Input/output

Sezioni

Questa sezione contiene alcuni suggerimenti su come effettuare l'input/output (in senso lato) usando le API di Java (con esempi di codice sorgente che possono essere liberamente copiati e adattati nella soluzione della prova pratica).

Argomenti sulla linea di comando

Per *argomenti sulla linea di comando* si intendono tutte le parole (stringhe massimali non contenenti spazio) che seguono il nome della classe nell'invocazione della JVM. Ad esempio, se avete compilato una classe di nome `Soluzione` e ne invocate l'esecuzione tramite l'interprete come

```
java Soluzione uno          2 tr_e
```

gli argomenti saranno le tre parole: `uno`, `2` e `tr_e`.

La funzione `main` che ha segnatura

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

può accedere a tali parole tramite l'array `args` il cui i-esimo puntatore punta alla stringa corrispondente all'i-esimo argomento (l'argomento di posto 0 è la prima parola).

Osservate che gli argomenti sono *stringhe*, qualora sia richiesto trattare alcuni di essi come numeri sarà necessario usare una funzione di conversione, come ad esempio `parseT` delle varie sottoclassi di **`Number`** (dove `T` è uno dei tipi primitivi), come ad esempio con il metodo **`parseInt`** di **`Integer`**.

Si riporta, a titolo di esempio, un programma che, dati per argomenti alcuni numeri interi, ne stampa la somma

```
public class SommaArgs {
  public static void main(String[] args) {
    int somma = 0;
    for (String arg : args)
      somma += Integer.parseInt(arg);
    System.out.println(somma);
  }
}
```

che, invocato ad come `java SommaArgs 1 2 3`, produce l'output

```
6
```

Input/Output

Di seguito sono riportati alcuni scampoli di codice Java necessari a gestire l'input in formato testuale che tipicamente è richiesto dalla soluzione degli esercizi di laboratorio e d'esame.

La gestione di tale input può essere organizzata secondo due coppie indipendenti di varianti a seconda

1. di come viene consumato
    1. come sequenza di linee,
    2. *tokenizzato* come sequenza di tipi primitivi (`int`, `float`, ...) e *stringhe*;
2. che provenga
    1. dal flusso standard (`System.in`),
    2. da un file (indicato tramite il suo *path*).

Facendo uso dell'istruzione try-with-resources (che consente, tra l'altro, di gestire in modo automatico il rilascio delle risorse in caso di errore) il codice ha in generale la seguente struttura

```
α try (... in = new ...(...)) {
β  while (/* c'è input */)
β    /* consuma l'input */
α }
```

ed è organizzato in due parti

- istanziazione (e gestione) di un oggetto che rappresenti l'input (α),
- ciclo che consuma (ed elabora) l'input (β).

Secondo l'organizzazione logica discussa all'inizio, il modo in cui sarà consumato (1.) e l'origine dell'input (2.) daranno luogo a quattro diverse implementazioni della parte (α), mentre la modalità in cui l'input sarà consumato (1.) darà luogo a due diverse implementazioni della parte (β).

Parte (α): istanziare l'oggetto usato per l'input

Per leggere l'input una linea dopo l'altra (1.1.) è sufficiente usare un `BufferedReader`. Il costruttore di tale classe ha per parametro un `Reader`, che può essere istanziato (2.1.) come un `InputStreamReader` che a sua volta avvolga `System.in`, o (2.2.) come un `FileReader`.

D'altro canto, per *tokenizzare* l'input (1.2.) è sufficiente usare uno `Scanner`. Il costruttore di tale classe ha per parametro un `InputStream`, che può essere direttamente (2.1.) un `System.in`, o istanziato (2.2.) come `FileInputStream`.

Le quattro versioni della parte α del codice sono pertanto:

```
(1.1., 2.1.)  BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
(1.1., 2.2.)  BufferedReader in = new BufferedReader(new FileReader(path));
(1.2., 2.1.)  Scanner in = new Scanner(System.in);
(1.2., 2.2.)  Scanner in = new Scanner(new FileInputStream(path));
```

dove si assume che `path` sia una variabile di tipo stinga che contiene il *path* del file che contiene l'input.

Parte (β): consumare l'input

Per consumare (ed elaborare) l'input, sono sufficienti due solite implementazioni della parte (β), dal momento che il tipo dell'oggetto `in` può essere solo un `BufferedReader` o uno `Scanner`, a seconda di (1.), ma indipendentemente da (2.).

Per leggere una sequenza di linee (1.1.) si può utilizzare il metodo `readLine`; per di più, tale metodo è in grado di segnalare la fine dell'input restituendo il valore speciale `null`. Il ciclo che consuma l'input, in questo caso, è

```
String linea = null;
while ((linea = in.readLine()) != null)
  /* consuma l'input */
```

Tipi primitivi

Per leggere una sequenza di tipi primitivi (1.2.) si possono utilizzare i metodi `nextT` (dove `T` è uno dei tipi primitivi), ad esempio, per gli

interi, si può usare il metodo `nextInt`; per sapere se l'input è finito (o se ci sono ancora a disposizione altri elementi), si può usare il metodo `hasNextT` (dove `T` è, come sopra, uno dei tipi primitivi), ad esempio, ancora nel caso degli interi `hasNextInt`. Il ciclo che consuma l'input, sempre nel caso degli interi, è

```
while (in.hasNextInt()) {
  int intero = in.nextInt();
  /* consuma l'input */
}
```

Stringhe

Qualora sia necessario leggere delle stringhe (1.2.), intese come delle sequenze massimali di caratteri diversi da *whitespace* (che sono spazio, segno di tabulazione orizzontale e verticale e a-capo), si possono usare i metodi `next` e `hasNext` in modo del tutto analogo al caso precedente

```
while (in.hasNext()) {
  String stringa = in.next();
  /* consuma l'input */
}
```

Osservazioni ed esempi

Mettendo assieme gli esempi di codice delle parti (α) e (β) è possibile elaborare l'input, come sequenza di linee o tipi primitivi, sia che provenga dal flusso standard che da un file.

Un dettaglio utile da ricordare è che nella lettura del flusso standard da console (senza redirezione, cioè), la **terminazione del flusso** va *segnalata esplicitamente* tramite l'immissione dell'apposito carattere di controllo `^D` denominato `EOF` (**end of file**), che si ottiene usualmente premendo assieme i tasti `ctrl` e `d` (minuscolo).

Altro dettaglio importante è che *alcuni dei costruttori e metodi invocati possono sollevare eccezioni* di tipo `IOException` (o sue sottoclassi), che devono essere *opportunamente gestite* (sia che il codice sia avvolto dalla `try-with-resources` o meno). Nel contesto della prova d'esame, qualora tali metodi fossero invocati all'interno del metodo `main`, una soluzione plausibile è quella di aggiungere `throws IOException` alla dichiarazione di tale metodo (come nel codice riportato di seguito).

A titolo di esempio, riportiamo due piccoli programmi. Il primo legge l'input dal flusso standard ed emette ogni riga preceduta dal suo numero progressivo

```
public class NumeraLinee {
  public static void main(String[] args) throws IOException {
    try (BufferedReader in = new BufferedReader(new InputStreamReader(System.in))) {
      int n = 0;
      String linea;
      while ((linea = in.readLine()) != null)
        System.out.println(String.format("%02d: %s", ++n, linea));
    }
  }
}
```

Per provare il suo funzionamento è possibile usare il comando `java java NumeraLinee < NumeraLinee.java` che, facendo uso della *redirezione* dell'input, numererà le linee del programma stesso producendo l'output

```
01: public class NumeraLinee {
02:   public static void main(String[] args) throws IOException {
03:     try (BufferedReader in = new BufferedReader(new InputStreamReader(System.in))) {
04:       int n = 0;
05:       String linea;
06:       while ((linea = in.readLine()) != null)
07:         System.out.println(String.format("%02d: %s", ++n, linea)));
08:     }
09:   }
10: }
```

Il secondo legge una sequenza di numeri in virgola mobile da un file il cui *path* è specificato come parametro (all'invocazione della JVM), e ne stampa la somma

```
public class SommaInput {
  public static void main(String[] args) throws IOException {
    String path = args[0];
    float somma = 0.0f;
    try (Scanner in = new Scanner(new FileInputStream(path))) {
      while (in.hasNextFloat()) {
        float numero = in.nextFloat();
        somma += numero;
      }
    }
    System.out.println(somma);
  }
}
```

Assumendo che esista un file `input.txt` che contenga:

```
1
2.5
3
```

eseguendo il programma con il comando `java SommaInput input.txt`, verrà prodotto l'output

```
6.5
```

Per concludere osservate che la classe `Scanner` ha un costruttore che accetta una stringa come argomento (e quindi attingerà da tale stringa per rispondere alle varie chiamate di `nextT` e `hasNextT`); considerate ad esempio l'esecuzione di

```
try (Scanner linea = new Scanner("somma 1 e 3.2")) {
  System.out.println(
    "Prima parola: " + linea.next() + ",\n" +
    "doppio del primo intero: " + 2 * linea.nextInt() + ",\n" +
    "seconda parola: " + linea.next() + ",\n" +
    "metà dell'ultimo float: " + linea.nextFloat() / 2
  );
}
```

```
Prima parola: somma,
doppio del primo intero: 2,
seconda parola: e,
metà dell'ultimo float: 1.6
```

che mostra che chiamate consecutive dei metodi `next` consentono di "decodificare" le parti della stringa `somma 1 e 3.2` a seconda del loro tipo (primitivo o stringa).

> **Suggerimento**
>
> Grazie al fatto che può essere costruita a partire da una stringa, la classe `Scanner` può essere utilizzata per realizzare una sorta di "parser" in grado di decodificare un input costituito da linee ciascuna delle quali sia a sua volta costituita da parti (separate da spazi) corrispondenti a tipi primitivi (o stringhe) secondo un assegnato "formato"; ad esempio
>
> ```
> try (Scanner in = new Scanner(System.in)) {
>   while (in.hasNextLine())
>     try (Scanner linea = new Scanner(in.nextLine())) {
>       /* consuma le parti della linea */
>     }
> }
> ```
>
> leggerà il flusso standard una linea alla volta dallo scanner `in` e per ciascuna di esse costruirà lo scanner `linea` che potrà essere usato come nell'esempio precedente per "decodificare" le parti della linea che corrispondono ai vari tipi primitivi (e stringhe) indicati dal "formato" specificato.

Altri approcci

La ricchezza delle API di Java rende possibile risolvere il problema descritto in questa guida in molti altri modi. Questo è certamente una ricchezza, ma produce anche molta confusione in chi si avvicina per la prima volta al linguaggio e alle sue librerie.

Ad esempio, l'input di tipi primitivi potrebbe anche essere implementato leggendo l'input per linea, suddividendo poi la linea con uno `StringTokenizer`, o con il metodo `split` di `String`, traducendo in fine le singole parti nei tipi primitivi con i metodi `parseT` delle varie sottoclassi `Number` (dove `T` è uno dei tipi primitivi), come ad esempio con il metodo `parseInt` di `Integer`. Evidentemente, l'uso della classe `Scanner` appare una soluzione molto più elementare a questo tipo di problema. In ogni modo, una soluzione alternativa, in questo senso, dell'esercizio due potrebbe essere la seguente:

```
public class SommaInputBis {
  public static void main(String[] args) throws IOException {
    String path = args[0];
    float somma = 0.0f;
    try (BufferedReader in = new BufferedReader(new FileReader(path))) {
      String linea = null;
      while ((linea = in.readLine()) != null) {
          float numero = Float.parseFloat(linea);
          somma += numero;
      }
    }
    System.out.println(somma);
  }
}
```

D'altro canto, a ben guardare, c'è un metodo `nextLine` tra quelli di `Scanner` che si comporta sostanzialmente come il metodo `readLine` di `BufferedReader`; in linea di principio, quindi, tutta la discussione si potrebbe di gran lunga semplificare limitandosi ad utilizzare la classe `Scanner` sia per leggere l'input linea per linea che in modo *tokenizzato*. Ma è altresì vero che l'uso di una classe complessa come `Scanner` per uno scopo così banale come quello di leggere l'input per linee sembra del tutto sproporzionato; inoltre, tale classe ha fatto la sua comparsa solo nelle versioni più recenti di Java, ragion per cui è bene conoscere anche alternative che siano praticabili nel caso in cui si abbia a disposizione sono una versione meno recente del linguaggio.

Dati non testuali

Come ultima osservazione, si noti che in questa guida (per brevità e semplicità) si è trattato solo il caso di file in formato, per così dire, testuale. Le API di Java mettono a disposizione anche classi e metodi per il trattamento di dati in formato binario (ad esempio, tramite le interfacce `DataInput` e `DataOutput` e relative implementazioni), che meritano una discussione a se stante.

Una interessante aggiunta nelle API delle nuove versioni di Java è la classe `Files` che mette a disposizione una serie di metodi statici