Azzolini Riccardo 2020-02-20

Stream di I/O

1 Stream di ingresso e uscita

Il pacchetto `java.io` definisce le operazioni di ingresso e uscita in termini di **stream** (**flussi**): sequenze ordinate di dati che hanno una **sorgente** (per gli stream di ingresso) o una **destinazione** (per gli stream di uscita).

- Per ricevere dati in ingresso, un'applicazione apre uno stream di input, collegato a una sorgente che può essere in memoria, sul disco, o anche remota, e legge sequenzialmente le informazioni da tale stream.
- Analogamente, per esportare dati, un'applicazione apre uno stream verso una destinazione (che, anche in questo caso, può essere in memoria, sul disco, o remota) e vi scrive sequenzialmente le informazioni.

Gli stream nascondono i dettagli del sistema operativo sottostante, rendendo (per quanto possibile) trasparente la gestione di file, ecc.

1.1 Schema delle operazioni di I/O

In genere, le operazioni sugli stream ricalcano i seguenti schemi:

- Lettura:

```
creazione dello stream
while (ci sono informazioni da leggere)
    leggi
chiusura dello stream
```

- Scrittura:

```
creazione dello stream
while (ci sono informazioni da scrivere)
    scrivi
chiusura dello stream
```

2 Tipi di stream

Gli stream forniti dal pacchetto `java.io` sono di due tipi:

Flussi di byte (byte stream): gestiscono come unità di informazione i singoli byte (8 bit), quindi non fanno alcuna ipotesi sul tipo dei dati trattati, e, di conseguenza, sono "universali": possono essere usati per qualsiasi tipo di informazione (immagini, audio, testo, ecc.).

Le classi che realizzano questi stream sono chiamate:

- *stream di ingresso* (*input stream*);
- *stream di uscita* (*output stream*).

Flussi di caratteri (character stream): l'unità di informazione gestita sono caratteri Unicode a 16 bit. Essi permettono di effettuare l'I/O testuale in modo più conveniente, senza dover gestire manualmente la codifica dei caratteri, cosa che sarebbe invece necessaria lavorando direttamente con i flussi di byte.[1]

Le classi corrispondenti sono indicate come:

- *lettori* (*reader*);
- *scrittori* (*writer*).

In pratica, si usano gli stream di caratteri per i dati testuali, e i byte stream per tutti gli altri tipi di dati. Come caso particolare, i dati composti da caratteri rappresentati con un solo byte ciascuno (ad esempio ASCII) possono essere facilmente gestiti con entrambi i tipi di stream.

3 Errori

Le operazioni di I/O possono sempre fallire (ad esempio, a causa di file non esistenti, permessi non adeguati, ecc.).

La maggior parte delle classi di `java.io` segnalano gli errori lanciando un'*eccezione* di tipo `IOException` (invece, in alcuni casi particolari, un errore viene segnalato *cambiando lo stato* dello stream).

[1] I flussi di caratteri sono solitamente implementati aggiungendo uno strato di traduzione sopra ai flussi di byte.

4 Lettura di file binari

Per leggere file in formato binario, si utilizza un'istanza della classe `FileInputStream`.

- Per aprire un file, il cui nome deve essere conosciuto, si utilizza uno dei costruttori disponibili:

  ```
  public FileInputStream(String name) throws FileNotFoundException;
  ```

  Ad esempio:

  ```
  FileInputStream in = new FileInputStream(fileName);
  ```

  `FileNotFoundException`, sollevata da questo costruttore quando il file non esiste, o non può essere letto per un qualsiasi altro motivo, è un'eccezione più specifica (sottoclasse) di `IOException` (quindi è possibile, ma non necessario, gestirla separatamente).

- Per la lettura, questa classe mette a disposizione, tra gli altri, il metodo `read()`,

  ```
  public int read() throws IOException;
  ```

  che legge un singolo byte, e restituisce un `int`:

  - il byte letto, se disponibile, viene inserito negli 8 bit meno significativi dell'`int`, che ha quindi un valore compreso tra 0 e 255;
  - se, invece, non sono disponibili byte perché si è raggiunta la fine dello stream (**EOF**, End Of File), viene restituito il valore -1.[2]

- Una volta terminate le operazioni di ingresso, il file deve essere chiuso, invocando il metodo `close()`. Eventuali tentativi di lettura su uno stream già chiuso sollevano una `IOException`.

5 Scrittura su file binari

La classe usata per scrivere un file in formato binario è `FileOutputStream`.

- Il file da scrivere viene aperto mediante il costruttore

  ```
  public FileOutputStream(String name) throws FileNotFoundException;
  ```

  Se un file con il nome specificato non esiste, esso viene creato. Altrimenti, il file esistente viene sovrascritto.[3]

[2] La necessità di restituire un valore speciale (che non possa essere confuso con un byte valido) per segnalare la fine dello stream è il motivo per cui il tipo restituito da questo metodo è `int`, e non `byte`.

[3] Esiste anche un costruttore alternativo, che permette di aggiungere nuovi dati alla fine di un file esistente invece di sovrascriverlo.

- Un singolo byte può essere scritto attraverso il metodo `write(int b)`, che trasferisce nel file solo gli 8 bit meno significativi dell'`int` passato come parametro (mentre i restanti 24 bit vengono ignorati).
- Quando non è più necessario, il file deve essere chiuso, mediante il metodo `close()`.

  Nel caso degli output stream, la chiusura è particolarmente importante. Spesso, infatti, le operazioni di scrittura non vengono effettuate immediatamente: i dati sono semplicemente inseriti in un buffer, e vengono poi trasferiti nel file solo quando tale buffer si riempie. Chiudendo il file, si forza il trasferimento di eventuali dati rimasti nel buffer, che, altrimenti, potrebbero andare persi.[4]

6 Esempio: copia di un file

```
import java.io.*;

public class CopyBin {
    public static void main(String[] args) throws IOException {
        FileInputStream in = new FileInputStream(args[0]);
        FileOutputStream out = new FileOutputStream(args[1]);
        int b;
        while ((b = in.read()) != -1) {
            out.write(b);
        }
        out.close();
        in.close();
    }
}
```

7 Lettura di file di testo

Per la lettura di dati testuali da un file, è disponibile un particolare reader chiamato `FileReader`:

```
public class FileReader extends InputStreamReader;
```

Esso estende la classe `InputStreamReader`, che "aggiunge" la gestione dei caratteri a un `InputStream`.

- Uno dei costruttori di questa classe è:

[4] Lo svuotamento del buffer può anche essere effettuato manualmente, in qualsiasi momento, invocando il metodo `flush()`.

```
public FileReader(String fileName) throws FileNotFoundException;
```

- Il metodo

  ```
  public int read() throws IOException;
  ```

  permette di leggere un singolo carattere, che viene inserito nei 16 bit meno significativi dell'`int` restituito (il quale assume quindi valori compresi tra 0 e 65535). Se, invece, si è raggiunta la fine del file, viene restituito il valore -1.

- Con il metodo

  ```
  public int read(char[] buf) throws IOException;
  ```

  è possibile leggere una sequenza di caratteri, che vengono inseriti in posizioni successive dell'array passato come argomento. Il valore restituito è il numero di caratteri letti, che è pari, al massimo, alla lunghezza dell'array, ma può anche essere inferiore, a seconda dei caratteri disponibili nello stream. Se, invece, al momento dell'invocazione non era disponibile alcun carattere, perché si era già raggiunta la fine dello stream, questo metodo restituisce il valore -1.

- Come gli altri stream, anche questo deve essere chiuso al termine delle operazioni, invocando il metodo `close()`.

7.1 Esempio: visualizzazione di un file di caratteri

```
import java.io.*;

public class ViewCharFile {
    public static void main(String[] args) throws IOException {
        FileReader frd = new FileReader(args[0]);
        int i;
        while ((i = frd.read()) != -1) {
            System.out.print((char)i);
        }
        frd.close();
    }
}
```

8 Scrittura di file di testo

La scrittura dei file di testo avviene mediante la classe `FIleWriter`, che ha un'interfaccia simile agli altri stream visti finora.

8.1 Esempio

```
import java.io.*;

public class FileWriterExample {
    public static void main(String[] args) {
        FileWriter fileWriter = null;
        try {
            try {
                fileWriter = new FileWriter("file.txt");
                fileWriter.write('C');
                fileWriter.write('i');
                fileWriter.write('a');
                fileWriter.write('o');
            } finally {
                if (fileWriter != null) {
                    fileWriter.close();
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Questo esempio mostra anche una semplice gestione delle eccezioni. In particolare, la chiusura dello stream è messa nel blocco `finally`, in modo da assicurare che essa venga eseguita indipendentemente dalle eccezioni. Se, però, l'eccezione è stata causata proprio dalla creazione del `FileWriter`, allora la variabile corrispondente rimane nulla, ed è quindi necessario controllare che non lo sia prima di invocare il metodo `close()`, al fine di evitare una `NullPointerException`.

L'uso di `try` – `catch` e `try` – `finally` annidati, invece che di una singola struttura `try` – `catch` – `finally`, serve a gestire anche le eventuali eccezioni generate nel blocco `finally`.

9 Lettura e scrittura di una riga alla volta

Siccome i file di caratteri sono generalmente organizzati per righe, leggere e scrivere una riga alla volta è più comodo e semplice. Inoltre, è anche più efficiente, poiché ogni operazione di lettura/scrittura ha un costo, quindi conviene effettuare poche operazioni su più caratteri che tante operazioni su singoli caratteri.

Una *riga* è una sequenza di caratteri terminati da un'apposita stringa *EndOfLine* (terminatore di riga, letteralmente "fine riga"), la quale varia a seconda del sistema operativo. Ad esempio:

- Windows usa `"\r\n"`;
- Unix e derivati (Linux, macOS, ecc.) usano `"\n"`;
- il sistema Mac OS "classico" (cioè fino a Mac OS 9) usava `"\r"`.

9.1 Classe `BufferedReader`

Per la lettura di righe, si usa la classe `BufferedReader`, il cui costruttore,

```
public BufferedReader(Reader in);
```

riceve come argomento un `Reader` esistente, al quale, in pratica, viene "aggiunta" la capacità di leggere una riga alla volta, attraverso il metodo

```
public String readLine() throws IOException;
```

Esso restituisce una stringa contenente la riga letta, *senza* i caratteri terminatori, o `null` se, al momento dell'invocazione, non era disponibile alcuna riga di testo perché era stata già raggiunta la fine dello stream.

A prescindere dal sistema operativo, in lettura vengono riconosciuti tutti e tre i possibili terminatori di riga (`"\n"`, `"\r"` e `"\r\n"`).

9.1.1 Esempio: visualizzazione di un file di caratteri

```
import java.io.*;

public class ViewCharFile2 {
    public static void main(String[] args) throws IOException {
        FileReader frd = new FileReader(args[0]);
        BufferedReader bfr = new BufferedReader(frd);
        String str;
        while ((str = bfr.readLine()) != null) {
            System.out.println(str);
        }
        bfr.close();
        frd.close();
    }
}
```

Osservazioni:

- Nell'esempio `ViewCharFile`, veniva usato il metodo `print` per stampare i caratteri. Infatti, lavorando un carattere alla volta, si leggono e stampano anche eventuali caratteri di fine riga, quindi viene naturalmente riprodotta la suddivisione in righe del file visualizzato.

  Invece, in questo caso, la lettura per righe rimuove i terminatori, quindi è necessario usare il metodo `println`, per reintrodurli in fase di visualizzazione.

- La chiusura del `FileReader` e del `BufferedReader` deve avvenire in ordine inverso di apertura.

9.2 Classe `PrintWriter`

La classe `PrintWriter` permette la scrittura di caratteri una riga alla volta, analogamente a ciò che `BufferedReader` fa per la lettura, e, inoltre, permette anche la stampa di dati di altri tipi.

Uno dei suoi costruttori, analogo a quello di `BufferedReader`, è:

```
public PrintWriter(Writer out);
```

Le istanze di questa classe mettono a disposizione una serie di metodi `print` e `println`, che convertono l'argomento in una stringa e trasferiscono quest'ultima in uscita. Nel caso di `println`, viene aggiunto un terminatore di riga alla fine della stringa.

9.3 Esempio: copia di un file di caratteri

```
import java.io.*;

public class CopyChar {
    public static void main(String[] args) throws IOException {
        FileReader frd = new FileReader(args[0]);
        BufferedReader bfr = new BufferedReader(frd);
        FileWriter fwr = new FileWriter(args[1]);
        PrintWriter pwr = new PrintWriter(fwr);
        String str;
        while ((str = bfr.readLine()) != null) {
            pwr.println(str);
        }
        bfr.close();
        frd.close();
        pwr.close();
        fwr.close();
    }
}
```

10 Scrittura di dati primitivi su stream di byte

La classe `PrintStream`, che estende `OutputStream`, permette di convertire dati primitivi in sequenze di byte,[5] che poi vengono scritti su un altro `OutputStream`, passato come argomento al costruttore (uno dei tanti disponibili):

```
public PrintStream(OutputStream out);
```

Esso mette a disposizione i metodi

```
void print(boolean b);
void print(int i);
void print(long l);
void print(float f);
void print(double d);
void print(char c);
void print(char[] s);
void print(String s);
void print(Object obj);
```

e, per ciascuno di essi, esiste anche il corrispondente `println`.

10.1 Esempio

```
import java.io.*;

public class PrintStreamExample {
    public static void main(String[] args) throws IOException {
        FileOutputStream fos = new FileOutputStream("file.txt");
        PrintStream ps = new PrintStream(fos);
        ps.println("Provo valori di vario tipo");
        ps.println(100);
        ps.println(3 / 4.0);
        ps.println(true && false);
        ps.close();
        fos.close();
    }
}
```

[5] In particolare, viene prima costruita una rappresentazione testuale dei dati, e poi i caratteri che la compongono vengono convertiti in byte. Quindi, ad esempio, per un numero intero non viene scritto direttamente il suo valore binario, bensì i byte corrispondenti ai caratteri delle sue cifre decimali.

11 Path dei file

Quando si crea uno stream riferito a un file, la stringa "nome" passata al costruttore può in realtà contenere anche l'intero *path* (percorso) del file: ciò permette l'accesso a file situati in directory diverse da quella corrente.

Inoltre, `java.io` mette a disposizione la classe `File`, le cui istanze rappresentano path di file/directory, in modo astratto e indipendente dal sistema.

12 Flussi di I/O standard

La classe `java.lang.System` ha alcuni campi statici, contenenti degli stream che corrispondono ai flussi di:

- *standard input* (tipicamente collegato alla tastiera)

  ```
  public static final InputStream in;
  ```

- *standard output* e *standard error* (solitamente corrispondenti allo schermo)

  ```
  public static final PrintStream out;
  public static final PrintStream err;
  ```