



Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Gestione dell'I/O in Java

Parte 2: I/O di testo

Corso di Laurea in Ingegneria Informatica
Anno accademico 2021/2022

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

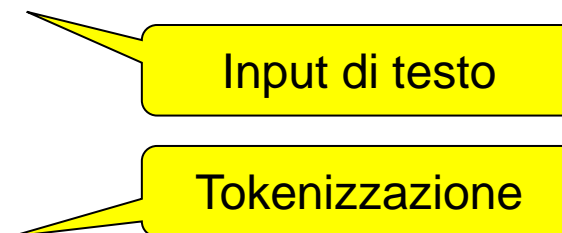


UN PO' DI STORIA

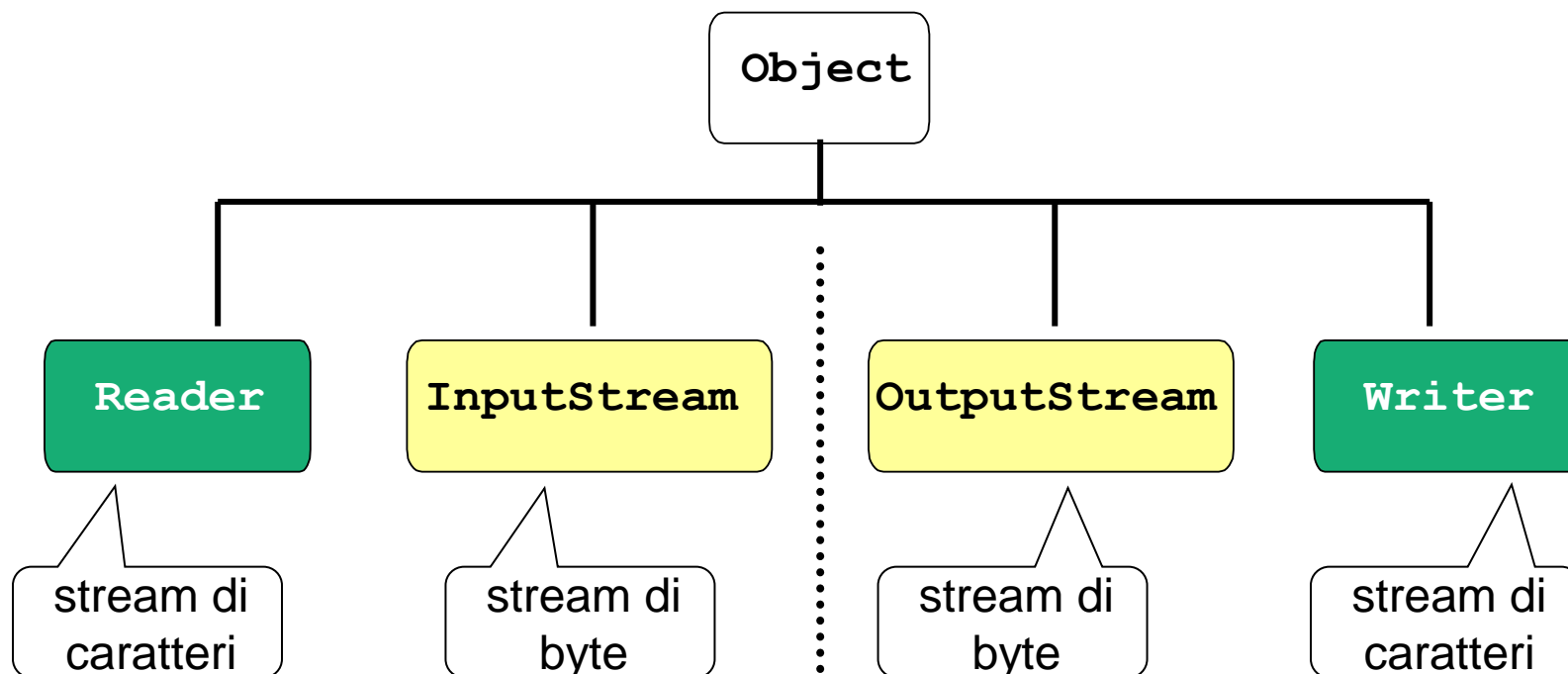
- Java 2
 - package fondamentale **java.io** e astrazioni correlate
- Java 6
 - nuova astrazione *console*
 - astrae dai molti dettagli di **System.in** e **System.out**
- Java 7
 - nuovo package **java.nio.file**
 - gestione completa dell'I/O da file (incluso *zip file system*)
 - nuova architettura
 - nuove classi factory & libreria (**Files**, **Paths**)
 - nuovo entry point **Path** (al posti di **File**)
 - nuova sintassi per le eccezioni (*try-with-resources*)

PECULIARITÀ DELL'INPUT DI TESTO

- Raramente tutto si riduce solo a leggere "del testo"
- Tipicamente, nelle applicazioni:
 - prima si **leggono** *interi righe di testo che contengono dati strutturati*
 - un treno con le sue fermate e orari
 - una valuta coi suoi tassi di cambio
 - ...
 - poi si **analizzano** le righe lette per *estrarre tali dati*
 - ad esempio, separare le singole fermate, i singoli orari, i tassi di cambi
 - infine si **usano** tali dati per *costruire oggetti (il modello dei dati)*
 - ad esempio, creare istanze di `Train` a partire dai dati letti
- Il package di I/O si occupa del *solo input*:
per la tozenizzazione si utilizzano classi accessorie



CLASSI BASE ASTRATTE



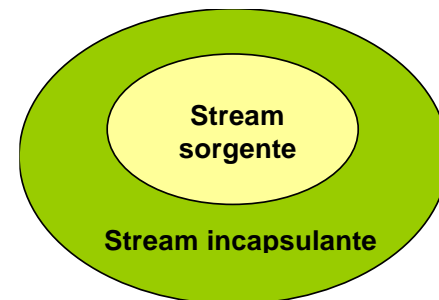
Reader e Writer

- sono *più efficienti* nella gestione di caratteri rispetto agli stream di byte
- *convertono correttamente* UNICODE nella codifica della piattaforma locale e tenendo conto della cultura locale (internazionalizzazione)

APPROCCIO "A CIPOLLA"

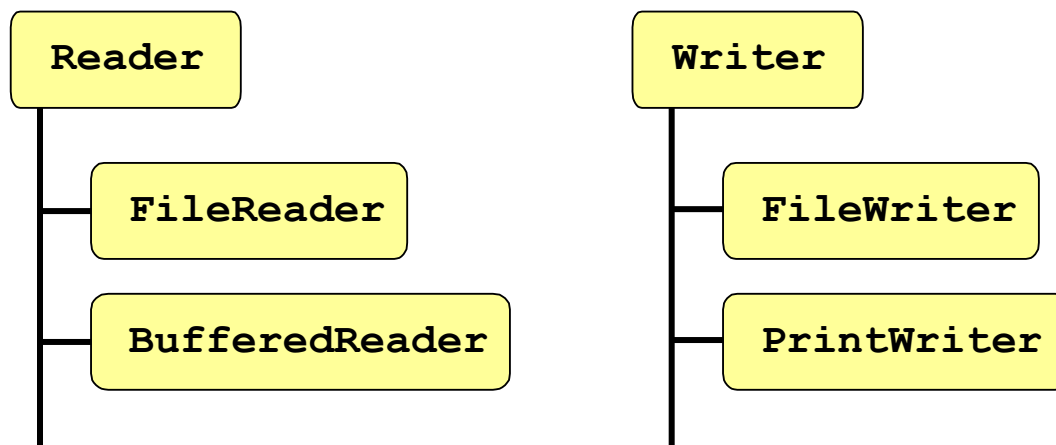
Anche gli stream di caratteri adottano l'approccio ad adapter

- al centro, stream che incapsulano *sorgenti fisiche* di dati o *dispositivi fisici di uscita*
 - i loro costruttori hanno come argomento *il dispositivo che interessa*: file, connessioni di rete, array di byte,...
- intorno, *stream di adattamento* che aggiungono funzionalità
 - i loro costruttori hanno come argomento *uno stream già esistente* (il nucleo da "avvolgere")



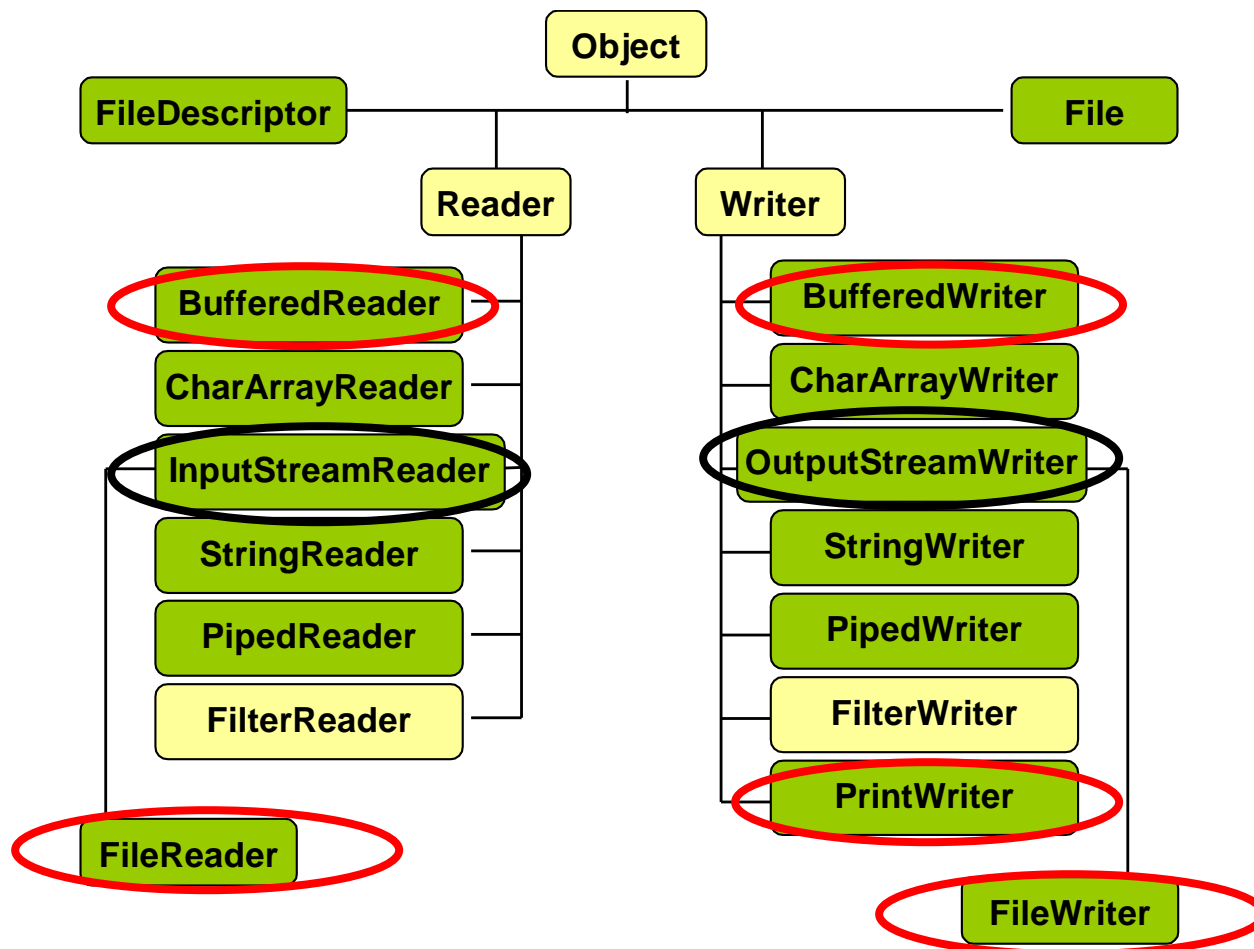
CLASSI CONCRETE

- Dalle due classi base derivano varie *classi concrete*
 - nucleo base: **FileReader** e **FileWriter**
 - adapter principali: **BufferedReader** e **PrintWriter**
 - altri adapter (verso stream binari, verso stringhe, etc.)
 - ...





TASSONOMIA COMPLETA





I/O DI TESTO

- Il file si apre istanziando un **FileReader** o **FileWriter**
- Il metodo **read** o **write** legge/scrive un **int** che contiene un carattere Unicode
 - ricorda: un carattere Unicode è lungo *due byte*
 - il metodo **read** restituisce -1 in caso di fine stream
 - occorre un *cast* per convertire il carattere Unicode letto in **int**
- Nei fatti, **FileReader** e **FileWriter** sono sempre incapsulati in un **BufferedReader** o **BufferedWriter** (o altro adapter più specifico)
- La struttura generale del codice è *del tutto analoga* a quella vista per gli stream di byte.



ESEMPIO: INPUT DA FILE (1/3)

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



ESEMPIO: INPUT DA FILE (2/3)

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

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



ESEMPIO: INPUT DA FILE (3/3)

Esempio d'uso:

```
C:\temp>java LetturaDaFileDiTesto prova.txt
```

Il risultato:

```
N e l   m e z z o   d e l   c a m m i n   d i  
n o s t r a   v i t a  
Totale caratteri: 35
```

Analogo esercizio può essere svolto per la scrittura su file di testo carattere per carattere.



INPUT BUFFERIZZATO

- **FileReader** legge solo *carattere per carattere*
- **BufferedReader** *incapsula un Reader* e lo dota di un buffer di accumulo, permettendo la *lettura per righe*
- Il metodo **readLine** restituisce la riga letta come **String**
 - se il file è finito, restituisce **null**
 - non lancia mai **EOFException**
- La riga così acquisita può poi essere tokenizzata
 - a mano (giammai!)
 - tramite classi accessorie come **StringTokenizer** o **Scanner**
 - sfruttando il metodo **split** della classe **String** (per casi semplici)



ESEMPIO: INPUT BUFFERIZZATO

```
BufferedReader f = null;
try {
    f = new BufferedReader(new FileReader("f.txt"));

    while ((String riga = f.readLine()) != null) {
        ... // elaborazione riga
    }
}
```

Test sul fine file: quando il file è finito, **readLine** restituisce **null**

A ogni iterazione:

- si legge con **readLine** una nuova *potenziale* riga
- se il risultato non è **null**, la riga esiste → la si elabora



OUTPUT BUFFERIZZATO

- **FileWriter** scrive solo *carattere per carattere*
- **BufferedWriter** *incapsula un Writer* e lo dota di un buffer di accumulo, introducendo così la *nozione di riga*
 - il metodo **newline** emette un *separatore di riga*
 - offre metodi **write** per scrivere caratteri o pezzi di stringa
 - **non offre** però metodi per una vera *stampa di righe*
- **PrintWriter** introduce la *vera stampa di righe*
 - i metodi **print/println** stampano *stringhe, oggetti, valori*
 - *gestisce le eccezioni → non richiede try/catch*

ESEMPIO: OUTPUT BUFFERIZZATO

```
PrintWriter f =  
    new PrintWriter(new FileWriter("f.txt"));  
f.print(...);  
...  
f.println(...);
```

Trattandosi di una situazione *molto frequente*, esiste un costruttore che salta un passaggio, accettando direttamente il nome del file:

```
PrintWriter f =  
    new PrintWriter("f.txt");  
f.print(...);  
...  
f.println(...);
```

Costruttore diretto dal
nome del file



LA CLASSE `Console` DI JAVA 6

- La classe **`Console`** rappresenta la *console di sistema*
 - *può non esistere* se la JVM non è stata attivata "da console" (se è stata attivata da `javaw` o Eclipse...)
 - *se esiste, ce n'è un'unica istanza (singleton)*
 - per questo, non può essere costruita direttamente
 - si ottiene solo tramite il *metodo factory* **`System.console`**
- Si usa tramite il **`Reader`** e **`Writer`** associati
 - tali stream si recuperano tramite i due metodi `reader` e `writer`
- Cosa si può fare?
 - *leggere righe intere* tramite **`readLine`**
 - *scrivere righe formattate* con **`format`** e/o **`printf`**
AARGH! (ancora lei? ebbene sì! ahi ahi ...)



Console: METODI

Method Summary

void	flush() Flushes the console and forces any buffered output to be written immediately .
Console	format(String fmt, Object... args) Writes a formatted string to this console's output stream using the specified format string and arguments.
Console	printf(String format, Object... args) A convenience method to write a formatted string to this console's output stream using the specified format string and arguments.
Reader	reader() Retrieves the unique Reader object associated with this console.
String	readLine() Reads a single line of text from the console.
String	readLine(String fmt, Object... args) Provides a formatted prompt, then reads a single line of text from the console.
char[]	readPassword() Reads a password or passphrase from the console with echoing disabled.
char[]	readPassword(String fmt, Object... args) Provides a formatted prompt, then reads a password or passphrase from the console with echoing disabled.
PrintWriter	writer() Retrieves the unique PrintWriter object associated with this console.



Console: CAUTELE D'USO

PRECISAZIONI IMPORTANTI SUI METODI

1. Invoking `close()` on the objects returned by `reader()` and `writer()` will **not** close the underlying stream of those objects.
2. The console-read methods *return null when the end of the console input stream is reached*, for example by typing control-D on Unix or control-Z on Windows. *Subsequent read operations will succeed if additional characters are later entered on the console's input device.*

RIGUARDO ALLA LETTURA DELLE PASSWORD:

1. To read a password or other secure data, an application should use `readPassword()` or `readPassword(String, Object...)`
2. After reading, it should *manually zero the returned character array* after processing, to minimize the lifetime of sensitive data in memory.



Console – ESEMPIO

```
Console console = System.console();  
String username = console.readLine("User Name? ");  
char[] password = console.readPassword("Password? ");  
...  
Arrays.fill(password, ' ');  
console.printf("Welcome, %1$s.", username);  
console.printf(System.getProperty("line.separator"));
```

Potrebbe essere *null*...

Cancellazione password

ESEMPIO DI RECUPERO PASSWORD & SUCCESSIVA CANCELLAZIONE

```
Console cons;  
char[] passwd;  
if ((cons = System.console()) != null &&  
    (passwd = cons.readPassword("[%s]", "Password:")) != null) {  
    ...  
    java.util.Arrays.fill(passwd, ' ');  
}
```

Console – ESEMPIO

```
Console console = System.console();
```

Potrebbe essere *null*...

```
ConsoleTest.java X
import java.io.Console;

public class ConsoleTest {
    public static void main(String[] args){
        Console console = System.console();
        if (console==null) {
            System.out.println("console is null");
            System.exit(1);
        }
        //else
        String username = console.readLine("User Name? ");
        char[] password = console.readPassword("Password? ");
        Arrays.fill(password, ' ');
        console.printf("Welcome, %1$s.", username);
        console.printf(System.getProperty("line.separator"));
    }
}
```

```
        System.out.println("User Name? ");
        console.readPassword("Password? ");
        Arrays.fill(password, ' ');
        console.printf("Welcome, %1$s.", username);
        console.printf(System.getProperty("line.separator"));
    }
}
```

...mentre qui
va tutto bene

Console X

```
<terminated> ConsoleTest [C:\Program Files\Java\jdk1.7.0_05\bin>
console is null
```

..E INFATTI...

C:\Windows\system32\cmd.exe

```
User Name? enrico
Password?
Welcome, enrico.
Premere un tasto per continuare . . .
```

Tokenizzazione di righe



DOPO LA LETTURA

- Come già anticipato, in molte applicazioni la lettura delle righe di testo è solo il primo passo.
- Il passo successivo è **estrarre da ogni singola riga le "parti costitutive", dette *token***
 - sottostringhe delimitate o da separatori, o da una implicita informazione sulla loro lunghezza
 - rappresentano solitamente nomi, valori numerici, etc.
- Si può fare a mano, coi metodi della classe **String...**
 - a suon di `indexOf`, `substring`, `split`, etc.
- ...ma è nettamente preferibile affidarsi a classi apposite come ***StringTokenizer*** e ***Scanner*** o al metodo ***String:split***
 - le due classi sono definite nel package `java.util`



StringTokenizer

- Il tokenizzatore **StringTokenizer** avvolge una stringa *già letta* e offre metodi per estrarne le parti costitutive

```
StringTokenizer stk = new StringTokenizer(s);  
String token = stk.nextToken();
```

- una volta creato, la stringa avvolta *non si può più cambiare*: per elaborare un'altra riga occorre creare un nuovo tokenizer
- il costruttore di default usa come **separatori fra i token** i caratteri *spazio, tabulazione, e a capo*, ma è possibile **personalizzarlo**:

```
StringTokenizer stk =  
    new StringTokenizer(s, ":-");
```

- oppure, si possono cambiare i separatori all'atto dell'uso:

```
String token = stk.nextToken(" ;");
```



ESEMPIO

```
// RIGA: parola    parola342423423
try {
    String ln;
    FileReader f = null;
    // ... segue normale apertura del file
    BufferedReader rdr = new BufferedReader(f);
    while ((line=rdr.readLine()) != null) {
        StringTokenizer stk = new StringTokenizer(line);
        String first  = stk.nextToken();
        String second = stk.nextToken("0123456789");
        String third  = stk.nextToken("\n\r");
        // fai quel che devi con first, second, third
    }
    ...
}
```




Scanner

- La classe **Scanner** (JDK 1.5) ha l'obiettivo di:
 - superare la limitazione di **StringTokenizer** di lavorare su una stringa fissata
 - essere costruibile non solo da stringa, ma anche da **Reader**, **File** o **InputStream**, così da *gestire direttamente anche la tastiera*
 - gestire le problematiche dei diversi *character encoding*
 - tenere conto delle *culture locali* nella scansione di numeri ("." vs ",", " ")
- È più potente di **StringTokenizer**, ma anche *più complessa da usare* al di fuori dei casi "banali"

COSTRUZIONE

- Si può costruire uno *Scanner*:
 - da una **stringa** già nota e letta:
`Scanner s1 = new Scanner(riga);`
 - da un **Reader** (più esattamente, da un **Readable**):
`Scanner s2 = new Scanner(new BufferedReader(...));`
 - da un **File**:
`Scanner s3 = new Scanner(new File("mynum.txt"));`
 - da **tastiera**:
`Scanner kbd = new Scanner(System.in);`

Può lanciare eccezione

Costruttore da `InputStream`

USO

- Una volta costruito, allo scanner si può chiedere:
 - **se il prossimo token è ciò che ci si aspetta**
 - `hasNextInt`, `hasNextBoolean`, `hasNextFloat`,
`hasNext`, `hasNext` (*pattern*) ,...
 - (se sì), di **estrarre un token di un ben preciso tipo:**
 - `nextInt`, `nextBoolean`, `nextFloat`,
`next`, `next` (*pattern*) ,...
 - **verificare o cambiare i delimitatori**
 - `delimiter`, `useDelimiter`, ...
- Il concetto di *pattern* si basa sulla nozione di *espressione regolare* – potente, ma non sempre immediata da usare

`java.util.regex.Pattern`



ESEMPI

- Elaborazioni varie

- lettura di un intero e di una parola da tastiera:

```
Scanner sc = new Scanner(System.in);  
int i = sc.nextInt(); String word = sc.next();
```

- lettura di una serie di interi long da file di testo:

```
Scanner sc = new Scanner(new File("numeri.txt"));  
while (sc.hasNextLong()) {  
    long x = sc.nextLong(); myList.add(x); }  
}
```

- cambio di delimitatori:

```
sc.useDelimiter("\\s*beep\\s*"); // spazi beep spazi
```

- ritorno ai delimitatori predefiniti:

```
sc.reset();
```

String: split

- **split** è un metodo aggiunto alla classe **String** (JDK 1.4) per **incapsulare un tokenizzatore**
 - funziona in modo molto simile a **Scanner**
 - facile da usare nei casi semplici (delimitatore singolo carattere sempre uguale), nettamente più complicato nel caso generale

split

```
public String[] split(String regex)
```

Dietro c'è sempre **Pattern** !
Occhio ai caratteri speciali!

Splits this string around matches of the given regular expression.

This method works as if by invoking the two-argument `split` method with the given expression and a limit argument of zero. Trailing empty strings are therefore not included in the resulting array.

The string "boo:and:foo", for example, yields the following results with these expressions:

Regex	Result
:	{ "boo", "and", "foo" }
o	{ "b", "", ":and:f" }

ESEMPI

- Situazione tipica: stringa costituita da token separati da un singolo pattern costante, ad esempio la virgola:

```
String s1 =  
"Giulio Verdi, via Indipendenza 38 , Bologna";
```

- Per tokenizzarla, basta scrivere:

```
String[] parti = riga. split(",");
```

- ottenendo (**occhio agli spazi spuri!!**):

```
parti[0] = "Giulio Verdi"  
parti[1] = " via Indipendenza 38 "  
parti[2] = " Bologna"
```

Per liberarsi degli
spazi spuri, `trim` !

Approcci tipici per casi concreti

FORMATI DI DATI TESTUALI (1/2)

I dati testuali sono tipicamente organizzati:

- o in **campi di larghezza fissa**
- o in **campi di larghezza variabile, con separatori**

Ovviamente, il modo di leggerli è diverso nei due casi.

- Ogni campo dati ha una larghezza K fissa e nota a priori
- Non servono separatori espliciti, poiché si sa già dove termina un campo e inizia il successivo
- **Strategia di lettura:**
 K caratteri per volta

- Ogni campo dati ha una larghezza NON nota a priori
- Occorrono perciò separatori espliciti (o deducibili) per capire dove termina un campo e inizia il successivo
- **Strategia di lettura:**
fino al prossimo separatore



FORMATI DI DATI TESTUALI (2/2)

I dati testuali sono tipicamente organizzati:

- o in **campi di larghezza fissa**
- o in **campi di larghezza variabile, con separatori**

Esempio con campi a larghezza fissa:

Giovanni Rossi	via Indipendenza 38	Bologna
Gian Paolo Prinzi	via Altabella 46	Bologna
Anna Maria Senzi	via dell'Arco 18/2	S.Lazzaro

Esempio con campi a larghezza variabile e separatori:

Giovanni Rossi	\$	via Indipendenza 38	\$	Bologna
Gian Paolo Prinzi	\$	via Altabella 46	\$	Bologna
Anna Maria Senzi	\$	via dell'Arco 18/2	\$	S.Lazzaro

CAMPI A LARGHEZZA FISSA

Per leggere dati in campi a **larghezza fissa**:

- si leggono K_i caratteri (K_i = lunghezza campo i-esimo)
OPPURE si estraggono dalla riga (già letta) K_i caratteri
- si tolgono gli eventuali "spazi extra" in testa e in coda.

----- 20 caratteri -----	+++++++ 23 caratteri +++++++	--11 caratteri --
Giovanni Rossi	via Indipendenza 38	Bologna
Gian Paolo Prinzi	via Altabella 46	Bologna
Anna Maria Senzi	via dell'Arco 18/2	S.Lazzaro

Strategie operative

- 1° caso:

```
char buf[] = new char[30]; fileIn.read(buf,0,20);  
String s = new String(buffer).trim();
```
- 2° caso:

```
String s = fileIn.readLine();  
String c1 = s.substring(20);
```



CAMPI A LARGHEZZA VARIABILE (1/3)

Per leggere campi a **larghezza variabile** bisogna *sapere* (o poter dedurre) *dove finisce ogni campo*

- ogni campo dev'essere **delimitato da un separatore**
OPPURE **si deve poter dedurre per altra via dove termina**
- anche qui si devono poi togliere gli eventuali "spazi extra" in testa e in coda, con **trim**

ESEMPIO

- campi separati da spazi o tabulazioni o altro
- primo campo separato dal secondo tramite virgola, etc.

ESEMPIO

- non c'è separatore esplicito,
- MA il primo campo è alfabetico mentre il secondo è numerico
→ confine implicito



CAMPI A LARGHEZZA VARIABILE (2/3)

Esempio: campi a larghezza variabile *e separatore fisso (\$)*

```
Giovanni Rossi $   via Indipendenza 38$Bologna
Gian Paolo Prinzi $ via Altabella 46$Bologna
Anna Maria Senzi$ via dell'Arco 18/2 $S.Lazzaro
```

Esempio: campi a larghezza variabile *e separatori diversi*

```
Giovanni Rossi, via Indipendenza, 38 - Bologna
Gian Paolo Prinzi, viale dei Mille, 42 - Reggio Emilia
Anna Maria Senzi, via dell'Arco, 18/2- S.Lazzaro
```

Esempio: campi a larghezza variabile *e separatori impliciti*

```
      alfabetico      numerico      alfabetico
Enrico Denti 0512093015 Bologna
Ambra Molesini 0512093274 Bologna
```



CAMPI A LARGHEZZA VARIABILE (3/3)

- Strategia operativa
 - prima, leggere l'intera riga → `readLine`
 - poi, tokenizzarla → `StringTokenizer`, `Scanner`, `split`

1° caso: campi a larghezza variabile e separatore **fisso**
→ `StringTokenizer` configurato in modo fisso
o `split` configurato in modo fisso e semplice

2° caso: campi a larghezza variabile e separatori **diversi**
→ `StringTokenizer` con `nextToken` diversa da campo a campo, in base al prossimo separatore
(`split` difficile da configurare in casi siffatti)

3° caso: campi a larghezza variabile e separatori **impliciti**
→ `StringTokenizer` con `nextToken` diversa da campo a campo, in base al prossimo insieme di separatori



1° CASO con StringTokenizer

```
...  
// rdr è un BufferedReader  
while ((riga = rdr.readLine()) != null) {  
    StringTokenizer t = new StringTokenizer(riga, "$\n\r");  
    String nomeCognome = t.nextToken().trim();  
    String indirizzo = t.nextToken().trim();  
    String città = t.nextToken().trim();  
    // costruzione modello a partire dai dati  
    list.add(new Persona(nomeCognome, indirizzo, città));  
}  
...
```

separatori dei campi
(fissi)



1° CASO con Scanner

```
...
// rdr è un BufferedReader
while ((riga = rdr.readLine()) != null) {
    Scanner sc = new Scanner(riga);
    sc.useDelimiter("\\$+(\\n\\r)*");
    String nomeCognome = sc.next().trim();
    String indirizzo = sc.next().trim();
    String città = sc.next().trim();
    list.add(new Persona(nomeCognome, indirizzo, città));
}
...
```

separatori dei campi
(fissi)

- La versione di `next` senza argomenti procede semplicemente fino al primo delimitatore
- La versione con argomento cerca una sequenza di caratteri che faccia match col pattern *ma usando i delimitatori precedentemente impostati* → occhio, non dimenticare di impostarli!



2° CASO con StringTokenizer

...

```
// rdr è un BufferedReader
```

```
while ((riga = rdr.readLine()) != null) {
```

```
    StringTokenizer t =
```

separatore fra 1° e 2° campo: la virgola

```
        new StringTokenizer(riga, ",");
```

la virgola resta lì

```
    String nomeCognome = t.nextToken().trim();
```

C'è una virgola davanti!

separatore fra 2° e 3° campo: lineetta

```
    String indirizzo = t.nextToken("-").trim();
```

il trattino resta lì

C'è un trattino davanti!

separatori fra 3° campo e fine riga

```
    String città = t.nextToken("\n\r").trim();
```

...

```
}
```




2° CASO con StringTokenizer

```
...  
// rdr è un BufferedReader  
while ((riga = rdr.readLine()) != null) {  
  
    StringTokenizer t =  
        new StringTokenizer(riga, ",");  
    String nomeCognome = t.nextToken().trim();  
    t.nextToken(" ").trim(); // butta via la virgola  
    String indirizzo = t.nextToken("-").trim();  
    t.nextToken(" ").trim(); // butta via il trattino  
    String città = t.nextToken("\n\r").trim();  
  
    ...  
}
```

la virgola resta lì

il trattino resta lì



2° CASO con Scanner

...

```
// rdr è un BufferedReader
```

```
while ((riga = rdr.readLine()) != null) {
```

```
    Scanner sc = new Scanner(riga);
```

```
    sc.useDelimiter(",");
```

primo separatore: la virgola

```
    String nomeCognome = sc.next();
```

```
    sc.skip("\\s*,\\s*");
```

```
    sc.useDelimiter("-");
```

Salta "spazi trattino spazi",
poi cambia separatore.
Nuovo separatore il trattino

```
    String indirizzo = sc.next();
```

```
    sc.skip("\\s*-\\s*");
```

```
    sc.useDelimiter("\\n\\r");
```

```
    String città = sc.next();
```

Salta "spazi trattino spazi",
poi cambia separatore.
Nuovo separatore: newline

...

```
}
```



2° CASO con Scanner

...

```
// rdr è un BufferedReader
```

```
while ((riga = rdr.readLine()) != null) {
```

```
    Scanner sc = new Scanner(riga);
```

```
    sc.useDelimiter(",");
```

fa match con qualunque
sequenza di caratteri ≠ virgola

```
    String nomeCognome = sc.next("[^,]*");
```

```
    sc.skip("\\s*,\\s*");
```

```
    sc.useDelimiter("-");
```

```
    String indirizzo = sc.next("[^-]*");
```

```
    sc.skip("\\s*-\\s*");
```

```
    sc.useDelimiter("\\n\\r");
```

```
    String città = sc.next();
```

...

```
}
```

fa match con qualunque
sequenza di caratteri ≠ trattino



3° CASO con StringTokenizer

...

```
// rdr è un BufferedReader
```

```
while ((riga = rdr.readLine()) != null) {
```

```
StringTokenizer t =
```

Il 2° campo inizia con una cifra numerica

```
    new StringTokenizer(riga, "0123456789");
```

```
String nomeCognome = t.nextToken().trim();
```

separatori fra 2° e 3° campo (spazi)

```
String telefono = t.nextToken(" ").trim();
```

separatori fra 3° campo e fine riga

```
String città = t.nextToken("\n\r").trim();
```

...

```
}
```



3° CASO con Scanner

...

```
// rdr è un BufferedReader
```

```
while ((riga = rdr.readLine()) != null) {
```

```
    Scanner sc = new Scanner(riga);
```

```
    sc.useDelimiter("\\d");
```

separatori = digits

```
    String nomeCognome = sc.next();
```

```
    sc.reset();
```

separatori successivi: spazi
e newline

```
    String indirizzo = sc.next();
```

```
    String città = sc.next();
```

...

```
}
```



GLI STESSI ESEMPI con `split`

Le due stringhe seguenti sono multi-linea con, entro ogni stringa, più token separati da un singolo pattern costante:

```
String fintoFile1 =  
"Giulio Verdi, via Indipendenza 38 , Bologna\n" +  
"Pier Nicola Prinzi, viale dei Mille 42, Reggio Emilia\n" +  
"Vanna Maria Senzi, via dell'Arco 18/2, S.Lazzaro\n" ;
```

```
String fintoFile2 =  
"Giovanni Rossi $ via Indipendenza 38$Bologna\n" +  
"Gian Paolo Prinzi $ via Altabella 46$Bologna\n" +  
"Anna Maria Senzi$ via dell'Arco 18/2 $S.Lazzaro\n" ;
```

Occhio: la prima usa un separatore standard (la virgola),
la seconda invece un *carattere speciale* (il dollaro)



GLI STESSI ESEMPI con `split`

Costruiamoci una funzioncina di test che *simuli la presenza di un file di testo* alimentando un reader con una stringa:

```
static void readFromString(String fakeFile, String sep)
                                throws IOException {
    BufferedReader rdr =
        new BufferedReader(new StringReader(fakeFile));
    String riga;
    while((riga = rdr.readLine()) != null) {
        String[] parti = riga.split(sep);
        System.out.println(Arrays.asList(parti));
    }
    System.out.println("-----");
}
```

Stampiamo i vari token come lista solo per nostra comodità



GLI STESSI ESEMPI con `split`

Ora proviamo a leggere e tokenizzare la prima stringa:

- il separatore è la virgola, un carattere standard: quindi, di base basta indicare tale carattere come pattern:

```
readFromString(fintoFile1, ",");
```

- però, così facendo **restano spazi puri intorno ai token:**

```
[Giulio Verdi, □via Indipendenza 38 □, □Bologna]  
[Pier Nicola Prinzi, □viale dei Mille 42, □Reggio Emilia]  
[Vanna Maria Senzi, □via dell'Arco 18/2, □S.Lazzaro]
```

Token stampati come lista solo
per nostra comodità



GLI STESSI ESEMPI con `split`

Possiamo allora **specificare un pattern più sofisticato**, una vera *regular expression* usando la "giusta" combinazione

- il pattern "`\\s`" indica lo spazio
- il pattern "`*`" la ripetizione (per 0 o più volte) di ciò che precede

```
readFromString(fintoFile1, "\\s*,\\s*");
```

- così facendo **gli spazi spuri scompaiono** e i singoli token sono perfettamente *puliti*:

```
[Giulio Verdi, via Indipendenza 38, Bologna]
```

```
[Pier Nicola Prinzi, viale dei Mille 42, Reggio Emilia]
```

```
[Vanna Maria Senzi, via dell'Arco 18/2, S.Lazzaro]
```

Token stampati come lista solo
per nostra comodità



GLI STESSI ESEMPI con `split`

La seconda stringa è più complicata:

- il separatore è il dollaro, un carattere speciale nel senso che all'interno dei pattern *avrebbe un suo significato particolare*
- perciò se scrivessimo `readFromString(fintoFile1, "$");` essa *NON funzionerebbe come previsto* (\$ significa "a capo")

Boundary matchers

^

The beginning of a line

\$

The end of a line



GLI STESSI ESEMPI con `split`

La seconda stringa è più complicata:

- il separatore è il dollaro, un carattere speciale nel senso che all'interno dei pattern *avrebbe un suo significato particolare*
- perciò se scrivessimo `readFromString(fintoFile1, "$");` essa *NON funzionerebbe come previsto* (\$ significa "a capo")
- ergo, *è necessario neutralizzare il significato speciale del \$* antepoendo una *barra inversa* (escaping)

```
readFromString(fintoFile1, "\\$");
```

- versione con *spazi spuri intorno ai token*:

```
[Giovanni Rossi$   $via Indipendenza 38$ Bologna]  
[Gian Paolo Prinzi$ $via Altabella 46$ Bologna]  
[Anna Maria Senzi$ $via dell'Arco 18/2$ S.Lazzaro]
```



GLI STESSI ESEMPI con `split`

Anche qui possiamo far eliminare gli spazi spuri:

```
readFromString(fintoFile1, "\\s*\\$\\s*");
```

- versione con token ripuliti e senza spazi spuri:

```
[Giovanni Rossi$ via Indipendenza 38$ Bologna]  
[Gian Paolo Prinzi$ via Altabella 46$ Bologna]  
[Anna Maria Senzi$ via dell'Arco 18/2$ S.Lazzaro]
```

In definitiva:

il metodo `split` è molto comodo, ma bisogna conoscerlo!

Altrimenti, si rischia di perdere (molto) tempo a fare debug cercando la ragione di comportamenti "inspiegabili"...

@ZioEnrico: usare solo con caratteri standard, non speciali



BILANCIO

- **StringTokenizer** è più semplice per estrarre frammenti di stringhe con delimitatori variegati
 - in tali situazioni, `Scanner` richiede di specificare delimitatori e pattern con *espressioni regolari*, potenti ma complesse da usare
- **Scanner** è vincente per estrarre valori primitivi con delimitatori semplici e per operare da tastiera
 - i metodi `nextInt...` effettuano già tutte le conversioni
 - molto comodo il costruttore che incapsula direttamente la tastierama la sua potenza va a discapito della semplicità d'uso
- **split** è molto comodo nei casi di delimitatore fisso
 - MA facendo attenzione ai caratteri speciali..

Stream di byte vs testo: adapter fra i due mondi



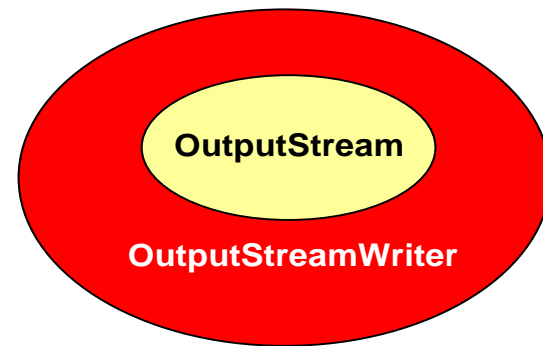
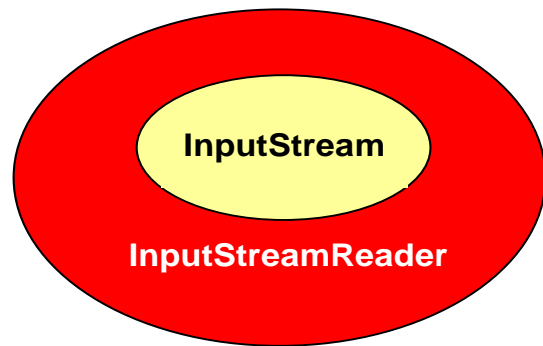
STREAM BINARI O DI TESTO ?

- Finora abbiamo diviso rigidamente la lettura / scrittura *binaria* dalla lettura / scrittura di *testo*
- Tuttavia, la suddivisione non è sempre così netta
 - la sorgente può emettere *tipi di dati diversi in momenti diversi* (es: prima testo, poi immagini, poi altro testo)
 - stream di testo «non Java» possono apparire binari
- Inoltre, se i byte in transito in quel momento sono testo, ha senso poter fare letture/scritture *per righe*

Può essere necessario
*trattare uno stream binario
come stream di testo*

UN ADAPTER FRA I DUE MONDI

A tal fine, due *stream di adattamento* incapsulano
uno stream di byte in uno stream di testo

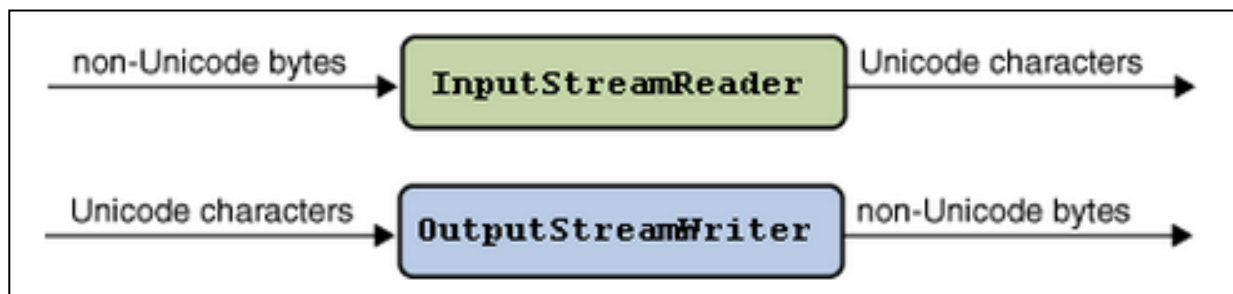


- ***InputStreamReader*** ingloba un *InputStream* e lo fa apparire all'esterno come un *Reader*
- ***OutputStreamWriter*** ingloba un *OutputStream* e lo fa apparire fuori come un *Writer*

UN ADAPTER FRA I DUE MONDI

Più in generale, i due adapter convertono

- IN LETTURA, sequenze di *byte non-Unicode* in sequenze corrette di **caratteri Unicode**
- IN SCRITTURA, sequenze corrette di **caratteri Unicode** in sequenze di *byte non-Unicode*



Quel bizzarro caso di **System.in**



Il caso tipico: TASTIERA & VIDEO

- Video e tastiera sono legati ai due stream standard (statici) **System.in** e **System.out**
- Essi sono formalmente degli stream di byte, perché risalgono a Java 1.0: però, *di solito si usano a caratteri*
 - CONSEGUAENZA: non è detto che i caratteri Unicode vengano interpretati correttamente
 - MOTIVO: negli stream di byte non esiste il concetto fondamentale di *character encoding* (poi *Charset*)
 - RISULTATO: rischio di *comportamenti "bizzarri"* in presenza di lettere accentate e altri caratteri "strani"
- Per questi motivi, il loro uso da console (prompt dei comandi) dovrebbe *sempre* essere filtrato dall'opportuno *adapter*.



UN ESEMPIO DI "BIZZARRIA"

- Si supponga di voler leggere da tastiera una stringa *contenente lettere accentate*
 - ad esempio, il giorno della settimana "**venerdì**"
- e di volerlo poi *ristampare a video*.

Come leggere da tastiera?

- **System.in** è un **InputStream**, non ha metodi adatti
- Il vecchio adapter **DataInputStream** offre un metodo **readLine** che però è *giustamente deprecato* perché *"does not properly convert bytes to characters"*
- Il modo corretto è interporre un **InputStreamReader** correttamente configurato (o usare la classe **Console**)



ESEMPIO con System.in

Con il vecchio adapter deprecato:

```
System.out.print("Inserire giorno: ");  
DataStream in = new DataInputStream(System.in);  
String giorno = in.readLine();  
System.out.println(giorno);
```

Output:

```
C:>java TestAccenti  
Inserire giorno: venerdì  
venerd?
```

BELLO, VERO?

Il metodo `readLine` di
`DataInputStream`
**sbaglia la conversione in
Unicode
→ DEPRECATO**

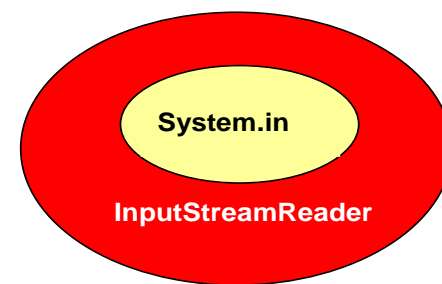
INCAPSULARE L'INPUT

La cura è adattare `System.in` a *stream di testo*:

```
InputStreamReader myIn =  
    new InputStreamReader(System.in, ...);
```

A sua volta, costui sarà incapsulato in un `BufferedReader` per avere `readLine`:

```
InputStreamReader tastiera =  
    new BufferedReader(  
        new InputStreamReader(System.in, ...) );
```





L'ESEMPIO RIVISTO

Esempio ristrutturato:

```
BufferedReader in = new BufferedReader(  
    new InputStreamReader(System.in, "CP850"));
```

Il giusto encoding

```
String giorno = in.readLine();
```

```
System.out.println(giorno);
```

Il metodo `readLine` di `BufferedReader` opera correttamente in Unicode

Output:

```
C:>java TestAccenti  
Inserire giorno: venerdì  
venerdì
```

Funziona!

CP850 è la codifica standard delle finestre «prompt dei comandi» in Windows versione italiana.

```
>chcp  
Tabella codici attiva: 850
```



L'ESEMPIO RIVISTO

Attenzione a quali finestre di output usate

- **CP 850** è l'encoding dei terminali «prompt comandi»,
non di tutte le finestre Windows!
- Come è facile scoprire, l'encoding standard delle
finestre Windows è **CP1252**:

CP1252

```
System.out.println(  
    new InputStreamReader(System.in).getEncoding());
```

Perciò, nel caso Windows:

- Se l'applicazione deve funzionare da terminale,
occorre impostare **CP 850**
- Se invece deve funzionare da finestra grafica,
va bene l'encoding di default **CP 1252**



L'ESEMPIO con Console

Esempio ristrutturato:

```
Console console = System.console();  
String giorno = console.readLine();  
System.out.println(giorno);
```

Il giusto encoding

Output:

Il metodo `readLine` di
`Console` opera
correttamente in Unicode

```
C:>java TestAccenti  
Inserire giorno: venerdì  
venerdì
```

Funziona!

Ricorda:

- `Console` è *null* quando non c'è un terminale «adatto»!



INCAPSULARE L'OUTPUT

Analogamente, si può adattare `System.out`:

```
OutputStreamWriter video =  
    new OutputStreamWriter(System.out, ...);
```

da incapsulare poi, magari, in un `PrintWriter`.

La cosa è però meno grave del caso input, perché in Java 1.0 `System.out` era un `PrintStream`, meno critico.

```
PrintWriter video =  
    new PrintWriter(  
        new OutputStreamWriter(System.out, "CP850"), true);
```

Di nuovo, `Console` (se disponibile) risolve molti problemi.



CHARSET DISPONIBILI

Un **Charset** definisce un *mapping byte/caratteri*

Per sapere quali sono quelli disponibili:

```
System.out.print(Charset.availableCharsets())
```

```
Big5, Big5-HKSCS, CESU-8, EUC-JP, EUC-KR, GB18030, GB2312, GBK, IBM-Thai, IBM00858, IBM01140,
IBM01141, IBM01142, IBM01143, IBM01144, IBM01145, IBM01146, IBM01147, IBM01148, IBM01149, IBM037,
IBM1026, IBM1047, IBM273, IBM277, IBM278, IBM280, IBM284, IBM285, IBM290, IBM297, IBM420, IBM424,
IBM437, IBM500, IBM775, IBM850, IBM852, IBM855, IBM857, IBM860, IBM861, IBM862, IBM863, IBM864,
IBM865, IBM866, IBM868, IBM869, IBM870, IBM871, IBM918, ISO-2022-CN, ISO-2022-JP, ISO-2022-JP-2,
ISO-2022-KR, ISO-8859-1, ISO-8859-13, ISO-8859-15, ISO-8859-2, ISO-8859-3, ISO-8859-4, ISO-8859-
5, ISO-8859-6, ISO-8859-7, ISO-8859-8, ISO-8859-9, JIS_X0201, JIS_X0212-1990, KOI8-R, KOI8-U,
Shift_JIS, TIS-620, US-ASCII, UTF-16, UTF-16BE, UTF-16LE, UTF-32, UTF-32BE, UTF-32LE, UTF-8,
windows-1250, windows-1251, windows-1252, windows-1253, windows-1254, windows-1255, windows-1256,
windows-1257, windows-1258, windows-31j, x-Big5-HKSCS-2001, x-Big5-Solaris, x-euc-jp-linux, x-
EUC-TW, x-eucJP-Open, x-IBM1006, x-IBM1025, x-IBM1046, x-IBM1097, x-IBM1098, x-IBM1112, x-
IBM1122, x-IBM1123, x-IBM1124, x-IBM1364, x-IBM1381, x-IBM1383, x-IBM300, x-IBM33722, x-IBM737,
x-IBM833, x-IBM834, x-IBM856, x-IBM874, x-IBM875, x-IBM921, x-IBM922, x-IBM930, x-IBM933, x-
IBM935, x-IBM937, x-IBM939, x-IBM942, x-IBM942C, x-IBM943, x-IBM943C, x-IBM948, x-IBM949, x-
IBM949C, x-IBM950, x-IBM964, x-IBM970, x-ISCII91, x-ISO-2022-CN-CNS, x-ISO-2022-CN-GB, x-iso-
8859-11, x-JIS0208, x-JISAutoDetect, x-Johab, x-MacArabic, x-MacCentralEurope, x-MacCroatian, x-
MacCyrillic, x-MacDingbat, x-MacGreek, x-MacHebrew, x-MacIceland, x-MacRoman, x-MacRomania, x-
MacSymbol, x-MacThai, x-MacTurkish, x-MacUkraine, x-MS932_0213, x-MS950-HKSCS, x-MS950-HKSCS-XP,
x-mswin-936, x-PCK, x-SJIS_0213, x-UTF-16LE-BOM, x-UTF-32BE-BOM, x-UTF-32LE-BOM, x-windows-50220,
x-windows-50221, x-windows-874, x-windows-949, x-windows-950, x-windows-iso2022jp
```

La libreria & factory **Files**



LA LIBRERIA & FACTORY Files

È una libreria + factory: contiene solo metodi statici

1. metodi per operare su file e directory

- copiare o spostare file o directory
- creare / eliminare file, directory, link simbolici
- creare file o directory temporanee
- verificare l'esistenza di file o directory
- recuperare attributi di file o directory

2. metodi per *leggere e scrivere piccoli file in blocco*

- leggere / scrivere tutti i byte di un piccolo file binario
- leggere / scrivere tutti i caratteri di un piccolo file di testo

3. metodi factory per *creare stream di I/O*

- per non dover creare a mano gli stream (come si è sempre fatto)
- metodi non essenziali, di comodità



LAVORARE SU PICCOLI FILE (1)

Lettura di un *piccolo* file di testo

```
public static void main(String args[]){  
    try {  
        List<String> rows =  
            Files.readAllLines(Paths.get("dante.txt"));  
        System.out.println("Lette " + rows.size() + " righe");  
    }  
    catch(IOException e){  
        System.out.println("Problema di lettura");  
        System.exit(1);  
    }  
}
```

Lette 3 righe

se il file esiste ed è stato
aperto regolarmente

Problema di lettura

se il file non esiste o non
può essere aperto

MAI usare questo approccio per file "non piccoli" !



LAVORARE SU PICCOLI FILE (2)

Scrittura di un *piccolo* file di testo

```
public static void main(String args[]){  
    try {  
        String[] righe = {"Nel mezzo", "del cammin di nostra vita",  
                           "mi ritrovai", "per una selva oscura",  
                           "che la diritta via", "era smarrita." };  
        Files.write(Paths.get("dante2.txt"), Arrays.asList(righe));  
        System.out.println("Scritte " + righe.length + " righe");  
    }  
    catch(IOException e){  
        System.out.println("Problema di scrittura");  
        System.exit(1);  
    }  
}
```

Scritte 6 righe

se il file esiste ed è stato
aperto regolarmente

Problema di scrittura

se il file non esiste o non
può essere aperto

MAI usare questo approccio per file "non piccoli" !



LA FABBRICA DEGLI STREAM

- Come con gli stream binari, anche qui è possibile creare gli stream direttamente (con **new**), strato per strato
- Tuttavia, *la cosa diventa particolarmente noiosa*
 - la lettura a righe richiede un **BufferedReader**
 - la scrittura a righe richiede un **BufferedWriter** (o **PrintWriter**)
- Per questo i **factory methods** della classe **Files** producono direttamente un **BufferedReader/Writer**

```
newBufferedReader(Path p)
```

```
newBufferedReader(Path p, Charset cs)
```

```
newBufferedWriter(Path p, Charset cs, OpenOption... opts)
```

```
newBufferedWriter(Path p, OpenOption... opts)
```

dove **Charset** rappresenta lo specifico *set di caratteri* da usare

Text-based persistence

PERSISTENZA TEXT-BASED

Spesso, anziché in formato binario, si preferisce **salvare i dati in formato testo**

- Vantaggi (per gli utilizzatori)
 - Leggibilità, Fruibilità, Indipendenza dalla piattaforma
- Svantaggi (per i programmatori)
 - Qualche difficoltà in più, soprattutto nella lettura

Si parla perciò di "***serializzazione in formato testo***"

From Wikipedia: *serialization is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer, or transmitted across a network connection link) and reconstructed later in the same or another computer environment*

PERSISTENZA TEXT-BASED

Necessità di stabilire un protocollo preciso

- tutti i dati vanno opportunamente scritti e letti
- tutto è stringa, progettato per essere leggibile
 - in lettura va trasformato in *entità usabili* all'interno del programma

Attenzione a:

- *formati dei numeri*: separatori di migliaia e decimali?
- *formati delle date/ore*: tipo di formattazione, separatori, ecc.
- *separatori dei vari campi*: quali (gruppi di) caratteri usare?

ESEMPIO: NOTAZIONE XML

- Una sintassi per esprimere linguaggi text-based
 - in questo caso, per descrivere la GUI di una app grafica:

```
<Text text="Welcome"
      GridPane.columnIndex="0" GridPane.rowIndex="0"
      GridPane.columnSpan="2"/>
<Label text="User Name:"
      GridPane.columnIndex="0" GridPane.rowIndex="1"/>
<TextField
      GridPane.columnIndex="1" GridPane.rowIndex="1"/>
<Label text="Password:"
      GridPane.columnIndex="0" GridPane.rowIndex="2"/>
<PasswordField fx:id="passwordField"
      GridPane.columnIndex="1" GridPane.rowIndex="2"/>
```



ESEMPIO: FORMATI «CUSTOM»

- Formati specifici per applicazioni specifiche

- Voti alle elezioni

SEGGI 20

Lista A 9.100.000, Lista E 1.320.000, Lista C 4.880.000,
Lista D 4.100.000, Lista B 1.500.000

- Linee di autobus con tempi di percorrenza e fermate

Linea 32

0, 40, Porta San Mamolo

3, 42, Aldini

5, 44, Porta Saragozza - Villa Cassarini

17, 16, Stazione Centrale

38, 40, Porta San Mamolo

ESEMPIO: FORMATI «CUSTOM»

– Tracking di un volo aereo

UTC;Position;Altitude;Speed;Direction

2019-05-10T10:54:39Z;45.661972,8.726303;1975;183;356

2019-05-10T10:54:49Z;45.67049,8.725822;2450;182;358

2019-05-10T10:55:11Z;45.689159,8.726234;3100;185;1

– Tracciamento pacchi e consegne

Failed;442HHASD882233;2016-08-06T22:08:36.242;non in casa

Succeeded;442HHASD882233;2016-08-06T22:09:01.103;Nonna

Piera;trovato finalmente

– Magazzino di componenti per computer

Ram;Kingston;KVR16LS11/4;19.98;Non-ECC CL11 SODIMM;Y;2016-04-28;-

;4096;DDR3;1600; Cpu;Intel;BX80662I76700K;347.3;Box Core I7-

6700K;Y;2015-04-28;2016-01-01;4;3.5;Y;

Monitor;Benq;GW2270H;104;Display da 21,5'' Full-HD;Y;2015-02-02;-

;1920x1080;16:9;25;

PowerSupply;Thermaltake;Smart SE;54.9;Alimentatore

Modulare;Y;2016-03-04;2016-04-04;530;n.a.;


ESEMPIO: FORMATO «CSV»


- CSV = Comma Separated Values
 - in teoria, la cosa più standard dell'universo: la virgola è la virgola!
 - in realtà, spesso si intende non la «virgola» in sé (concetto assoluto) ma il *separatore di elenco* (concetto relativo, locale)
 - a volte si intendono anche «tabulazioni» o spazi o altro
- Per l'Italia, il separatore di elenco è «;», non «,»
 - questo può portare a risultati inattesi: elenchi esportati con il Locale inglese possono dare errore quando riletti con il Locale italiano
- Ovviamente, il separatore scelto *non* può far parte dei dati
 - in caso sia presente, andrà sostituito da qualcos'altro o «escaped» con qualche carattere speciale → *formati specifici*

ESEMPIO: EXCEL «CSV»

- Caso tipico: Excel







Esporta

 Crea documento PDF/XPS





 **Cambia tipo di file**

Cambia tipo di file

Tipi di file delle cartelle di lavoro

 Cartella di lavoro (*.xlsx) Usa il formato foglio di calcolo di Excel	 Cartella di lavoro di Excel 97-2003 (*.xls) Usa il formato foglio di calcolo Excel 97-2003
 Formato ODS (*.ods) Usa il formato ODS	 Modello (*.xltx) Punto di partenza per i nuovi fogli di calcolo
 Cartella di lavoro con attivazione macro... Foglio di calcolo con attivazione macro	 C...

Altri tipi di file

 Testo (con valori delimitati da tabulazio... Formato testo separato da tabulazioni	 CSV (delimitato dal separatore di elenco... Formato testo con valori separati da virgole
 Testo formattato (delimitato da spazio)... Formato testo con valori separati da spazi	 Salva come altro tipo di file

Sono CSV anche i tab-separated

Sono CSV anche i o space-separated

Virgole..?!?

ESEMPIO: EXCEL «CSV»

- Caso tipico: Excel

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Matricola	Cognome	Nome	codice fiscale	data di nascita	luogo di nascita	provincia	stato	indirizzo	città	provincia	stato	cap
2	12345678	Rossi	Mario	RSSMRA76H12A944I	12/06/1976	Bologna	BO	Italia	via Indipendenza 22	Bologna	BO	Italia	40121
3	12345699	Rossi	Mario	RSSMRA90A01F205Z	01/01/1990	Milano	MI	Italia	piazza del Duomo 1	Milano	MI	Italia	20121

Nome file:

Salva come: CSV delimitato dal separatore di elenco (*.csv)

- il separatore di elenco dipende dalle impostazioni di località
→ per l'Italia, è il «;»

```
Matricola;Cognome;Nome;codice fiscale;data di nascita;luogo di nascita;provincia;stato;indirizzo;città;provincia;stato;cap
12345678;Rossi;Mario;RSSMRA76H12A944I;12/06/1976;Bologna;BO;Italia;via Indipendenza 22;Bologna;BO;Italia;40121
12345699;Rossi;Mario;RSSMRA90A01F205Z;01/01/1990;Milano;MI;Italia;piazza del Duomo 1;Milano;MI;Italia;20121
```

E se un dato conteneva già il «;» ?

ESEMPIO: EXCEL «CSV»

- Caso tipico: Excel con un dato contenente il separatore

	A	B	C	D	E	F	G	H	I	J
1	Matricola	Cognome	Nome	codice fiscale	data di nascita	luogo di nascita	provincia	stato	indirizzo	città
2	12345678	Rossi	Mario	RSSMRA76H12A944I	12/06/1976	Bologna	BO	Italia	via Indipendenza 22; Torre Asinelli 1	Bologna
3	12345699	Rossi	Mario	RSSMRA90A01F205Z	01/01/1990	Milano	MI	Italia	piazza del Duomo 1	Milano

- Cosa succede a quel «;» ?

- nel caso di Excel, quando il file viene salvato in CSV, quel campo dati viene virgolettato

```
e;Nome;codice fiscale;data di nascita;luogo di nascita;provincia;stato;indirizzo;città;provincia;s
ario;RSSMRA76H12A944I;12/06/1976;Bologna;BO;Italia;"via Indipendenza 22; Torre Asinelli 1";Bologna
ario;RSSMRA90A01F205Z;01/01/1990;Milano;MI;Italia;piazza del Duomo 1;Milano;MI;Italia;20121
```

- MA è una scelta di questo specifico formato!
- altri avrebbero potuto anteporre una «\», o inventarsi qualcos'altro
- Importante verificare *sempre* la specifica *esatta* del formato!