di scrivere caratteri in un file di testo occorre **aprire** il file in modalità **scrittura**
 - Si crea un oggetto di tipo **PrintWriter**:

```
PrintWriter writer = new PrintWriter("foo.txt")
```

- Se il file non esiste, viene **creato**
 - Se esiste, di norma il suo contenuto viene **sovrascritto**
 - Se vogliamo aggiungere elementi **in coda** al file senza sovrascriverlo (modalità **append**) si usa:

```
FileWriter fw = new FileWriter("foo.txt", true);
```

FileWriter vs PrintWriter

- **FileWriter** non ha **print()/println()**
- Si può però creare un oggetto di tipo **PrintWriter** che **incapsula** un oggetto di **FileWriter**

```
FileWriter writer = new FileWriter("file.txt", true);
PrintWriter pw = new PrintWriter(writer);
pw.println("Ciao");
...
```


PrintWriter

- Sull'oggetto creato usiamo i normali metodi **print()**, **println()** che in genere vengono utilizzati per stampare a video

```
pw.println("Ciao mare");
```

- Al termine della scrittura occorre chiudere il file:

```
pw.close();
```

Esempio

```
import java.io.*;
import java.util.Scanner;
...
Scanner in = new Scanner(new File("input.dat"));
PrintWriter out = new PrintWriter("output.dat");
// Leggo da un file, scrivo in un altro
while (in.hasNextLine())
    out.println(in.nextLine());
// Chiudo gli stream
in.close();
out.close();
// A questo punto, ho copiato input.dat in
// output.dat
```

Esempi dal libro

LineNumberer

- Il programma legge tutte le righe di un file in input e le invia ad un file di output, facendole precedere dal corrispondente numero di riga “commentato”. Es. se il file in input è:

Mary had a little lamb
Whose fleece was white as snow.
And everywhere that Mary went,
The lamb was sure to go!

- allora quello di output sarà:

```
/* 1 */ Mary had a little lamb  
/* 2 */ Whose fleece was white as snow.  
/* 3 */ And everywhere that Mary went,  
/* 4 */ The lamb was sure to go!
```

LineNumberer

```
01: import java.io.File;
02: import java.io.FileNotFoundException;
03: import java.io.PrintWriter;
04: import java.util.Scanner;
05:
06: public class LineNumberer
07: {
08:     public static void main(String[] args)
09:         throws FileNotFoundException
10:     {
11:         Scanner scan = new Scanner(System.in);
12:         System.out.print("Input file: ");
13:         String inputFileName = scan.next();
14:         System.out.print("Output file: ");
15:         String outputFileName = scan.next();
16:
17:         File reader = new File(inputFileName);
18:         Scanner in = new Scanner(reader);
19:         PrintWriter out = new PrintWriter(outputFileName);
20:         int lineNumber = 1;
```

LineNumberer

```
21:
22:     while (in.hasNextLine())
23:     {
24:         String line = in.nextLine();
25:         out.println("/* " + lineNumber + " */ " + line);
26:         lineNumber++;
27:     }
28:
29:     out.close();
30: }
31: }
```

- Questo programma può essere utilizzato per aggiungere i numeri di riga a un sorgente Java
 - Anche il file `LineNumberer` stesso
- Anzichè `File` si può usare **`FileReader`** per leggere da file
 - Non uso metodi come `exists` o `canRead`

LineNumberer

```
$ javac LineNumberer.java && java LineNumberer
```

```
Input file: LineNumberer.java
```

```
Output file: LineNumberer2.java
```

```
$ more LineNumberer2.java
```

```
/* 1 */ import java.io.File;
/* 2 */ import java.io.FileNotFoundException;
/* 3 */ import java.io.PrintWriter;
/* 4 */ import java.util.Scanner;
/* 5 */
/* 6 */ public class LineNumberer
/* 7 */ {
/* 8 */     public static void main(String[] args)
/* 9 */         throws FileNotFoundException
/* 10 */     {
/* 11 */         Scanner console = new Scanner(System.in);
/* 12 */         System.out.print("Input file: ");
...

```

Altro esempio

- Vediamo un esempio completo di programma che utilizza la gestione di eccezioni e file.
- Il programma chiede all'utente il nome del file, che deve contenere dati secondo queste specifiche:
 - la prima riga contiene il numero totale di valori **N**
 - le righe successive contengono **N numeri double**, uno per riga
 - Quindi il file deve avere in totale $N+1$ righe
- Ad es.:
 - 3
 - 1.45
 - 2.1
 - 0.05

Possibili eccezioni

- Cosa può andare storto?
 - Il file in input potrebbe non esistere
 - Il file potrebbe contenere dati in formato errato
- Chi può individuare tali errori?
 - Se il file non esiste, il costruttore di Scanner o FileReader lancerà un'**eccezione**
- Anche il metodo che elabora i dati in input può lanciare una eccezione se c'è un errore nel formato
 - Oppure se ci sono più o meno di **N** valori

Eccezioni

- Quali eccezioni possono essere lanciate?
- Se il file non esiste, viene lanciata un'eccezione di tipo **FileNotFoundException**
- Il metodo **close** di **FileReader** può lanciare una **IOException**
- Se il file di input non è in formato corretto definiamo un'eccezione *ad hoc*: **BadDataException**
- Queste eccezioni verranno gestite nel **main**

DataAnalyzer

```
01: import java.io.FileNotFoundException;
02: import java.io.IOException;
03: import java.util.Scanner;
04:
05: /**
06:     Questo programma legge un file contenente numeri e ne analizza il
07:     contenuto. Se il file non esiste o contiene stringhe che non siano
08:     numeri, visualizza un messaggio d'errore.
09: */
10: public class DataAnalyzer
11: {
12:     public static void main(String[] args)
13:     {
14:         Scanner in = new Scanner(System.in);
15:         DataSetReader reader = new DataSetReader(); /* definita da noi
16:             contiene metodi readFile, readData e readValue */
17:         boolean done = false;
18:         while (!done)
19:         {
20:             try
21:             {
22:                 System.out.println("Please enter the file name: ");
23:                 String filename = in.next();
```

DataAnalyzer

```
24:
25:         double[] data = reader.readFile(filename);
26:         double sum = 0;
27:         for (double d : data)
28:             sum = sum + d;
29:         System.out.println("The sum is " + sum);
30:         done = true;
31:     } // try
32:     catch (FileNotFoundException exception)
33:     {
34:         System.out.println("File not found.");
35:     }
36:     catch (BadDataException exception)
37:     {
38:         System.out.println("Bad data: " + exception.getMessage());
39:     }
40:     catch (IOException exception)
41:     {
42:         exception.printStackTrace();
43:     }
44: } // while
45: } // main
46: } // DataAnalyzer
```

Ciclo su tutti
gli elementi **d**
di tipo double
di **data**

DataSetReader.readFile

```
public double[] readFile(String filename)
    throws IOException, BadDataException
/* FileNotFoundException IS-A IOException */ {
    FileReader reader = new FileReader(filename);
    try {
        Scanner in = new Scanner(reader);
        readData(in);
    }

    // Nessuna clausola catch: eventuali eccezioni
    // sono rilanciate al chiamante
    finally {
        reader.close();
    }
    return data; // data è un campo privato della
                // classe di tipo double[]
}
```

DataSetReader.readData

readData, che legge il numero di valori, costruisce un array e invoca **readValue** per ciascun valore

```
private void readData(Scanner in)
throws BadDataException {
    if (!in.hasNextInt())
        throw new BadDataException("Length expected");
    int numberOfValues = in.nextInt();
    data = new double[numberOfValues];
    for (int i = 0; i < numberOfValues; i++)
        readValue(in, i);
    if (in.hasNext())
        throw new BadDataException("End of file expected");
}
```

DataSetReader.readValue

```
private void readValue(Scanner in, int i)
throws BadDataException {
    if (!in.hasNextDouble())
        throw new BadDataException("Data value expected");
    data[i] = in.nextDouble();
}
```

BadDataException

```
/**
    Questa classe segnala un errore nei dati in ingresso
 */
public class BadDataException extends Exception {

    public BadDataException() {}

    public BadDataException(String message) {
        super(message); // chiama il costruttore della
                        // classe "genitore"
    }
}
```


Esempio di eccezione

1. **main** chiama **DatasetReader.readFile**
2. **readFile** chiama **readData**
3. **readData** chiama **readValue**
4. **readValue** non trova il valore atteso e lancia un'eccezione (non catturata) di tipo **BadDataException**
5. **readData** non gestisce tale eccezione quindi rilancia
6. **readFile** non gestisce tale eccezione quindi rilancia
 - **dopo** aver eseguito la clausola **finally** che chiude il file
7. **DataAnalyzer.main** ha un gestore di eccezioni di tipo **BadDataException** che visualizza un messaggio di errore
 - **Nota:** l'esecuzione riprende con una nuova iterazione del ciclo while (la variabile **done** rimane **false**)

Esercizio (DistanzaMax.java)

- Implementare le classi **DataAnalyzer**, **DataSetReader** e **BadDataException**
 - ▢ Testarle con diversi input
- Modificare l'es. in modo che i valori di input siano punti del piano Cartesiano, cioè **coppie di double** separate da virgola
- DataAnalyzer deve ritornare la coppia di punti avente massima **distanza Euclidea**: $dist((x,y), (x',y')) = \sqrt{(x-x')^2 + (y-y')^2}$
- Es. di input valido (la prima riga è sempre il numero di valori)

3

2.5,7

2.6,7.5

3,11

$dist((2.5, 7), (2.6, 7.5)) = 0.509902$

$dist((2.5, 7), (3, 11)) = 4.031129$

$dist((2.6, 7.5), (3, 11)) = 3.522783$

→ Deve stampare (2.5,7) e (3,11)

Altri esercizi

Esercizio 1

- Chiedere da input due numeri interi **N** e **M**, e generare **N** stringhe **casuali** di lunghezza **M**
 - I caratteri con codice ASCII dal 32 al 126 sono **stampabili** (32 = spazio)
- Scrivere le stringhe su file, una sotto l'altra
- Es. con $N = 3$, $M = 8$

xmSsa0@#

e90d'c1"

?[;-wWqx

Esercizio 2

- Leggere da **file di input** una sequenza di **int** uno sotto l'altro e stamparli a video senza ripetizioni
 - Se si verifica un errore durante la lettura, notificare l'utente.
- Ad es. se il file contiene:
 - 3
 - 5
 - 4
 - 280
 - 4
- Il programma stamperà [-3, 5, 4, 280]

Esercizio 3

- Leggere una sequenza di **double** in un file, e scriverli in ordine **inverso** in un altro file
- Se la lettura non va a buon fine, il file di output **non** deve essere creato

Esercizio 4

- Leggere un file di testo in input, contare e stampare quante cifre da 0 a 9 sono presenti nel file
- Es. Se il file contiene:

*Scuola, da lunedì **5,7** milioni di studenti a casa: ma rischiano di rimanerci **9** su **10***

Va stampato:

Numero di 0: 1

Numero di 1: 1

...

Numero di 8: 0

Numero di 9: 1

Esercizio 5

- Creare una classe **Book** con 3 campi privati e immutabili:
 - **titolo** (String)
 - **autori** (array di String)
 - **anno** (int)
- E relativi metodi costruttori e metodi get
- Chiedere un intero **N** e successivamente inserire da tastiera **N** libri
 - Per ogni libro chiedere titolo, autori (separati da virgola) e anno di prima scrittura

Esercizio 5

- Se l'inserimento va a buon fine, scrivere in modalità **append** ogni libro nel formato:

<BOOK>

<TITLE>*titolo***<\TITLE>**

<AUTHOR>*autore1***<\<AUTHOR>**

<AUTHOR>*autore2***<\AUTHOR>**

...

<YEAR>*anno***<\YEAR>**

<\BOOK>

- Eliminare spazi prima e dopo titolo/autori
- Salvare il file con estensione **.book**

Esercizio 5

```
Dammi il numero di libri:
3
*** 1o libro ***
Dammi il titolo:
Libro
Dammi autore(i) separati da virgola:
Maccio Capatonda
Dammi l'anno di prima scrittura:
2020
*** 2o libro ***
Dammi il titolo:
Le mie prigioni
Dammi autore(i) separati da virgola:
Silvio Pellico
Dammi l'anno di prima scrittura:
1832
*** 3o libro ***
Dammi il titolo:
Intelligenza Artificiale. Un approccio moderno
Dammi autore(i) separati da virgola:
P. Norvig, S.J. Russel
Dammi l'anno di prima scrittura:
1994
```

Esercizio 5

```
<BOOK>
  <TITLE>Libro<\TITLE>
  <AUTHOR>Maccio Capatonda<\AUTHOR>
  <YEAR>2020<\YEAR>
<\BOOK>
<BOOK>
  <TITLE>Le mie prigioni<\TITLE>
  <AUTHOR>Silvio Pellico<\AUTHOR>
  <YEAR>1832<\YEAR>
<\BOOK>
<BOOK>
  <TITLE>Intelligenza Artificiale. Un approccio moderno<\TITLE>
  <AUTHOR>P. Norvig<\AUTHOR>
  <AUTHOR>S.J. Russel<\AUTHOR>
  <YEAR>1994<\YEAR>
<\BOOK>
```

Esercizio 6

- Scrivere un programma che dato un file **.book** stampa la lista dei libri in formato:

[n] Autore1, ..., AutoreN. “Titolo”. (Anno)

- **[n]** indica l'n-esimo libro stampato
- La lista degli autori deve essere in **ordine alfabetico**
 - *Hint:* per ordinare un ArrayList a, usare **Collections.sort(a)**
- *Bonus:* stampare la in **ordine cronologico inverso**, es.

[1] Maccio Capatonda, “Libro”. (2020)

[2] P. Norvig, S.J. Russel, “Intelligenza Artificiale. Un approccio moderno”. (1994)

[3] Silvio Pellico. “Le mie prigioni”. (1832)