

# Appunti di Progettazione del Software

Canale 1 · Prof. Paolo Liberatore

## Capitolo 9: Lettura e scrittura a flussi

**Autore: Tommaso C.**

---

Appunti tratti dalle lezioni in aula e dal materiale del corso

---

Argomenti: Scrittura su file · Lettura per caratteri e per righe · Scanner · Dati CSV e costruttori · Flussi di byte e file ZIP

## Indice

|          |                                                 |           |
|----------|-------------------------------------------------|-----------|
| <b>1</b> | <b>Scrivere su un file</b>                      | <b>1</b>  |
| 1.1      | L'eccezione <code>IOException</code> . . . . .  | 2         |
| 1.2      | Il buffer . . . . .                             | 2         |
| <b>2</b> | <b>Leggere carattere per carattere</b>          | <b>2</b>  |
| <b>3</b> | <b>Leggere per righe: una catena di oggetti</b> | <b>3</b>  |
| 3.1      | La catena . . . . .                             | 4         |
| <b>4</b> | <b>Lo Scanner</b>                               | <b>4</b>  |
| 4.1      | Il delimitatore . . . . .                       | 5         |
| <b>5</b> | <b>Dati CSV: una riga, un oggetto</b>           | <b>6</b>  |
| 5.1      | Più costruttori con lo stesso nome . . . . .    | 7         |
| <b>6</b> | <b>Flussi di byte: file ZIP e flussi propri</b> | <b>8</b>  |
| 6.1      | Un flusso scritto dal programma . . . . .       | 9         |
| <b>7</b> | <b>Esercizi</b>                                 | <b>10</b> |
| <b>8</b> | <b>Soluzioni</b>                                | <b>11</b> |

## 1 Scrivere su un file

### Definizione – Flusso

Un **flusso** è una successione di dati che il programma legge da una sorgente, per esempio un file, oppure scrive su una destinazione, un dato dopo l'altro. In Java un flusso è un oggetto: leggere e scrivere vuol dire chiamarne i metodi.

### Proposizione – Scrivere su un file

1. `new FileWriter("prova.txt")` apre il file in scrittura: lo crea se non esiste, altrimenti lo svuota.
2. `fw.write(s)` scrive la stringa `s`. L'a capo non si aggiunge da solo: va scritto come `\n`.
3. `fw.flush()` manda al file i caratteri ancora in memoria.
4. `fw.close()` chiude il file, dopo aver fatto **flush**.

```
import java.io.FileWriter;
import java.io.IOException;

public class Scrittura {
    public static void main(String[] args) throws IOException {
        FileWriter fw = new FileWriter("prova.txt");
        fw.write("prima linea\n");
        fw.write("seconda linea\n");
        fw.write("terza linea\n");
        fw.flush();
        fw.close();
    }
}
```

Un nome senza cartella indica un file nella cartella in cui il programma viene eseguito: in Eclipse è la cartella del progetto. Senza i `\n` le tre stringhe finiscono su una sola riga, prima lineaseconda lineaterza linea.

## 1.1 L'eccezione `IOException`

### Definizione – `IOException` e `throws`

Un'operazione su un file può fallire, per esempio perché il file non esiste o mancano i permessi sulla cartella. In quel caso genera un'eccezione `IOException`. Un metodo che non gestisce l'errore lo dichiara nell'intestazione:

```
public static void main(String[] args) throws IOException
```

Se l'errore avviene, il programma si interrompe come con le eccezioni del Capitolo 4. Senza `throws` il programma non compila: `javac` segnala *unreported exception IOException; must be caught or declared to be thrown*. Le `RuntimeException` del Capitolo 4 invece non si dichiarano.

## 1.2 Il buffer

### Proposizione – `flush` e `close`

`write` può tenere i caratteri in un **buffer**, una zona di memoria, e mandarli al file più tardi, tutti insieme. I caratteri sono certamente nel file solo dopo `flush()` o `close()`.

Il buffer serve all'efficienza: scrivere sul disco pochi blocchi grandi costa meno che molti caratteri singoli. Se il programma termina senza `flush` né `close`, il file può restare vuoto: con le sole tre `write` dell'esempio, `prova.txt` ha 0 byte.

## 2 Leggere carattere per carattere

### Definizione – `FileReader`

`new FileReader("prova.txt")` apre il file in lettura. Il metodo `read()` restituisce il carattere successivo come `int`, oppure `-1` se i caratteri sono finiti.

```
FileReader fr = new FileReader("prova.txt");
for (int i = 0; i < 10; i++) {
    int c = fr.read();
    System.out.print((char) c);
}
fr.close();
```

Il codice stampa **prima line**, i primi dieci caratteri. Senza il cast (**char**) stamperebbe i codici numerici, **112** per **p**. Anche l'apertura e la chiusura in lettura possono generare **IOException**, quindi anche qui il **main** dichiara **throws IOException**.

Il risultato è un **int** perché **-1** deve distinguersi da ogni carattere: i codici dei caratteri vanno da 0 a 65535, e **-1** non è fra questi. Gli altri metodi di **FileReader** leggono più caratteri in un array, ma non righe né numeri.

### 3 Leggere per righe: una catena di oggetti

#### Definizione – **BufferedReader**

**new BufferedReader(fr)** riceve nel costruttore il **FileReader** e ne conserva il riferimento. Il metodo **readLine()** restituisce la riga successiva, senza il terminatore, oppure **null** se le righe sono finite.

Per costruire la riga, **br** chiede caratteri a **fr** finché non trova il terminatore di riga. Leggere tutto il file richiede un ciclo.

#### Proposizione – Ciclo di lettura

1. Si legge una riga con **readLine()**.
2. Se è **null**, ci si ferma.
3. Altrimenti si elabora la riga e si torna al punto 1.

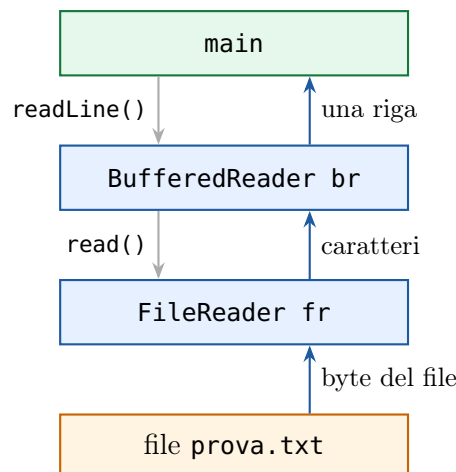
```
FileReader fr = new FileReader("prova.txt");
BufferedReader br = new BufferedReader(fr);
while (true) {
    String s = br.readLine();
    if (s == null) {
        break;
    }
    System.out.println(s);
}
br.close();
```

Lo schema resta lo stesso in ogni programma che legge un file per righe: cambia solo che cosa si fa con la riga. Si trova anche la forma compatta **while ((s = br.readLine()) != null)**, che fa lo stesso in modo meno leggibile.

### 3.1 La catena

#### Definizione – Filtro e catena di flussi

Un **filtro** è un oggetto che riceve nel costruttore un altro flusso, gli chiede i dati e li restituisce in una forma più comoda. Più filtri formano una **catena**: ognuno legge dal precedente, e il primo legge la sorgente.



Il `main` chiama solo `br.readLine()`: `fr` e il file stanno dietro `br`. Come `fr` legga il file dal disco non riguarda il programma. Separare i compiti permette di costruire sulla lettura per caratteri altre letture, per righe, per parole o per interi.

## 4 Lo Scanner

#### Definizione – Scanner

Uno **Scanner** (`java.util`) legge da un **FileReader**, da un flusso di byte (Sezione 6) o da una stringa, e restituisce il dato successivo nel tipo chiesto. I dati sono separati dal **delimitatore**, che di norma è uno spazio o un a capo.

| Metodo                                                                          | Restituisce                                              |
|---------------------------------------------------------------------------------|----------------------------------------------------------|
| <code>next()</code>                                                             | la parola successiva, come <code>String</code>           |
| <code>nextInt()</code>                                                          | l'intero successivo, come <code>int</code>               |
| <code>nextLine()</code>                                                         | il resto della riga, senza il terminatore                |
| <code>hasNext()</code> , <code>hasNextInt()</code> , <code>hasNextLine()</code> | <code>true</code> se c'è una parola, un intero, una riga |
| <code>useDelimiter(d)</code>                                                    | imposta il delimitatore <code>d</code>                   |
| <code>close()</code>                                                            | chiude lo scanner e il flusso da cui legge               |

Il file `interi.txt` contiene `4 3 6` sulla prima riga e `2 9` sulla seconda. Lo scanner legge un intero alla volta, comunque siano disposti:

```
FileReader fr = new FileReader("interi.txt");
Scanner sc = new Scanner(fr);
int somma = 0;
while (sc.hasNextInt()) {
    somma = somma + sc.nextInt();
}
sc.close();
System.out.println(somma);
```

Il programma stampa 24. Con un `BufferedReader` e `Integer.parseInt` la somma funziona solo con un intero per riga (Esercizio 1): qui `parseInt` riceverebbe `"4 3 6"` e genererebbe `NumberFormatException`.

I metodi si combinano secondo la struttura del file. Il file `misto.txt` ha la riga `Numeri e parole`, poi le righe `4 3 6` e `2 9`, poi uno due tre:

```
FileReader fr = new FileReader("misto.txt");
Scanner sc = new Scanner(fr);
System.out.println(sc.nextLine());
int somma = 0;
while (sc.hasNextInt()) {
    somma = somma + sc.nextInt();
}
System.out.println(somma);
while (sc.hasNext()) {
    System.out.println(sc.next());
}
sc.close();
```

Il programma stampa `Numeri e parole`, poi 24, poi uno, due e tre su tre righe. Il primo ciclo si ferma su `uno`, che non è un intero, e il secondo legge le parole rimaste.

## 4.1 Il delimitatore

### Proposizione – Cambiare il delimitatore

Dopo `sc.useDelimiter(",")` i dati sono separati dalle virgole, e solo da quelle: il nuovo delimitatore sostituisce spazi e a capo. Su `"uno, due, tre"` lo scanner restituisce `uno`, `due` e `tre`.

### Osservazione – Errori tipici con la virgola

- **Spazio dopo la virgola.** Su "Rossi, 20" il secondo dato è " 20", con lo spazio, e `hasNextInt()` dà false.
- **A capo.** Su due righe `a,b` e `c,d` un dato è `b` seguito da `a` capo e `c`, perché l'a capo non separa più.

Per questo un file con più righe di dati separati da virgole si legge una riga alla volta (Sezione 5).

## 5 Dati CSV: una riga, un oggetto

### Definizione – File CSV

In un file **CSV** (*comma-separated values*) ogni riga descrive un elemento, e i suoi campi sono separati da virgole, come in `Mario,Rossi,20`.

### Proposizione – Leggere un file CSV

1. Un `BufferedReader` legge il file una riga alla volta.
2. `new Scanner(riga)` con `useDelimiter(",")` scandisce la riga.
3. `next()` e `nextInt()` restituiscono i campi, nell'ordine.

Uno scanner costruito su una stringa chiede i caratteri alla stringa, non a un file. Conviene mettere la scansione della riga nel costruttore della classe dei dati, che riceve la riga:

```
import java.util.Scanner;

public class Persona {
    String nome;
    String cognome;
    int eta;

    public Persona(String nome, String cognome, int eta) {
        this.nome = nome;
        this.cognome = cognome;
        this.eta = eta;
    }

    public Persona(String s) {
        Scanner sc = new Scanner(s);
        sc.useDelimiter(",");
        this.nome = sc.next();
```

```
        this.cognome = sc.next();
        this.eta = sc.nextInt();
        sc.close();
    }

    public String toString() {
        return nome + " " + cognome + " " + eta;
    }
}
```

Il costruttore che riceve la riga fa l'inverso di `toString`: `toString` trasforma una persona in una stringa, il costruttore una stringa in una persona. Il programma raccoglie le persone in una lista:

```
FileReader fr = new FileReader("persone.csv");
BufferedReader br = new BufferedReader(fr);
LinkedList<Persona> lista = new LinkedList<Persona>();
while (true) {
    String s = br.readLine();
    if (s == null) {
        break;
    }
    lista.add(new Persona(s));
}
br.close();
System.out.println(lista);
```

Con le righe `Mario,Rossi,20` e `Anna,Bianchi,22` il programma stampa `[Mario Rossi 20, Anna Bianchi 22]`: come nel Capitolo 2, la lista mostra fra parentesi quadre il `toString` di ogni elemento. I caratteri passano da `fr` a `br`, che li riunisce in una riga, e lo scanner nel costruttore trasforma la riga in una persona.

## 5.1 Più costruttori con lo stesso nome

### Definizione – Sovraccarico

Una classe può avere più costruttori, o più metodi con lo stesso nome, se le liste dei parametri differiscono per numero o per tipo. È il **sovraccarico**: a ogni chiamata Java usa la versione i cui parametri corrispondono agli argomenti.

`new Persona(s)` usa il costruttore con una stringa, `new Persona("Luca", "Verdi", 21)` quello con tre parametri. Il sovraccarico è diverso dalla ridefinizione del Capitolo 3, in cui una sottoclasse riscrive un metodo con gli stessi parametri. Per questo `equals(Punto p)` nel Capitolo 6 non ridefinisce `equals(Object)`: lo sovraccarica.

## 6 Flussi di byte: file ZIP e flussi propri

### Definizione – FileInputStream

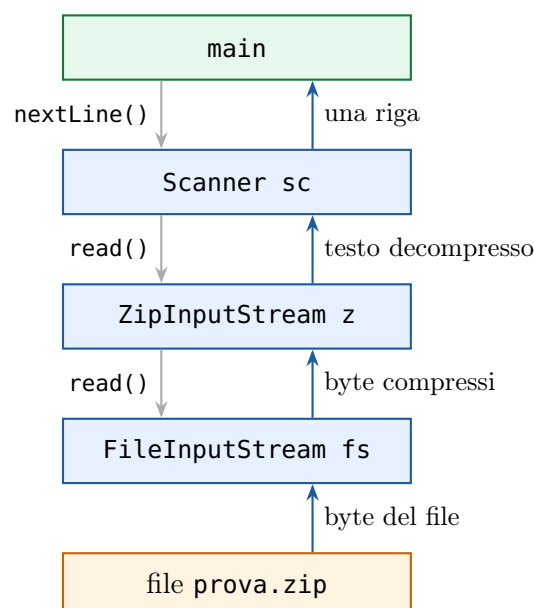
`new FileInputStream("prova.zip")` legge un file come `FileReader`, ma restituisce byte invece di caratteri: `read()` dà un valore da 0 a 255, oppure `-1` alla fine. I flussi di byte sono sottoclassi di `InputStream`, quelli di caratteri di `Reader`.

Un archivio ZIP contiene dati compressi, non testo: per leggerlo serve `ZipInputStream`, che li decompime e accetta solo un `InputStream`. Con un `FileReader` il programma non compila.

### Proposizione – Leggere un file in un archivio ZIP

1. Un `FileInputStream` legge i byte dell'archivio.
2. Uno `ZipInputStream`, che lo riceve nel costruttore, li decompime.
3. `getNextEntry()` passa al file successivo dell'archivio: la prima chiamata al primo.
4. Uno `Scanner` sullo `ZipInputStream` legge quel file.

```
FileInputStream fs = new FileInputStream("prova.zip");
ZipInputStream z = new ZipInputStream(fs);
z.getNextEntry();
Scanner sc = new Scanner(z);
System.out.println(sc.nextLine());
System.out.println(sc.nextLine());
System.out.println(sc.nextLine());
sc.close();
```



La catena ha tre oggetti fra il file e il **main**. Ognuno conosce solo il precedente: lo scanner non sa che i dati erano compressi.

## 6.1 Un flusso scritto dal programma

### Proposizione – Un flusso di input proprio

**InputStream** è una classe astratta con un solo metodo astratto, **read()**. Una sottoclasse che lo definisce è un flusso come gli altri: si può passare a uno **Scanner**. Il suo **read()** restituisce un byte, da 0 a 255, oppure **-1** quando il flusso è finito, e da lì in poi sempre **-1**.

```
import java.io.InputStream;

public class MioInputStream extends InputStream {
    boolean finito = false;

    public int read() {
        if (finito || Math.random() < 0.01) {
            finito = true;
            return -1;
        }
        if (Math.random() < 0.2) {
            return ' ';
        }
        return 'A';
    }
}
```

```
MioInputStream mis = new MioInputStream();
Scanner sc = new Scanner(mis);
while (sc.hasNext()) {
    System.out.println(sc.next());
}
sc.close();
```

Ogni esecuzione stampa parole diverse, come **AAAA**, **AAAAAA**, **A**. Lo scanner non sa da dove vengono i byte: li chiede a **read()** e compone le parole finché non riceve **-1**. Un filtro come **ZipInputStream** funziona allo stesso modo: il suo **read()** chiede byte al flusso ricevuto nel costruttore e li restituisce trasformati.

### Riepilogo

- **Scrittura:** `FileWriter` crea il file o lo svuota; `write` non va a capo, serve `\n`.
- **Buffer e IOException:** i caratteri sono nel file solo dopo `flush()` o `close()`; chi usa file senza gestire l'errore dichiara `throws IOException`.
- **FileReader:** `read()` dà il carattere come `int`, `-1` alla fine.
- **Catena:** `new BufferedReader(fr); readLine()` dà una riga o `null`, in un ciclo che si ferma su `null`.
- **Scanner:** legge da un flusso o da una stringa; `next`, `nextInt`, `nextLine` con i loro `has`; `useDelimiter`.
- **CSV:** una riga alla volta con `readLine()`, scandita con uno `Scanner` nel costruttore `Persona(String)`.
- **Sovraccarico:** costruttori o metodi con lo stesso nome e parametri diversi.
- **Flussi di byte:** `FileInputStream`, `ZipInputStream` con `getNextEntry()`, `Scanner`; un flusso proprio definisce `read()`.

## 7 Esercizi

### Esercizio 1 – Il sommatore

Il file `interi.txt` contiene un intero per riga: 4, 3, 6, 2, 9. Scrivere un programma che legga il file con un `BufferedReader` e stampi la somma degli interi.

*Soluzione a pagina 11 →*

## 8 Soluzioni

### Soluzione dell'Esercizio 1 – Il sommatore

Il ciclo è quello della Sezione 3. Ogni riga è una stringa, e `Integer.parseInt` la converte in intero (Capitolo 8) prima di sommarla.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class Sommatore {
    public static void main(String[] args) throws IOException {
        FileReader fr = new FileReader("interi.txt");
        BufferedReader br = new BufferedReader(fr);
        int somma = 0;
        while (true) {
            String s = br.readLine();
            if (s == null) {
                break;
            }
            somma = somma + Integer.parseInt(s);
        }
        br.close();
        System.out.println(somma);
    }
}
```

Il programma stampa  $4 + 3 + 6 + 2 + 9 = 24$ .

[← Torna all'Esercizio 1, pagina 10](#)