Note sulla lettura e scrittura sequenziale di file di testo (e canali di input/output)

Questa pagina propone alcuni semplici schemi per leggere e scrivere file di testo in modo sequenziale. Per altri modi e una trattazione più ampia ed esaustiva, si faccia riferimento alla sezione I/O Streams del Java Tutorial[∞] e in particolare al paragrafo Line-Oriented I/O della sezione Character Streams[∞].

Lettura sequenziale di file di testo (e canali di input)

Passo 1: apertura file in lettura

Per leggere un file di testo avente come nome la stringa `s`, creare innanzitutto un oggetto `java.io.FileReader` come segue (dopo aver importato `java.io.*`):

```
FileReader f;

try {
    f = new FileReader(s);
}
catch(FileNotFoundException e){
    // errore: non è possibile aprire il file s
}
```

Si noti che il costruttore `FileReader(s)` lancia una eccezione verificata `FileNotFoundException` se non è possibile aprire il file `s`.

Passo 2: creazione canale di lettura associato al file da leggere

Poiché non è possibile usare direttamente l'oggetto `FileReader f` per leggere dal file, bisogna creare un secondo oggetto di classe `java.io.BufferedReader` passando `f` al costruttore `BufferedReader` in modo da creare un canale di lettura associato al file.

```
BufferedReader b = new BufferedReader(f);
```

Nota importante: la parte introduttiva della documentazione standard della classe `BufferedReader` fornisce un esempio di creazione di un canale di lettura associato a un file (passi 1 e 2).

Passo 3: lettura di stringhe dal canale di lettura

A questo punto, invocando ripetutamente il metodo `readLine()` sull'oggetto `BufferedReader b` si leggono una alla volta le righe del file. Quando `readLine()` restituisce `null` vuole dire che le righe del file sono state tutte lette e l'oggetto `BufferedReader` ha esaurito il suo scopo.

Ad esempio, il seguente frammendo di codice stampa a video tutte le righe del file:

```
try {
    for (;;) {
        String r = b.readLine(); // legge la prossima riga del file
        if (r == null) break; // esce dal ciclo se le righe sono finite
        System.out.println(r); // stampa a video la riga letta dal file
    }
}
catch(IOException e){
    // Errore nella lettura del file
}
```

Si noti che il metodo `readLine` potrebbe lanciare una eccezione verificata di tipo `IOException`.

Passo 4: chiusura canale di input associato al file

Quando la lettura del file è stata ultimata, è buona norma chiudere il canale di input associato al file invocando il metodo `close()`:

```
b.close();
```

Nota: fare attenzione a non chiamare `close()` su un canale `BufferedReader` non associato a un file, ma ad esempio al canale standard `System.in`.

Esempio 1

Scrivere una classe pubblica `LetturaFile` con un metodo pubblico `void stampaFile(String s)` che, dato il nome di un file `s`, ne stampa a video tutte le righe, oppure lancia un'eccezione non verificata se si incontrano errori nella lettura.

LetturaFile.java

```
import java.io.*;

public class LetturaFile {
    public static void stampaFile(String s){
        BufferedReader b = null;

        try {
            // apre file
            b = new BufferedReader(new FileReader(s));

            // legge file e scrive le righe lette su System.out
            for (;;) {
                String r = b.readLine(); // legge la prossima riga del file
                if (r == null) break; // esce dal ciclo se le righe sono finite
                System.out.println(r); // stampa a video la riga letta dal file
            }

            // chiude il canale di input
            b.close();
        }
        catch(FileNotFoundException e) { // cattura errori prodotti da new FileReader(s
```

```
            throw new RuntimeException("Errore: impossibile aprire il file " + s);
        }
        catch(IOException e) { // cattura errori prodotti da readLine()
            if (b!= null) b.close();
            throw new RuntimeException("Errore di lettura del file " + s);
        }
    }
}
```

Notare che `FileNotFoundException` è una sottoclasse di `IOException` e quindi togliendo il `catch(FileNotFoundException e)` il programma sarebbe comunque corretto, solo che darebbe lo stesso errore sia in caso di errore di apertura file (`new FileReader(...)`) che di lettura file (`b.readLine()`).

Programma di prova:

```
    public static void main(String[] args){
        stampaFile("LetturaFile.java");
    }
```

Esempio 2

Una variante dell'esercizio precedente assume che il metodo non prenda come parametro un nome di file, ma un canale di lettura `BufferedReader` assumendo che sia stato già creato al momento dell'invocazione.

Scrivere una classe pubblica `LetturaFile` con metodo pubblico `void stampaCanaleInput(BufferedReader b)` che, dato un canale di lettura `b`, ne stampa a video tutte le righe, oppure lancia un'eccezione non verificata se si incontrano errori nella lettura.

LetturaFile.java

```
import java.io.*;

public class LetturaFile {
    public static void stampaCanaleInput(BufferedReader b){
        try {
            // legge file e scrive le righe lette su System.out
            for (;;) {
                String r = b.readLine(); // legge la prossima riga del file
                if (r == null) break; // esce dal ciclo se le righe sono finite
                System.out.println(r); // stampa a video la riga letta dal file
            }
        }
        catch(IOException e) { // cattura errori prodotti da readLine()
            throw new RuntimeException("Errore di I/O");
        }
    }
}
```

Programma di prova:

Programma di prova

```
    public static void main(String[] args){
        BufferedReader b;

        try { // prova ad aprire il file
            b = new BufferedReader(new FileReader("LetturaFile.java"));
        }
        catch(FileNotFoundException e) { // cattura errori prodotti da new FileReader()
            throw new RuntimeException("Errore: impossibile aprire il file");
        }

        stampaCanaleInput(b);
        b.close();
    }
```

Scrittura sequenziale di file di testo (e canali di output)

Passo 1: creazione canale di scrittura associato al file

Per creare un file di testo con nome `miofile.txt`, creare un oggetto `java.io.PrintStream` come segue (dopo aver importato `java.io.*`):

```
PrintStream f;

try {
    f = new PrintStream("miofile.txt");
}
catch(FileNotFoundException e){
    throw new RuntimeException("errore apertura file");
}
```

Si noti che il costruttore `PrintStream(s)` (si legga la documentazione):

- crea un nuovo file con il nome `s` se il file non esiste ancora;
- apre il file in scrittura se il file esiste, cancellandone il contenuto precedente;
- lancia una eccezione verificata `FileNotFoundException` se non è possibile aprire il file per la scrittura (la documentazione recita: "If the given file object does not denote an existing, writable regular file and a new regular file of that name cannot be created, or if some other error occurs while opening or creating the file").

Passo 2: scrittura sul canale di output associato al file

Da questo punto in poi, l'oggetto `p` può essere usato per scrivere con i metodi `print` e `println` della classe `PrintStream` (si noti che `System.out` è un oggetto `PrintStream`). Ad esempio:

```
p.println("prova scrittura su file");
```

Passo 3: chiusura canale di output associato al file

Quando la scrittura del file è stata ultimata, è buona norma chiudere il canale di output associato al file invocando il metodo `close()`:

```
p.close();
```

Nota: fare attenzione a non chiamare `close()` su un canale `PrintStream` non associato a un file, ma ad esempio al canale standard `System.out`:

```
PrintStream p = System.out; // System.out è un riferimento di tipo PrintStream al termi
p.close(); // chiude il terminale di output
System.out.println("questa stringa non verrà mai stampata");
```

Esempio 1

Scrivere una classe pubblica `ScritturaFile` con un metodo pubblico `void scriviSuFile(Object[] v, String s)` che, dato un array `v` di oggetti e una stringa `s`, crea un file di nome `s` contenente le stringhe associate agli oggetti dell'array, oppure lancia un'eccezione non verificata se si incontrano errori nella scrittura del file.

ScritturaFile.java

```
import java.io.*;

public class ScritturaFile {
    public static void scriviSuFile(Object[] v, String s){

        PrintStream p;

        try {
             p = new PrintStream(s);
        }
        catch(FileNotFoundException e){
            throw new RuntimeException("errore accesso file");
        }

        for (int i=0; i<v.length; i++) p.println(v[i]);

        p.close();
    }

    public static void main(String[] args){
        scriviSuFile(new String[]{"uno", "due", "tre"}, "miofile.txt");
    }
}
```

Programma di prova:

```
    public static void main(String[] args){
        scriviSuFile(new String[]{"uno", "due", "tre"}, "miofile.txt");
```

```
    }
```

Esempio 2

Una variante dell'esercizio precedente assume che il metodo non prenda come parametro un nome di file, ma un canale di scrittura `PrintStream` assumendo che sia stato già creato al momento dell'invocazione (ad esempio associandolo a un file o al terminale di output `System.out`).

Scrivere una classe pubblica `ScritturaFile` con un metodo pubblico `void scriviSuCanaleOutput(Object[] v, PrintStream p)` che, dato un array `v` di oggetti e un canale di output `p`, scrive su quel canale le stringhe associate agli oggetti dell'array, oppure lancia un'eccezione non verificata se si incontrano errori nella scrittura del file.

ScritturaFile.java

```
import java.io.*;

public class ScritturaFile {
    public static void scriviSuCanaleOutput(Object[] v, PrintStream p){
        for (int i=0; i<v.length; i++) p.println(v[i]);
    }
}
```

Nota importante: sarebbe errato fare `p.close()` perché il canale di output `p` potrebbe dover rimanere ancora in uso (esempio ovvio: se `p==System.out`).

Programma di prova:

```
    public static void main(String[] args){
        scriviSuCanaleOutput(new String[]{"uno", "due", "tre"}, System.out);
    }
```