Dal linguaggio C al linguaggio Java

(Seconda parte)

Riccardo Silvestri

15-3-2009 2-3-2010

Sommario della seconda parte

Tipi e oggetti speciali
- **Array** Array multidimensionali Inizializzatori di array Argomenti dalla linea di comando Numero variabile di parametri
  - **Esercizi** Err_array Err_array? Duplicati Ordina_caratteri Non_solo_vocali N_parole_verticali Grafico_a_barre Quadrati_magici Numeri_casuali Parole_casuali Matrici Costellazioni_di_caratteri
- **La prima classe - versione 2** Disaccoppiare e astrarre
  - **Esercizi** Cascata Scalinata Tessiture Collisione Piramidi_2
- **Tipi enumerativi**
  - **Esercizi** Date_versione_2 Date_giorni Calendario
- **Errori ed eccezioni** Catturare eccezioni Lanciare eccezioni Eccezioni controllate
  - **Esercizi** Digitando_interi Date_corrette

Tipi per dati persistenti: i file
- **Localizzare file**
  - **Esercizi** Numero_file/dir Conta_file Trova_file Trova_file_ext Profondità Info_file/dir
- **Accesso random**
  - **Esercizi** Modifica Appuntamenti_del_giorno Ordine_cronologico Rubrica Rubrica+ Contabilità Contabilità+ Appuntamenti+rubrica
- **Accesso sequenziale** Lettura Scrittura
  - **Esercizi** Linee Parole Log Log+ Elenchi_di_parole Ortografia Anagrammi Unire_elenchi Elenchi_ad_accesso_random

Tipi e oggetti speciali

Finora abbiamo visto i tipi di base di Java (tipi primitivi, tipi classe e tipi riferimento). Oltre a questi Java offre altri tipi di cui i più importanti sono i tipi array.

Array

Un *array* è un tipo speciale di oggetto che contiene zero o più componenti che sono variabili tutte dello stesso tipo. Il tipo delle componenti può essere un qualsiasi tipo primitivo o un qualsiasi tipo riferimento. Il numero delle componenti di un array (la sua *lunghezza*) è fissato quando l'array è creato e non può essere cambiato. La sintassi per dichiarare una variabile di tipo "array di *T*", dove *T* è un qualsiasi tipo, è la seguente:

```
T[] v;
```

Il tipo della variabile `v` è quindi denotato da `T[]`. Con la precedente dichiarazione, però, si è semplicemente dichiarata una variabile `v` di tipo "array di *T*" ma non è stato creato nessun array. Per creare un array di *n* elementi si usa la sintassi:

```
T[] v = new T[n];
```

dove *n* può essere una costante numerica, una variabile o più in generale una qualsiasi espressione a valore intero. Le componenti di un array sono numerate a partire da 0, come nel C, e la sintassi per accedere ad una componente dell'array è anch'essa uguale a quella del C: la componente di indice `i` dell'array `v` è `v[i]`. Ecco alcuni esempi:

```
int[] interi = new int[10];            // un array di 10 int
String[] stringhe = new String[5];     // un array di 5 riferimenti a oggetti String
Point[] punti = new Point[8];          // un array di 8 riferimenti a oggetti Point
```

Gli array appena creati hanno le componenti inizializzate con i relativi valori di default. Così, le componenti dell'array `interi` hanno valore 0 e le componenti degli array `stringhe` e `punti` hanno valore `null`.

I *tipi array* (*array types*) fanno parte della famiglia dei tipi riferimento ma sono distinti dai tipi classe. Però ogni array è un oggetto. Questo significa, ad esempio, che la variabile sopra definita `punti` ha come valore il riferimento ad un oggetto array (che a sua volta consiste di 8 componenti ognuna delle quali è un riferimento a un oggetto della classe `Point`). La lunghezza di un array è disponibile tramite un campo (costante) di nome `length`. La lunghezza di un array, però, non fa parte del tipo array. Questo ha due importanti conseguenze. La prima è che non è possibile dichiarare un metodo che accetta come argomento un array la cui lunghezza è fissata (nei parametri del metodo). Se un metodo accetta tra i suoi parametri, ad esempio, un array di `int` (dichiarato `int[]`) dovrà accettare array di `int` di qualsiasi lunghezza. Questo non è un problema grazie alla disponibilità del campo `length`. Ecco l'esempio di un metodo che prende in input un array di `Point` e lo stampa:

```
void stampaPunti(Point[] pp) {
    for (int i = 0 ; i < pp.length ; i++)    // pp.length è la lunghezza dell'array pp
        System.out.println("(" + pp[i].x + ", " + pp[i].y + ")");
}
```

La seconda conseguenza riguarda gli array di array (o array multidimensionali).

Array multidimensionali Siccome, come abbiamo già detto, il tipo delle componenti di un array può essere un qualsiasi tipo riferimento e i tipi array sono particolari tipi riferimento, è chiaro che è possibile definire array di array. Ovvero, array il cui tipo delle componenti è a sua volta un tipo array. È quindi possibile, ad esempio, dichiarare le seguenti variabili:

```
float[][] matrice;           // array le cui componenti sono di tipo float[] (array di float)
int[][][] cubo;              // array le cui componenti sono di tipo int[][]
String[][] matriceStringhe;  /* array le cui componenti sono di tipo String[], cioè array di
                                riferimenti a oggetti String */
```

Ovviamente, è anche possibile creare array di array:

```
matrice = new float[5][10];            // una matrice 5x10 di float
cubo = new int[4][4][4];               // un cubo 4x4x4 di int
matriceStringhe = new String[10][8];   // una matrice 10x8 di stringhe
```

Non è necessario specificare tutte le dimensioni, l'unica che è necessario specificare è la prima (da sinistra). Le altre possono essere specificate (e create) dopo. Ad esempio, la dichiarazione `matrice = new float[5][10]` è equivalente a:

```
matrice = new float[5][];       // crea un array di 5 riferimenti ad array di float
for (int i = 0 ; i < matrice.length ; i++)
    matrice[i] = new float[10];  /* crea un array di 10 float e ne assegna il riferimento
                                    alla i-esima componente dell'array matrice */
```

Si osservi però che le dimensioni non specificate possono essere solamente quelle più a destra:

```
cubo = new int[4][][];          // OK
cubo = new int[4][4][];         // OK
cubo = new int[4][][4];         // ERRORE
cubo = new int[][4][4];         // ERRORE
cubo = new int[][][4];          // ERRORE
cubo = new int[][4][];          // ERRORE
```

A questo punto arriviamo alla seconda conseguenza del fatto che la lunghezza non fa parte del tipo array: le lunghezze degli array componenti non devono essere necessariamente uguali. Così, è possibile creare, ad esempio, una matrice triangolare:

```
matrice = new float[5][];       // crea un array di 5 riferimenti ad array di float
```

```
for (int i = 0 ; i < matrice.length ; i++)
    matrice[i] = new float[i + 1];     // crea un array di lunghezza i + 1
```

Si osservi che il tipo della variabile `matrice[i]` è riferimento ad array di `float` (cioè `float[]`) e che, dopo l'esecuzione di questo frammento di codice, risulterà `matrice[0].length = 1`, `matrice[1].length = 2`, . . . `matrice[4].length = 5`. Il seguente esempio mostra un metodo che stampa una matrice le cui dimensioni possono essere qualsiasi (qualsiasi numero di righe e ogni riga di lunghezza qualsiasi):

```
void stampaMatrice(float[][] matrice) {
    for (int r = 0 ; r < matrice.length ; r++) {
        for (int c = 0 ; c < matrice[r].length ; c++)
            System.out.printf("%8.2f ", matrice[r][c]);
        System.out.println();
    }
}
```

Ebbene sì, gli oggetti di tipo `PrintStream`, come lo è `System.out`, supportano anche un metodo chiamato `printf()` che è del tutto simile alla omonima funzione per la stampa formattata della libreria standard del C.

Inizializzatori di array Come nel C anche in Java è possibile creare un array e inizializzarne le componenti in un unica espressione. Inoltre, anche la sintassi è essenzialmente quella del C. Quindi possiamo spiegare gli inizializzatori tramite esempi:

```
int[] primi = {2, 3, 5, 7, 11};    // crea e inizializza un array di 5 int
String[] risposte = {"SI", "NO"};  // crea e inizializza un array di 2 stringhe
double x, y, x2, y2;
. . .
Point[] punti = {new Point(x, y), new Point(x2, y2)};  /* crea e inizializza un
                                                          array di 2 Point */
```

Come si vede dall'ultimo esempio le espressioni negli inizializzatori non devono essere necessariamente delle costanti ma possono essere espressioni qualsiasi perché l'inizializzazione viene effettuata durante l'esecuzione del programma. Anche per gli array multidimensionali si possono usare gli inizializzatori:

```
int[][] quadratoLatino = {{1, 2, 3},
                          {2, 3, 1},
                          {3, 1, 2}};   // crea e inizializza una matrice 3x3 di int
String[][] risposte = {{"SI", "NO"},
                       {"NO", "SI"}};   // crea e inizializza una matrice 2x2 di stringhe
int[][] triangolare = {{1},
                       {1, 2},
                       {1, 2, 3},
                       {1, 2, 3, 4}};   // crea e inizializza una matrice triangolare di int
```

In sostanza, si può dire che gli array e gli array multidimensionali sono trattati, a parte piccole differenze sintattiche, in modo del tutto simile a come nel C sono trattati gli array e gli array multidimensionali dinamicamente allocati.

Argomenti dalla linea di comando Adesso che conosciamo gli array sappiamo cosa significa il parametro di input del metodo `main`. È un array di stringhe (`String[]`). Ognuna delle stringhe dell'array contiene un argomento della linea di comando. Il seguente esempio mostra un programma che interpreta gli argomenti passati nella linea di comando come degli interi e ne stampa in output la somma:

```
public class Add {
    public static void main(String[] args) {
        int sum = 0;
        for (int i = 0 ; i < args.length ; i++)
            sum += Integer.parseInt(args[i]);
        System.out.println("La somma è "+sum);
    }
}
```

Per convertire le stringhe in numeri interi si è usato il metodo (statico) `int parseInt(String s)` della classe `Integer` (apartenente al package `java.lang`). Se, ad esempio, il programma è eseguito con

```
java Add 23 -12 56
```

allora l'array `args` avrà tre componenti

```
args[0]: "23"    args[1]: "-12"    args[2]: "56"
```

e il programma stamperà

```
La somma è 67
```

Numero variabile di parametri In Java è estremamente semplice definire metodi che possono essere invocati con un numero variabile di parametri. Ad esempio, un metodo per calcolare il massimo di un numero variabile di interi può essere così definito:

```
int max(int first, int...rest) {
    int max = first;
    for (int i = 0 ; i < rest.length ; i++)
        if (rest[i] > max) max = rest[i];
    return max;
}
```

Le seguenti sono delle possibili invocazioni per il metodo `max()`:

```
max(2)
max(3, 9, 6)
max(2, 8, 1, 10)
```

La sintassi `int...rest` sta a significare zero o più parametri di tipo `int`. Dal punto di vista dell'implementazione, è come se il metodo fosse stato dichiarato `int max(int first, int[] rest)`. In effetti, anche se fosse stato definito così, sarebbe stato comunque possibile invocarlo con un numero variabile di parametri. Però si preferisce usare la sintassi dei "tre punti" nei casi in cui si vuole evidenziare la variabilità del numero di parametri.

Ovviamente, non c'è nulla di speciale nel tipo `int`, è possibile definire metodi che prendono un numero variabile di parametri di tipo qualsiasi. Purché quelli variabili sono tutti dello stesso tipo e sono quelli più a destra. Più avanti vedremo come è possibile definire metodi che accettano un numero variabile di parametri di tipi non necessariamente uguali (similmente alla funzione `printf()` del C).

Esercizi

[Err_array] Trovare e spiegare i due errori contenuti nel seguente programma:

```
public class Test {
    public static void main(String[] args) {
        String[][] mstr = new String[10][];
        for (int i = 0 ; i < mstr.length ; i++)
            mstr[i][0] = "A";
        mstr[0] = new String[5];
        int len = mstr[0][0].length();
    }
}
```

[Err_array?] Il seguente programma contiene errori?

```
public class Test {
    public static void main(String[] args) {
        String[][] m = new String[4][4];
        m[0][0] = "A";
        m[1] = new String[10];
        m[1][7] = "B";
    }
}
```

[Duplicati] Scrivere un programma che legge un intero *n* e poi legge una sequenza di *n* interi e se la sequenza contiene dei valori ripetuti stampa tutti i valori che si ripetono almeno due volte, altrimenti stampa `"non ci sono duplicati"`.

[Ordina_caratteri] Scrivere un programma che legge una stringa e stampa la sequenza ordinata dei caratteri della stringa. Ad esempio, se la stringa è `"io programmo in Java"` allora il programma stampa

```
   Jaaagiimmnoooprrv
```

Modificare il programma in modo tale che possa leggere la stringa direttamente dalla linea di comando.

[Non_solo_vocali] Scrivere un programma che legge una linea di testo e per ogni carattere alfabetico, maiuscolo o minuscolo, stampa il numero di volte che appare nella linea di testo. Ad esempio, se la linea di testo è `"Contare Caratteri Alfabetici"` allora il programma stampa:

```
a: 4        A: 1
b: 1        B: 0
c: 1        C: 2
d: 0        D: 0
.           .
.           .
.           .
z: 0        Z: 0
```

Suggerimento: in Java (come in C) facendo il cast ad `int` di `char` che rappresentano caratteri alfabetici minuscoli (o maiuscoli) si ottengono interi consecutivi.

[N_parole_verticali] Scrivere un programma che legge un intero positivo *n*, poi legge *n* parole e infine stampa le *n* parole in verticale come nell'esercizio [Parole_verticali]. Modificare il programma in modo che possa leggere le parole direttamente dalla linea di comando.

[Grafico_a_barre] Scrivere un programma che legge un intero *n* e poi legge una sequenza di *n* interi positivi e stampa un grafico a barre della sequenza. Ad esempio, se la sequenza è `2, 5, 1, 3, 7, 5, 4` allora il programma stampa:

```
      *
      *
  *   **
  *   ***
  *  ****
 **  ****
 *******
```

Suggerimento: usare una matrice di caratteri di dimensione opportuna.

[Quadrati_magici] Un *quadrato magico* è una disposizione di numeri interi distinti in una tabella quadrata tale che la somma dei numeri presenti in ogni riga, in ogni colonna e in entrambe le diagonali dia sempre lo stesso numero. Ad esempio, il seguente è un quadrato magico di ordine 3:

8 1 6

3 5 7

4 9 2

Scrivere un programma che letto un intero dispari *n* stampa un quadrato magico di ordine *n*. I quadrati magici di ordine dispari possono essere costruiti tramite il semplice algoritmo descritto in http://it.wikipedia.org/wiki/Quadrato_magico.

[Numeri_casuali] Scrivere un programma che legge due interi `n` e `m`, poi genera `m` interi "casuali" nell'intervallo `[1, n]` (usando il metodo `Math.random()`) e infine stampa le frequenze e gli scarti percentuali rispetto alla frequenza media. Ad esempio, se `n = 12` e `m = 10000` il programma potrebbe stampare:

```
Frequenza media: 833.33%
Valore    Frequenza       Scarto percentuale
1         829             0.005%
2         813             0.024%
3         859             0.031%
4         802             0.038%
5         806             0.033%
6         843             0.012%
7         846             0.015%
8         824             0.011%
9         824             0.011%
10        842             0.010%
11        869             0.043%
12        843             0.012%
```

[Parole_casuali] Scrivere un programma che legge un intero `n` e stampa `n` parole "casuali". Le parole "casuali" possono essere generate servendosi dei seguenti tre array di stringhe:

```
String[] voc = {"a","e","i","o","u"};
String[] cons1 = {"b","c","d","f","g","l","m","n","p","q","r","s","t","v","z",
                  "br","cl","cr","dr","fl","fr","gl","gn","gr","pl","pn","pr","ps",
```

```
                  "sb","sc","sd","sf","sg","sm","sn","sp","sr","st","sv","tr"};
String[] cons2 = {"bb","cc","dd","ff","gg","ll","mm","nn","pp","rr","ss","tt","vv","zz",
                  "lb","lc","ld","lm","ln","lp","ls","lt","lv","lz","mb","mp","nc","nd",
                  "ng","ns","nt","nv","rb","rc","rd","rp","rs","rt","rt"};
```

Per ogni parola si sceglie un intero "casuale" `m` compreso tra 3 e 9, poi se `m` è dispari si inizia la parola con un elemento "casuale" di `voc` altrimenti con un elemento "casuale" di `cons1`. Poi si alternano scelte "casuali" da `voc` e da `cons1` o `cons2`, fino a `m` volte. Ad esempio, se `n = 20` il programma potrebbe stampare:

```
invicca
qisersenca
liqangoze
ubribbarriqe
uppumustoca
icca
esfezzofrazzo
scacrilpu
villa
nefupe
idru
urtarteresvo
plentigruvva
altizzansoppo
lamma
sbotri
stanni
ancaru
alleredessu
svasvo
```

Il programma può essere raffinato scegliendo in modo più accurato gli elementi degli array `voc`, `cons1` e `cons2` e la probabilità di scegliere gli elementi di un array potrebbe essere diversa da elemento a elemento (ad esempio, la probabilità di scegliere una `"a"` dovrebbe essere più alta della probabilità di scegliere una `"u"`). Inoltre, si potrebbero introdurre altri array e delle opportune regole di combinazione (non solo la semplice alternanza).

[Matrici] Definire una classe per rappresentare matrici quadrate i cui elementi sono numeri in virgola mobile. Prevedere un costruttore che prende come argomento la dimensione `n` della matrice e costruisce una matrice `nxn` con tutti gli elementi inizializzati a `0.0`. Prevedere anche metodi per scrivere e leggere i singoli elementi della matrice. Aggiungere un metodo per stampare la matrice. Considerare anche un metodo per moltiplicare due matrici (ovviamente solo se hanno la stessa dimensione).

[Costellazioni_di_caratteri] Definire una classe `CharPoint` che rappresenta un carattere e una posizione. La signature del costruttore dovrebbe essere `CharPoint(char c, int x, int y)` (il sistema di riferimento è lo stesso di quello di `CharRect`). Usare tale classe per definire una classe `CharPointSet` per rappresentare insiemi di `CharPoint`. La classe ha un costruttore che costruisce un insieme vuoto ed ha un metodo `add(CharPoint p)` che aggiunge un `CharPoint p` all'insieme. Inoltre, la classe ha un metodo `draw()` che stampa l'insieme dei `CharPoint`. Ad esempio, se tutti i `CharPoint` dell'insieme hanno il carattere uguale a `'*'` e le loro coordinate sono (0, 40), (3, 8), (3, 24), (4, 16), (6, 0), (7, 21), (6, 40), allora il metodo `draw()` stampa:

```
                                        *


        *               *
                *


*                                       *
                            *
```

Suggerimento: usare un array di `CharPoint` per mantenere l'insieme e ogniqualvolta viene aggiunto un `CharPoint` ricreare un nuovo array. Nell'implementazione del metodo `draw()` fare attenzione ai punti sulla stessa riga.

La prima classe - versione 2

Un difetto della nostra classe `CharRect` è che non permette di stampare in modo appropriato due o più rettangoli che, relativamente ad un sistema di riferimento condiviso, si estendono su una o più righe comuni. Consideriamo, ad esempio, i due rettangoli di coordinate `(0, 0, 8, 5)` e `(10, 2, 10, 4)`, dove le coordinate significano `(left, top, width, height)`. Al momento li possiamo stampare così

(l'uno dopo l'altro):

```
********
********
********
********
********


          **********
          **********
          **********
          **********
```

E quindi ognuno è stampato relativamente al proprio sistema di riferimento che fa ripartire sempre da 0 la numerazione delle righe. Invece, vorremmo stamparli relativamente ad un unico sistema di riferimento condiviso. Se stampati relativamente a un sistema condiviso, il risultato sarebbe questo:

```
********
********
********  **********
********  **********
********  **********
          **********
```

Come possiamo fare? La soluzione più semplice è di aggiungere un metodo alla classe `CharRect` che "stampa" il rettangolo su una matrice `s` di caratteri. Se si vuole quindi stampare un insieme di rettangoli basterà "stamparli" tutti su `s` e infine stampare la matrice `s` sullo schermo della console. In questo modo `s` rappresenta proprio il sistema di riferimento condiviso. E la ragione che impedisce di fare ciò direttamente sullo schermo è che il flusso di output standard (lo schermo della console) non può essere riscritto, non si può tornare indietro su posizioni già attraversate. Quindi basterebbe aggiungere alla classe un metodo come quello qui sotto riportato:

```
public void draw(char[][] screen) {
    for (int r = 0 ; r < height ; r++)
        for (int c = 0 ; c < width ; c++)
            screen[top + r][left + c] = fillChar;
}
```

Però se vogliamo mantenere ancora le funzionalità della classe di poter stampare direttamente sul flusso di output, dovremmo duplicare ogni metodo di stampa per stampare sia su una matrice di caratteri che direttamente sul flusso di output. Questa situazione si aggrava se consideriamo che potremmo voler aggiungere altri metodi che stampano il rettangolo in altre tessiture (a strisce verticali o orizzontali, a scacchi, ecc.). Inoltre, e se volessimo stampare su file? Dovremmo triplicare ogni metodo di stampa.

Disaccoppiare e astrarre Un modo per risolvere questo problema consiste nel *disaccoppiare* gli algoritmi che producono la tessitura del rettangolo dallo specifico mezzo sul quale deve avvenire la stampa. Per fare ciò è però necessario (e anche sufficiente) *astrarre* una interfaccia per i vari mezzi di stampa che possa essere usata dagli algoritmi che producono la tessitura. Questa interfaccia non è difficile da individuare, è sufficiente un metodo che permetta di stampare un carattere in una determinata posizione, come il seguente:

```
void printChar(int row, int col, char c)
```

È chiaro che un qualsiasi algoritmo che produce una tessitura può usare questo metodo per stamparla. Inoltre, l'implementazione del metodo `printChar()` sarà "nascosta" in un oggetto che rappresenterà lo specifico mezzo di stampa. Quindi quello che ci occorre è, prima di tutto, una classe i cui oggetti rappresentano i possibili mezzi di stampa (flusso di output, matrice di caratteri, ecc.) e che implementano il metodo `printChar()`. Chiamiamo tale classe `PrintMedium`. Una possibile definizione è la seguente:

```
import java.io.*;        // serve per la classe PrintStream

public class PrintMedium {
    private char[][] screen = null;
    private PrintStream stream = null;
    private int currRow = 0, currCol = 0;
        // costruttore per matrice di caratteri
    public PrintMedium(char[][] screen) { this.screen = screen; }
        // costruttore per flusso di output
```

```
    public PrintMedium(PrintStream stream) { this.stream = stream; }

    public void printChar(int row, int col, char c) {
        if (screen != null) {
            screen[row][col] = c;
        } else if (stream != null) {
            if (currRow < row) currCol = 0;
            for ( ; currRow < row ; currRow++) stream.println();
            for ( ; currCol < col ; currCol++) stream.print(' ');
            stream.print(c);
            currCol++;
        }
    }
                                // questo metodo serve a terminare la stampa
    public void end() {         // quando il mezzo è un flusso
        if (stream != null) {
            stream.println();
            currRow = currCol = 0;
        }
    }
}
```

Per i nostri attuali scopi, questa implementazione è sufficiente. Però, in generale, non è molto soddisfacente. Prima di tutto l'implementazione di `printChar()` per i flussi di output assume che i caratteri siano stampati in ordine, cioè dall'alto verso il basso e da sinistra verso destra. Se questa assunzione non è rispettata il risultato non sarà quello che ci si aspetta. Per rimediare si dovrebbe usare un buffer. L'altro motivo di insoddisfazione deriva da come sono rappresentati i diversi mezzi di stampa nella classe. L'oggetto di tipo `PrintMedium` distingue qual'è il mezzo dal valore dei campi `screen` e `stream` e se è matrice di caratteri contiene comunque i campi `currRow` e `currCol` che hanno senso solo per un flusso. Tutto ciò sarebbe ulteriormente peggiorato se la classe permettesse di rappresentare anche altri mezzi di stampa. I linguaggi orientati agli oggetti come Java offrono dei meccanismi (ereditarietà) che permettono di definire classi come `PrintMedium` in modo molto più elegante e soddisfacente. Finché non introdurremo tali meccanismi dovremo accontentarci dell'attuale implementazione.

Vediamo ora come modificare la classe `CharRect` affinché possa usare le funzionalità offerte dalla classe `PrintMedium`. Ogni oggetto di tipo `CharRect` avrà ora un nuovo campo `pMedium` che mantiene il mezzo su cui stampare. Siccome avrebbe poco senso avere un oggetto rettangolo che non avesse specificato un mezzo di stampa, il costruttore impone che questo sia specificato al momento della creazione. Però può essere liberamente modificato dopo la creazione grazie al metodo `setPM()`.

```
public class CharRect {
    private static final char DEF_FILLCHAR = '*';
    private static final char DEF_FILLCHAR2 = 'o';

    private int left, top;
    private int width, height;
    private char fillChar = DEF_FILLCHAR, fillChar2 = DEF_FILLCHAR2;
    private PrintMedium pMedium;    // nuovo campo per mantenere il mezzo di stampa

    public CharRect(PrintMedium pm, int l, int t, int w, int h) {
        left = l;
        top = t;
        width = w;
        height = h;
        pMedium = pm;
    }

    public void setChar(char c) { fillChar = c; }
    public void setChar(char c, char c2) { fillChar = c; fillChar2 = c2; }
    public void setPM(PrintMedium pm) { pMedium = pm; }

    public void draw() {
        for (int r = 0 ; r < height ; r++)
            drawLine(r, fillChar, fillChar);
        pMedium.end();
    }

    public void drawVStripes() {
        for (int r = 0 ; r < height ; r++)
            drawLine(r, fillChar, fillChar2);
        pMedium.end();
    }
```

```
    private void drawLine(int r, char ch1, char ch2) {
        for (int k = 0 ; k < width ; k++) {
            char ch = (k % 2 == 0 ? ch1 : ch2);
            pMedium.printChar(top + r, left + k, ch);
        }
    }
}
```

Si noti come la nuova implementazione dei metodi di stampa (`draw()`, `drawVStripes()`, `drawLine()`) sia più semplice e diretta di quella originale. Questo non è un caso. Spesso quando si trovano le giuste astrazioni (nel nostro caso l'interfaccia di `PrintMedium`) si ottiene come vantaggio addizionale la semplificazione delle implementazioni. Viceversa se questo non accade ciò può essere una indicazione che l'astrazione introdotta non è quella giusta.

Ed ora consideriamo un programma che mette alla prova la nuova versione della classe `CharRect`:

```
public class Test {
    private static void clear(char[][] screen) {   // metodo per "ripulire" una
        for (int r = 0 ; r < screen.length ; r++)  // matrice di caratteri
            for (int c = 0 ; c < screen[r].length ; c++)
                screen[r][c] = ' ';
    }
    private static void print(char[][] screen) {     // metodo per stampare una
        for (int r = 0 ; r < screen.length ; r++) {  // matrice di caratteri
            for (int c = 0 ; c < screen[r].length ; c++)
                System.out.print(screen[r][c]);
            System.out.println();
        }
    }
    public static void main(String[] args) {

        char[][] screen = new char[10][50];  // crea una matrice di caratteri
        clear(screen);                       // "pulisce" la matrice
            // crea l'oggetto che rappresenta il mezzo di stampa relativo alla
        PrintMedium pm = new PrintMedium(screen);    // matrice di caratteri
        CharRect rectA = new CharRect(pm, 3, 0, 10, 7);  // crea tre rettangoli
        CharRect rectB = new CharRect(pm, 15, 1, 12, 3); // che "stampano" sulla
        CharRect rectC = new CharRect(pm, 22, 4, 4, 4);  // matrice di caratteri
        rectA.draw();                // "stampa" i tre rettangoli
        rectB.drawVStripes();
        rectC.draw();
        print(screen);               // stampa la matrice di caratteri
        clear(screen);               // "ripulisce" la matrice
        rectA.drawVStripes();            // "ristampa" i rettangoli con
        rectB.setChar('#', '!');         // altre tessiture
        rectB.drawVStripes();
        print(screen);               // stampa la matrice di caratteri
            // crea un oggetto che rappresenta il mezzo di stampa relativo al
        PrintMedium out = new PrintMedium(System.out);    // flusso di output
        rectA.setPM(out);            // modifica il mezzo di stampa
        rectB.setPM(out);            // dei tre rettangoli
        rectC.setPM(out);
        rectA.draw();                // stampa i rettangoli sul
        rectB.drawVStripes();        // flusso di output
        rectC.draw();
    }
}
```

Il risultato dell'esecuzione è il seguente:

```
**********
**********  *o*o*o*o*o*o*o
**********  *o*o*o*o*o*o*o
**********  *o*o*o*o*o*o*o
**********         ****
**********         ****
**********         ****
                   ****


*o*o*o*o*o*o
*o*o*o*o*o*o  #!#!#!#!#!#!
*o*o*o*o*o*o  #!#!#!#!#!#!
```

```
   *o*o*o*o*o*  #!#!#!#!#!#!
   *o*o*o*o*o*o
   *o*o*o*o*o*o
   *o*o*o*o*o*o



   **********
   **********
   **********
   **********
   **********
   **********
   **********

               #!#!#!#!#!#!
               #!#!#!#!#!#!
               #!#!#!#!#!#!



                      ****
                      ****
                      ****
                      ****
```

Riassumendo abbiamo visto che operando un disaccoppiamento (tra ciò che deve essere stampato e il mezzo di stampa) e introducendo una opportuna astrazione (l'interfaccia del mezzo di stampa) che permette di incapsulare una parte del sistema in una classe separata (la classe `PrintMedium`), la struttura del sistema diventa più chiara (la netta separazione tra le tessiture e i dettagli del mezzo di stampa), alcune implementazioni si semplificano (i metodi di stampa) e il sistema diventa molto più facile da estendere (aggiungere altri mezzi di stampa o altre tessiture).

Tutto ciò, come avremo modo di vedere più avanti, non è qualcosa che vale solamente per questo caso specifico ma ha una valenza molto più generale. A ben vedere non abbiamo fatto altro che operare una decomposizione del sistema in parti più semplici individuando le parti che avevano un alto grado di mutua indipendenza. E abbiamo incapsulato le parti nelle rispettive classi applicando così il principio dell'information hiding. A questo punto si potrebbe essere indotti a credere che è sufficiente invocare questi due principi generalissimi (decomposizione e informaton hiding) per risolvere tutti i problemi connessi con la progettazione del software. Anche se fosse vero, detto così, è fuorviante perchè non mette in evidenza che la principale difficoltà sta proprio nel capire *come* applicare i due principi. Con il crescere della complessità degli esempi, questo diventerà, inesorabilmente, sempre più chiaro.

Esercizi

[Cascata] Sfruttando la nuova versione della classe `CharRect`, scrivere un programma che legge una parola e la stampa a mo' di cascata come nel seguente esempio: (la parola è `"CASCATA"`)

```
C
 AA
 AASS
  SSSCC
  SSSCCAA
   CCCCAATT
   CCCCAATTAA
    AAAAATTAA
    AAAAATTAA
     TTTTTTAA
     TTTTTTAA
      AAAAAAA
      AAAAAAA
```

[Scalinata] Sfruttando la nuova versione della classe `CharRect`, scrivere un programma che legge una parola e la stampa a mo' di scalinata come nel seguente esempio: (la parola è `"SCALINATA"`)

```
S
S
S
```

```
SCCC
SCCC
SCCC
SCCCAAA
SCCCAAA
SCCCAAA
SCCCAAALLL
SCCCAAALLL
SCCCAAALLL
SCCCAAALLLIII
SCCCAAALLLIII
SCCCAAALLLIII
SCCCAAALLLIIINNN
SCCCAAALLLIIINNN
CCCCAAALLLIIINNN
AAAAAAALLLIIINNNAAA
LLLLLLLLLLIIINNNAAA
IIIIIIIIIIIINNNAAA
NNNNNNNNNNNNNNNAAATTT
AAAAAAAAAAAAAAAAAATTT
TTTTTTTTTTTTTTTTTTTTT
AAAAAAAAAAAAAAAAAAAAAAA
```

[Tessiture] Aggiungere alla nuova versione della classe `CharRect` dei metodi per stampare i rettangoli in altre tessiture, ad esempio quelle degli esercizi [Strisce orizzontali], [Scacchiera] e [Scacchiera su misura].
[Collisione] Si consideri un mezzo di stampa che è uguale a quello che "stampa" su una matrice di caratteri con la differenza che quando si invoca il metodo `printChar(row, col, c)` se nella posizione `(row, col)` non c'è il carattere spazio `' '` allora il metodo scrive in quella posizione un carattere fissato (detto carattere di collisione), se invece c'è un carattere spazio allora il metodo scrive il carattere `c`. Aggiungere alla classe `PrintMedium` un mezzo di stampa come quello appena descritto. Per fare ciò aggiungere un costruttore `PrintMedium(char[][] screen, char collision)`, dove `collision` è il carattere di collisione. Il seguente esempio mostra la differenza tra l'usuale mezzo di stampa su matrice di caratteri e il nuovo mezzo di stampa con carattere di collisione `'#'`:

```
stampa senza collisioni          stampa con collisioni
AAAA                             AAAA
AABBBB                           AA##BB
AABBBB                           AA##BB
  BBBB                             BBBB
```

[Piramidi_2] Definire una classe `CharPyramid` per stampare triangoli come è descritto nell'esercizio [Piramidi]. La classe, al pari della nuova versione di `CharRect`, deve usare la classe `PrintMedium`.

Tipi enumerativi

In Java è possibile definire dei tipi che svolgono una funzione analoga a quella svolta dalle `enum` del C. Anche se Java usa la stessa parola chiave, le `enum` di Java sono molto più sofisticate di quelle del C. La prima differenza è che mentre in C gli elementi di una `enum` sono delle costanti numeriche, in Java sono invece degli oggetti (costanti). Vediamo subito un esempio che rappresenta i giorni della settimana:

```
public enum Giorno { LUN, MAR, MER, GIO, VEN, SAB, DOM }
```

Concettualmente questa definizione è equivalente all'aver dichiarato una classe `Giorno` e all'aver creato 7 oggetti di tale classe assegnando i loro riferimenti a 7 campi statici costanti (`final`) chiamati `LUN`, `MAR`, ... `DOM`. Infatti, per poter accedere agli elementi della `enum Giorno` si usa la stessa sintassi per l'accesso a campi statici. Ad esempio, per accedere all'elemento `LUN` si deve scrivere `Giorno.LUN`. Però una `enum` è una classe speciale perchè nessun'altro oggetto può essere istanziato oltre a quelli dichiarati nella definizione dell'`enum`. Gli oggetti di una `enum` sono chiamati *costanti enum*. Il tipo delle `enum` è un tipo classe (class type).
Uno dei vantaggi delle `enum` di Java rispetto alle omonime del C sta nel fatto che, essendo le costanti enum degli oggetti, il loro tipo può essere controllato dal compilatore. Inoltre, hanno altre caratteristiche che le rendono molto versatili e potenti. Adesso ne vedremo solamente alcune le altre saranno discusse più avanti.
Prima di tutto, vediamo un esempio che mostra uno degli usi più comuni delle `enum`. Consideriamo un metodo che prende in input una costante della `enum Giorno` e ritorna una stringa contenente il nome del

corrispondente giorno della settimana.

```
public static String nomeGiorno(Giorno g) {
    String nomeG = null;
    switch (g) {
        case LUN: nomeG = "Lunedì"; break;
        case MAR: nomeG = "Martedì"; break;
        case MER: nomeG = "Mercoledì"; break;
        case GIO: nomeG = "Giovedì"; break;
        case VEN: nomeG = "Venerdì"; break;
        case SAB: nomeG = "Sabato"; break;
        case DOM: nomeG = "Domenica"; break;
    }
    return nomeG;
}
```

Nelle clausole `case` di uno `switch` relativo ad una `enum` non bisogna usare il nome completo delle costanti enum ma solamente il nome semplice. Così, ad esempio, si è usato `LUN` invece di `Giorno.LUN`. Fra poco vedremo che il metodo `nomeGiorno()` può essere scritto in modo più compatto sfruttando alcune caratteristiche offerte dalle `enum` di Java. Ed ecco un elenco delle principali caratteristiche.

- Siccome le costanti enum si riferiscono ad oggetti immutabili è sempre possibile confrontare due costanti enum tramite l'operatore di uguaglianza `==`.
- Le costanti enum hanno un metodo `toString()` che ritorna la rappresentazione tramite stringa della costante. Ad esempio, `Giorno.DOM.toString()` ritorna la stringa `"DOM"`.
- I tipi `enum` hanno un metodo statico `valueOf()` che fa l'inverso di ciò che fa `toString()`. Ad esempio, `Giorno.valueOf("SAB")` ritorna la costante `Giorno.SAB`.
- I tipi `enum` hanno un metodo statico `values()`che ritorna un array le cui componenti contengono i riferimenti a tutte le costanti enum (nell'ordine in cui sono state dichiarate). Ogni volta che il metodo è invocato un nuovo array è creato.
- I tipi `enum` possono avere campi, metodi e costruttori. Quando una costante enum è dichiarata può invocare uno dei costruttori definiti.

Sfruttando l'ultima caratteristica possiamo scrivere una versione più raffinata della `enum Giorno`:

```
public enum Giorno { LUN("Lunedì"), MAR("Martedì"), MER("Mercoledì"),
    GIO("Giovedì"), VEN("Venerdì"), SAB("Sabato"), DOM("Domenica");    // si noti il ";"

    private String nome;
    Giorno(String nome) { this.nome = nome; }
    public String getNome() { return nome; }
}
```

Si noti che se la definizione di una `enum` ha un corpo allora dopo l'ultima costante ci deve essere un punto e virgola. Inoltre, il costruttore non può avere un modificatore di accesso. Ora il metodo `nomeGiorno()` può essere riscritto in modo molto più semplice:

```
public static String nomeGiorno(Giorno g) {
    return g.getNome();
}
```

Il metodo seguente mostra un esempio di uso del metodo `values()` per stampare i giorni della settimana:

```
public static void stampaGiorni() {
    Giorno[] giorni = Giorno.values();
    for (int i = 0 ; i < giorni.length ; i++)
        System.out.println(giorni[i].getNome());
}
```

Esercizi

[Date_versione_2] Rifare l'esercizio [Date] usando una `enum` (nidificata nella classe `Data`) per rappresentare i nomi dei mesi.
[Date_giorni] Aggiungere alla classe dell'esercizio precedente un metodo che stampa la data con il giorno della settimana (ad esempio, `sabato 7 marzo 2009`). Usare una `enum` per i nomi dei giorni della settimana.

[Calendario] Scrivere un programma che legge il nome di un mese e un anno e stampa il calendario di quel mese. Ad esempio, se l'input è `"marzo" 2009`, allora il programma stampa:

```
        Marzo 2009
Lun Mar Mer Gio Ven Sab Dom
                        1
 2   3   4   5   6   7   8
 9  10  11  12  13  14  15
16  17  18  19  20  21  22
23  24  25  26  27  28  29
30  31
```

Errori ed eccezioni

Durante l'esecuzione di un programma possono verificarsi degli errori. Per alcuni di questi il programma non può fare nulla se non "accorgersi" dell'errore e terminare in modo grazioso (ad esempio, se la memoria non è sufficiente), mentre per altri può in qualche modo cercare di riparare all'errore (ad esempio, se si è tentato di leggere un file inesistente, il programma può chiedere all'utente un nuovo nome di file). Il linguaggio Java offre dei meccanismi (ripresi dal C++) che permettono di gestire gli errori che avvengono durante l'esecuzione. Questi errori sono chiamati *eccezioni* (*exceptions*).

Quando si verifica un errore durante l'esecuzione di un programma, ovvero si verifica una eccezione, la macchina virtuale Java *lancia* (*throws*) una opportuna eccezione. Una eccezione è un oggetto il cui tipo è relativo alla natura dell'errore. Ecco subito due semplici esempi:

```
public class Test {
    public static void main(String[] args) {
        String str = null;
        int n = str.length();
    }
}
```

eseguendo il programma la JVM lancerà la seguente eccezione

```
Exception in thread "main" java.lang.NullPointerException
        at Test.main(Test.java:4)
```

mentre se si esegue quest'altro programma

```
public class Test {
    public static void main(String[] args) {
        int n = 1000, d = 0;
        int frazione = n/d;
    }
}
```

la JVM lancerà un'altra eccezione

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Test.main(Test.java:4)
```

Nel primo caso l'errore è dovuto al tentativo di accedere ad un oggetto inesistente (la variabile `str` ha valore `null`) e questo produce una eccezione di tipo `NullPointerException`. Nel secondo caso l'errore è dovuto al tentativo di dividere per zero e questo produce una eccezione di tipo `ArithmeticException`. Entrambi le eccezioni, insieme a molte altre, appartengono al package `java.lang`. E come tutte le eccezioni sono delle classi. Così `NullPointerException` è il nome di una classe e quando si tenta di accedere ad un oggetto tramite un riferimento `null` la JVM istanzia un oggetto di tipo `NullPointerException` e lo "lancia". Se nessuna parte del programma lo "cattura" la JVM termina l'esecuzione del programma riportando l'eccezione. Il meccanismo per la gestione degli errori o eccezioni offerto da Java si basa proprio sulla possibilità di *catturare* (*catch*) le eccezioni che possono venir lanciate durante l'esecuzione.

Catturare eccezioni Per catturare le eccezioni che possono essere lanciate durante l'esecuzione di una porzione del codice si usa la seguente sintassi:

```
try {
    <una o più istruzioni che si vogliono monitorare>
```

```
}
catch (ExceptionType1 ex) {
        // eseguito solo se si verifica un eccezione di tipo ExceptionType1
    <gestione dell'eccezione di tipo ExceptionType1>
}
catch (ExceptionType2 ex) {
        // eseguito solo se si verifica un eccezione di tipo ExceptionType2
    <gestione dell'eccezione di tipo ExceptionType2>
}
.
.
.
catch (ExceptionTypeN ex) {
        // eseguito solo se si verifica un eccezione di tipo ExceptionTypeN
    <gestione dell'eccezione di tipo ExceptionTypeN>
}
finally {    // eseguito anche se si verifica una eccezione (catturata o meno)
    <codice che è eseguito in ogni caso>
}
```

Vediamo subito come esempio un programma che legge due interi, ne calcola il quoziente e poi legge una stringa e un intero `p` e stampa il carattere in posizione `p` della stringa.

```
import java.util.*;             // serve sia per Scanner che per InputMismatchException
import static java.lang.System.*;

public class Test {
    public static void main(String[] args) {
        Scanner input = new Scanner(in);
        try {
            out.print("Inserire due interi: ");
            int n = input.nextInt();    // può lanciare InputMismatchException
            int m = input.nextInt();    // può lanciare InputMismatchException
            int quoziente = n/m;    // può lanciare ArithmeticException
            out.println(n+" / "+m+"  fa  "+quoziente);
            out.print("Inserire una parola e una posizione: ");
            String par = input.next();
            int p = input.nextInt();    // può lanciare InputMismatchException
            char car = par.charAt(p - 1);  // può lanciare StringIndexOutOfBoundsException
            out.println("Il carattere in pos. "+p+" di \""+par+"\" è "+car);
        } catch (InputMismatchException ex) {
            out.print("Eccezione InputMismatchException: ");
            out.println(ex.getMessage());
        } catch (ArithmeticException ex) {
            out.print("Eccezione ArithmeticException: ");
            out.println(ex.getMessage());
        } finally {
            out.println("Questa stampa viene sempre eseguita");
        }
        out.println("Il programma è terminato normalmente");
    }
}
```

Una eccezione di tipo `InputMismatchException` (appartenente al package `java.util`) può essere lanciata dai metodi (`nextInt()`, `nextDouble()`, ecc.) degli oggetti di tipo `Scanner` quando ciò che è letto in input non corrisponde al tipo atteso. Il metodo `getMessage()` è uno de metodi comuni a tutti i tipi di eccezione e ritorna una stringa contenente una descrizione dettagliata dell'eccezione (può anche ritornare `null`). Vediamo ora alcuni esempi di esecuzione del programma che mostrano cosa accade quando si verificano vari tipi di eccezione. Iniziamo con un esempio in cui non si verificano errori:

```
Inserire due interi: 12 3
12 / 3  fa  4
Inserire una parola e una posizione: Albero 3
Il carattere in pos. 3 di "Albero" è b
Questa stampa viene sempre eseguita
Il programma è terminato normalmente
```

Si noti che il codice delle clausole `catch` non è stato eseguito perchè nessun errore si è verificato. Mentre il codice della clausola `finally` è eseguito comunque. Nel seguente esempio si verifica un errore dovuto alla divisione per zero:

```
Inserire due interi: 12 0
```

```
Eccezione ArithmeticException: / by zero
Questa stampa viene sempre eseguita
Il programma è terminato normalmente
```

Si osservi che non appena si verifica una eccezione in un blocco `try {...}` il controllo passa al blocco di una clausola `catch`, se esiste, che cattura l'eccezione. Se esiste un blocco `finally` questo viene sempre seguito. Poi se l'eccezione è stata catturata da una opportuna clausola `catch` allora il controllo passa alla prossima istruzione, altrimenti il programma termina immediatamente. Il prossimo esempio mostra il verificarsi di un errore dovuto ad un input errato:

```
Inserire due interi: 12 tre
Eccezione InputMismatchException: null
Questa stampa viene sempre eseguita
Il programma è terminato normalmente
```

Ora invece consideriamo un esempio in cui si verifica una eccezione che non è catturata:

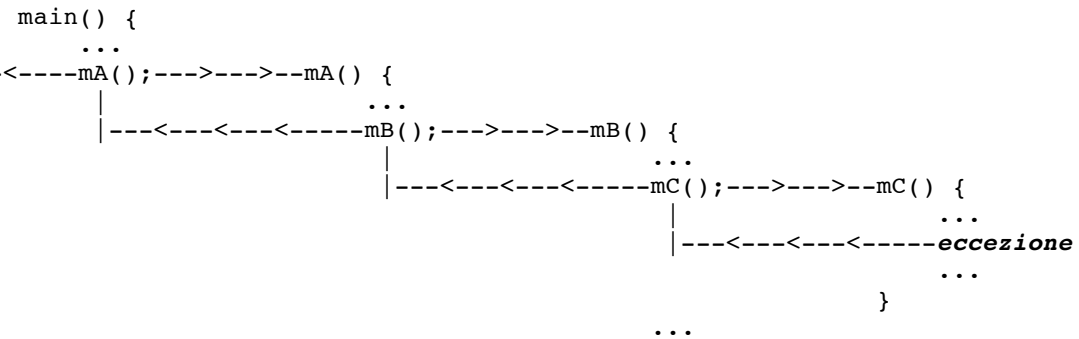
```
Inserire due interi: 12 3
12 / 3  fa  4
Inserisci una parola e una posizione: Albero 7
Questa stampa viene sempre eseguita
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 6
        at java.lang.String.charAt(String.java:558)
        at Test.main(Test.java:16)
```

In questo caso si è verificata una eccezione `StringIndexOutOfBoundsException` dovuta al tentativo di accedere a un carattere inesistente di una stringa. Si noti che il blocco di `finally` è stato comunque eseguito mentre le istruzione successive del programma non sono state eseguite.

Lanciare eccezioni Un metodo può creare e lanciare una eccezione o rilanciare una eccezione. Consideriamo subito un metodo che ritorna il nome dell'`i`-esimo giorno della settimana (per `i = 1,2,3,4,5,6,7`):

```
public static String giorno(int i) {
    if (i < 1 || i > 7)
        throw new IllegalArgumentException("deve essere compreso tra 1 e 7");
    Giorno[] giorni = Giorno.values();
    return giorni[i].getNome();
}
```

Tramite la parola chiave `throw` il metodo lancia una eccezione di tipo `IllegalArgumentException` (appartenente al package `java.lang`). L'esecuzione del metodo termina con il lancio dell'eccezione. L'oggetto di tipo `IllegalArgumentException` che viene creato, come qualsiasi altro oggetto, tramite l'operatore `new` è inizializzato con una stringa che descrive in dettaglio l'errore e che poi sarà quella ritornata dal metodo `getMessage()`. La parola chiave `throw` può essere usata per lanciare una qualsiasi eccezione anche una che non è stata creata dal metodo ma è dovuta, ad esempio, all'invocazione di un'altro metodo. Più specificatamente, nel blocco di una clausola `catch` può esserci una istruzione `throw` che rilancia l'eccezione che è stata catturata. Ci sono varie situazioni in cui ciò può essere utile ma per adesso non approfondiremo questi casi. Piuttosto è importante chiarire quali sono i possibili percorsi che una eccezione può seguire dal momento che viene creata. Consideriamo, ad esempio, un metodo `mC()` che è stato invocato da un metodo `mB()` che a sua volta è stato invocato da un metodo `mA()` che è stato invocato dal metodo `main()`. Se si verifica una eccezione durante l'esecuzione del metodo `mC()` e questa non è catturata all'interno di `mC()` (o è rilanciata), l'eccezione risale al metodo `mB()`, se neanche questo la cattura (o la rilancia), l'eccezione risale al metodo `mA()`, se neanche quest'ultimo la cattura (o la rilancia), l'eccezione risale al `main()` e se neanch'esso la cattura, l'eccezione non catturata causa la terminazione del programma. Il seguente diagramma schematizza il percorso dell'eccezione.



```
                                    }
                                ...
                            }
                ...
        }
```

Eccezioni controllate Alcune eccezioni sono "controllate" dal compilatore nel senso che il compilatore controlla che queste siano state prese in considerazione nel programma. Tutte le eccezioni che abbiamo visto finora e tantissime altre non sono di questo genere e sono chiamate *unchecked exceptions* (*eccezioni non controllate*). Le *checked exceptions* (*eccezioni controllate*) riguardano situazioni che possono normalmente verificarsi durante operazioni di Input/Output. Non si tratta di errori o condizioni che il programmatore può avere sotto il suo controllo (come una divisione per zero o l'accesso ad un array tramite un indice fuori range). Ad esempio, operazioni come la lettura da file o l'apertura di una connessione di rete possono fallire e il programma non può fare nulla per prevenire tali fallimenti perchè i file system e le reti non possono essere sotto il diretto controllo del programma. Quindi per le eccezioni di questo genere (cioè le eccezioni controllate) il compilatore richiede che il programma le gestisca esplicitamente. Questo significa che se si invoca un metodo che può lanciare una eccezione controllata il metodo invocante deve o catturare l'eccezione (tramite una opportuna clausola `catch`) o dichiarare nell'intestazione che può lanciare l'eccezione controllata. Quest'ultima possibilità è spiegata dal seguente esempio:

```
public static void primaLinea(String nomeFile) throws FileNotFoundException {
    Scanner scan = new Scanner(new File(nomeFile));
    String linea = scan.nextLine();
    System.out.println(linea);
}
```

Con la parola chiave `throws` semplicemente si dichiara che il metodo `primaLinea()` può lanciare una eccezione (controllata) di tipo `FileNotFoundException`. Questo tipo di eccezione può essere lanciata dal costruttore `Scanner(new File(nomeFile))` quando il file non esiste. Se questo accade il metodo termina immediatamente la sua esecuzione lanciando l'eccezione. Siccome il metodo `primaLinea()` non cattura l'eccezione controllata ma semplicemente la rilancia questo significa che il metodo invocante `primaLinea()` sarà obbligato a catturare l'eccezione o a rilanciarla. È bene sottolineare che il compilatore, non potendo verificare che le eccezioni controllate siano effettivamente gestite, si accontenta di verificare che esse siano state prese in considerazione o tramite cattura (anche se il blocco della clausola `catch` rimane vuoto) o tramite rilancio con la dichiarazione `throws`.

Quello che abbiamo qui spiegato non esaurisce il tema delle eccezioni. Però esaurisce ciò che potevamo dire delle eccezioni sulla base di quello che abbiamo finora introdotto del linguaggio. Quando avremo introdotto altre caratteristiche di Java potremmo approfondire anche le eccezioni.

Esercizi

[Digitando_interi] Scrivere un programma che legge due interi e ne stampa il prodotto. Prevedere che l'utente possa sbagliare a digitare gli interi e con una opportuna gestione delle eccezioni far sì che il programma in questo caso permetta all'utente di ridigitare l'intero (finché entrambi gli interi non sono digitati in modo corretto).
[Date_corrette] Migliorare il costruttore `Data(int g, int m, int a)` della classe `Data` (esercizio [Date]) in modo che controlli la correttezza della data `(g, m, a)`. Se la data non è corretta il costruttore deve lanciare una opportuna eccezione di tipo `IllegalArgumentException`.

Tipi per dati persistenti: i file

Tutti i tipi che abbiamo visto finora sono adatti alla manipolazione di dati volatili, cioè, dati che sono creati durante l'esecuzione del programma e che spariscono non appena l'esecuzione termina. Naturalmente, invece, in tantissime occasioni è conveniente, e spesso è anzi necessario, che i dati persistano tra una esecuzione e l'altra del programma. Per fare ciò i dati sono mantenuti in file su qualche unità per la memorizzazione persistente. La piattaforma Java offre una ricca libreria per manipolare dati

persistenti memorizzati in file.

A volte la ricchezza di una libreria software può, soprattutto agli inizi, disorientare e rendere difficoltoso anche il più elementare dei suoi possibili usi. La piattaforma Java, a differenza del C, offre più di 60 classi che hanno a che fare con i *flussi* (*streams*) e che direttamente o indirettamente sono quindi anche connesse all'uso dei file. In questa introduzione alla manipolazione dei file in Java vedremo solamente una parte di questa vasta libreria. Infatti, ci accontenteremo (almeno per il momento) di riuscire a localizzare un file (o una directory) nel file system, ottenere alcuni informazioni basilari (esistenza, nome, lunghezza, ecc.), leggere/scrivere i tipi primitivi in file binari ad accesso casuale e leggere/scrivere in file di testo ad accesso sequenziale.

Localizzare file

Il package `java.io` contiene la maggior parte delle classi che riguardano i file e, più in generale, i flussi. Il package contiene anche la classe `File` che permette di ottenere facilmente informazioni su file e directory. Però non permette di fare operazioni di lettura/scrittura, per queste ci sono, come vedremo presto, molte altre classi. Comunque la classe `File` è utile anche per le classi che manipolano i file perché gli oggetti di tale classe forniscono una conveniente specifica della localizzazione dei file.

Un oggetto di tipo `File` rappresenta un *pathname* (indipendente dal sistema e per questo più precisamente chiamato *pathname astratto*) che specifica la posizione di un file o una directory in un file system. Il file o la directory non devono necessariamente esistere. Vale a dire gli oggetti di tipo `File` possono anche specificare la posizione (tramite un pathname astratto) di un ipotetico file o directory che attualmente non esiste nel file system. La classe definisce molti costruttori e metodi, noi ne vedremo solo alcuni. Eccone l'elenco con le relative spiegazioni:

`File(String pathname)`
: Crea un oggetto di tipo `File` convertendo il `pathname` in un pathname astratto. Il file non viene "aperto" (cioè non viene predisposto un canale per poter leggere/scrivere il file) né viene creato se non esiste. Il `pathname` può essere assoluto o relativo. Nel caso sia relativo è inteso relativamente ad una directory di default (di solito la directory da cui è stata invocata la JRE per eseguire il programma Java). Inoltre, se la stringa `pathname` è vuota l'oggetto creato rappresenta il pathname astratto della directory di default ma non rappresenta la posizione di quella directory (questo significa che alcuni metodi non sono applicabili, vedi sotto).

`File(File parent, String child)`
: Crea un oggetto di tipo `File` a partire dal pathname astratto di una (directory) `parent` e dal pathname `child`. Il pathname `child` è interpretato come relativo anche quando è assoluto. Ad esempio, se l'oggetto `parent` rappresenta il pathname `/Users/RIK/JAVA/NetBeansProjects` (nella sintassi UNIX/Linux) della directory `NetBeansProjects` e `child` è il pathname `"Metodologie/dati.txt"` allora l'oggetto creato rappresenterà il pathname `/Users/RIK/JAVA/NetBeansProjects/Metodologie/dati.txt`.

`String getAbsolutePath()`
: Ritorna il pathname assoluto (dipendente dal sistema) dell'oggetto.

`String getParent()`
: Ritorna il pathname della directory che contiene il file/directory rappresentata dall'oggetto. Se l'oggetto è stato creato con un pathname vuoto il metodo ritorna `null`.

`File getParentFile()`
: Ritorna un oggetto di tipo `File` che rappresenta la directory che contiene il file/directory rappresentato dall'oggetto. Ritorna `null` se l'oggetto è stato creato con un pathname vuoto.

`boolean` `exists()`
: Ritorna `true` se l'oggetto rappresenta un file/directory esistente, altrimenti ritorna `false`.

`String getName()`
: Ritorna il nome del file/directory rappresentato dall'oggetto. Se l'oggetto è stato creato con un pathname vuoto il metodo ritorna la stringa vuota.

`boolean` `isDirectory()`
: Ritorna `true` se l'oggetto rappresenta una directory esistente, altrimenti ritorna `false`.

`long` `length()`
: Ritorna la lunghezza in bytes del file rappresentato dall'oggetto. Se il file non esiste ritorna 0 e se è una directory il valore ritornato non è specificato.

`boolean` `canRead()`
: Ritorna `true` se e solo se il file/directory rappresentato dall'oggetto esiste e può essere letto dal programma.

`boolean` `canWrite()`
: Ritorna `true` se e solo se il file/directory rappresentato dall'oggetto esiste e può essere scritto dal

programma.

`File[] listFiles()`

Ritorna un array di riferimenti ad oggetti di tipo `File` rappresentanti tutti i file/directory direttamente contenuti nella directory rappresentata dall'oggetto. Se la directory è vuota l'array ritornato avrà zero componenti. Se l'oggetto non rappresenta una directory, il metodo ritorna `null`.

Vediamo subito all'opera la classe `File` in un programma che stampa alcune informazioni circa la directory di default e alcuni file e directory limitrofe.

```
import static java.lang.System.*;
import java.io.*;        // serve per la classe File

public class FileTest {
        /* metodo di uso generale che stampa le principali informazioni circa un
           file/directory rappresentato da un oggetto di tipo File */
    public static void printFileInfo(File f) {
        out.println("Path: \""+f.getAbsolutePath()+"\"");
        if (f.exists()) {
            boolean isDir = f.isDirectory();
            out.println((isDir ? "DIR" : "FILE")+" Name: \""+f.getName()+"\"");
            out.print("    Readable: "+(f.canRead() ? "YES" : "NO"));
            out.println("  Writable: "+(f.canWrite() ? "YES" : "NO"));
            if (isDir) {
                File[] ff = f.listFiles();
                for (int i = 0 ; i < ff.length ; i++) {
                    out.printf("    %-5d%-5s", i+1, (ff[i].isDirectory() ? "Dir" : "File"));
                    out.printf("  \"%s\"\n", ff[i].getName());
                }
            } else out.println("    Length: "+f.length()+" bytes");
        } else out.println("    Does not exist!");
    }
    public static void main(String[] args) {
        File f = new File("");    // pathname vuoto: pathname della directory
        printFileInfo(f);         // di default, ma non rappresenta tale directory
        File cd = new File(f.getAbsolutePath());    // oggetto che rappresenta la
        printFileInfo(cd);                          // la directory di default
        File sd = new File(cd, "nbproject");    // sotto-directory della directory
        printFileInfo(sd);                      // di default
        File sf = new File(cd, "manifest.mf");  // file nella directory di default
        printFileInfo(sf);
        File par = new File(cd.getParentFile().getParent());    // la directory due
        printFileInfo(par);                // livelli sopra la directory di default
    }
}
```

Il risultato dell'esecuzione di questo programma su un certo sistema è il seguente:

```
Path: "/Users/RIK/ProgProjects/JAVA/NetBeansProjects/Metodologie"
    Does not exist!
Path: "/Users/RIK/ProgProjects/JAVA/NetBeansProjects/Metodologie"
DIR Name: "Metodologie"
    Readable: YES  Writable: YES
    1    Dir    "build"
    2    File   "build.xml"
    3    Dir    "lib"
    4    File   "manifest.mf"
    5    Dir    "nbproject"
    6    Dir    "src"
    7    Dir    "test"
Path: "/Users/RIK/ProgProjects/JAVA/NetBeansProjects/Metodologie/nbproject"
DIR Name: "nbproject"
    Readable: YES  Writable: YES
    1    File   "build-impl.xml"
    2    File   "genfiles.properties"
    3    Dir    "private"
    4    File   "project.properties"
    5    File   "project.xml"
Path: "/Users/RIK/ProgProjects/JAVA/NetBeansProjects/Metodologie/manifest.mf"
FILE Name: "manifest.mf"
    Readable: YES  Writable: YES
    Length: 82 bytes
Path: "/Users/RIK/ProgProjects/JAVA"
```

```
DIR Name: "JAVA"
    Readable: YES  Writable: YES
    1    Dir    "ANTLRWorks-1.1.7"
    2    Dir    "ArgoUML-0.24"
    3    Dir    "BlueJ 2.2.1a"
    4    File   "com.horstmann.violet-0.21.0.jar"
    5    Dir    "corejava8"
    6    Dir    "docs"
    7    Dir    "eclipse"
    8    Dir    "JavaExamples3"
    9    File   "jcip-annotations-src.jar"
    10   Dir    "NetBeans"
    11   File   "netbeans-6.5-ml-macosx.dmg"
    12   Dir    "NetBeansProjects"
    13   Dir    "openjdk"
    14   Dir    "ParsingDocs"
    15   Dir    "workspace"
```

Esercizi

[Numero_file/dir] Modificare il metodo `printFileInfo(File f)` così che quando `f` è una directory stampa accanto ad ogni eventuale sotto-directory il numero di file/directory in essa contenuti. Se la directory è `/Users/RIK/ProgProjects/JAVA/NetBeansProjects/Metodologie`, come nell'esempio precedente, allora il metodo stampa:

```
Path: "/Users/RIK/ProgProjects/JAVA/NetBeansProjects/Metodologie"
DIR Name: "Metodologie"
    Readable: YES  Writable: YES
    1    Dir    "build"    (1)
    2    File   "build.xml"
    3    Dir    "lib"    (5)
    4    File   "manifest.mf"
    5    Dir    "nbproject"    (5)
    6    Dir    "src"   (1)
    7    Dir    "test"    (0)
```

[Conta_file] Implementare un metodo `int countFile(String dirPath)` che ritorna il numero di tutti i file contenuti nella directory specificata da `dirPath` e nelle sue eventuali sotto-directory (nidificate a un qualsiasi livello di profondità).
[Trova_file] Implementare un metodo `String findFile(String dirPath, String fname)` che ritorna, se esiste, il pathname assoluto di un file di nome `fname` contenuto nella directory specificata da `dirPath` o nelle sue eventuali sotto-directory (nidificate a un qualsiasi livello di profondità). Se il file non esiste ritorna `null`.
[Trova_file_ext] Implementare un metodo `File[] findFileExt(String dirPath, String ext)` che ritorna un array di oggetti di tipo `File` relativi a tutti i file con estensione data dalla stringa `ext` (ad esempio `"txt"` o `"java"`) contenuti nella directory specificata da `dirPath` o nelle sue eventuali sotto-directory (nidificate a un qualsiasi livello di profondità). Se non ci sono file con quella estensione allora ritorna un array con zero componenti.
[Profondità] Per *profondità* di una directory intendiamo il numero massimo di livelli di sotto-directory nidificate contenute nella directory. Più precisamente possiamo definire induttivamente la profondità di una directory nel seguente modo. Una directory che non ha file/directory contenuti in essa ha profondità zero. Altrimenti, la sua profondità è pari a p+1, dove p è la massima profondità di un file/directory direttamente contenuto nella directory (per convenzione la profondità di un file è zero). Implementare un metodo `int depth(String pathname)` che ritorna la profondità del file/directory specificato da `pathname`.
[Info_file/dir] Definire una classe `FileDirInfo` per rappresentare file/directory con metodi che forniscono diversi generi di informazioni circa gli oggetti rappresentati. La classe dovrebbe implementare i seguenti costruttore e metodi:

`FileDirInfo(String pathname)`
: Costruisce un nuovo oggetto relativo al file/directory specificato da `pathname`.

`void print()`
: Stampa le informazioni relative all'oggetto come il metodo `printFileInfo()`.

`String find(String fname)`
: Come il metodo `findFile()` dell'esercizio [Trova_file]. Se l'oggetto non è una directory allora

ritorna `null`.

`int depth()`

Come il metodo `depth()` dell'esercizio [Profondità].

Accesso random

La classe `RandomAccessFile` permette di manipolare un file tramite un accesso random. In un modo che è simile a quello delle funzioni della libreria standard del C (`fopen()`, `fseek()`, `fread()`, `fwrite()`). L'accesso random permette di vedere un file come un array di bytes. C'è una sorta di *cursore* (chiamato *file pointer*) che indica la posizione nel file a partire dalla quale avverrà la prossima operazione di lettura/scrittura. Dopo che l'operazione è stata effettuata il cursore avanza automaticamente alla fine del blocco letto/scritto. La classe `RandomAccessFile` è adatta per leggere e scrivere file binari e non file di testo. Ecco l'elenco dei principali costruttori e metodi della classe `RandomAccessFile` con le relative spiegazioni.

`RandomAccessFile(String name, String mode)`

Crea un flusso ad accesso random relativo al file specificato dal pathname `name`. La modalità di lettura/scrittura è determinata dalla stringa `mode`:

- `"r"` Apertura in sola lettura. Se il file non esiste lancia una eccezione controllata `FileNotFoundException`
- `"rw"` Apertura in lettura e scrittura. Se il file non esiste tenta di crearlo.
- `"rws"` Apertura in lettura e scrittura, come `"rw"`, inoltre, qualsiasi aggiornamento del contenuto del file o dei suoi meta-dati (ultima data di accesso, numero di letture/scritture, ecc.) è scritto immediatamente e in modo sincrono nella unità fisica (quasi sempre un disco) che mantiene il file.
- `"rwd"` Come `"rws"` con la differenza che l'aggiornamento dei meta-dati non è richiesto che avvenga in modo sincrono.

Nelle modalità che iniziano con `"rw"` se per qualche ragione il file non può essere aperto viene lanciata una eccezione controllata `FileNotFoundException`. Questo può accadere se, ad esempio, il file non è un file regolare scrivibile o se il file non può essere creato.

`void` `close()`

Chiude il flusso ad accesso random e rilascia tutte le risorse associate. Può lanciare una eccezione controllata `IOException`.

`long` `getFilePointer()`

Ritorna la posizione attuale del cursore (il numero di bytes dall'inizio del file). Può lanciare una eccezione controllata `IOException`.

`void` `seek(`**`long`** `pos)`

Sposta il cursore nella nuova posizione `pos`. Il cursore può anche essere spostato oltre la fine del file. Può lanciare una eccezione controllata `IOException`.

`long` `length()`

Ritorna la lunghezza (in bytes) del file. Può lanciare una eccezione controllata `IOException`.

`void` `setLength(`**`long`** `newLength)`

Imposta la lunghezza del file. Se l'attuale lunghezza è maggiore di `newLength`, il file sarà troncato. Se invece la lunghezza attuale è minore, il file sarà esteso con contenuto indefinito. Può lanciare una eccezione controllata `IOException`.

`byte` `readByte(),` **`short`** `readShort(),` **`int`** `readInt(),` **`long`** `readLong(),`
`float` `readFloat(),` **`double`** `readDouble(),` **`char`** `readChar()`

Tutti questi metodi leggono i corrispondenti tipi primitivi interpretando la rappresentazione binaria contenuta nel file. Ad esempio, `readShort()` legge e interpreta sempre 2 bytes consecutivi e `readLong()` legge e interpreta sempre 8 bytes consecutivi. In particolare, `readChar()` legge e interpreta sempre 2 bytes consecutivi come rappresentanti un carattere nel formato unicode. Quindi `readChar()` non è adatto per leggere caratteri in un normale file di testo in cui ogni carattere è rappresentato da un singolo byte.

Se il file termina prima che la lettura sia stata completata, questi metodi lanciano una eccezione controllata `EOFException`. Tutti questi metodi possono anche lanciare una eccezione controllata `IOException`.

`void` `writeByte(`**`int`** `v),` **`void`** `writeShort(`**`int`** `v),` **`void`** `writeInt(`**`int`** `v),`
`void` `writeLong(`**`long`** `v),` **`void`** `writeFloat(`**`float`** `v),` **`void`** `writeDouble(`**`double`** `v),`
`void` `writeChar(`**`int`** `v)`

Tutti questi metodi scrivono nel file il valore `v` del corrispondente tipo primitivo in formato

binario. Ad esempio, `writeInt()` scrive sempre il valore rappresentato in 4 bytes. Tutti questi metodi possono lanciare una eccezione controllata `IOException`.

Come esempio di uso dell'accesso random per i file, consideriamo un semplice programma che gestisce una specie di agendina per la memorizzazione di appuntamenti. Ogni appuntamento consiste in una data, un orario e una breve descrizione. Il nostro programma usa un file per memorizzare gli appuntamenti. Le operazioni che il programma permette sono:

- visualizzazione di tutti gli appuntamenti (presenti nell'agenda, cioè, nel file);
- inserimento di un nuovo appuntamento;
- cancellazione di un appuntamento.

Nel file (ad accesso random) ogni appuntamento è memorizzato in un opportuno record. Tutti i record hanno la stessa dimensione per facilitare così l'accesso ad essi, sfruttando l'accesso random. Infatti, ogni record nel file può essere identificato da un indice (il primo record ha indice zero) come le componenti di un array. Il modo in cui un appuntamento è rappresentato in forma di record è incapsulato in una classe `Appuntamento`. Questa classe, i cui oggetti rappresentano appuntamenti, permette di gestire tutte le operazioni di lettura e scrittura di singoli appuntamenti, non solo relativamente al file, ma anche da console. Ciò significa che la struttura interna di un appuntamento e le operazioni che è possibile fare su di esso sono responsabilità esclusiva della classe `Appuntamento`. Mentre la gestione dell'agenda (cioè il file) che contiene gli appuntamenti e delle operazioni di visualizzazione, inserimento e cancellazione sono responsablità di un altra classe chiamata `GestioneAppuntamenti`. Questa classe implementa anche il metodo `main()`. Vediamo prima la definizione della classe `Appuntamento`.

```
import static java.lang.System.*;
import java.util.*;
import java.io.*;

public class Appuntamento {
    private static final int maxDescr = 80;       // numero max di char per la dscrizione
    public static final int size = 7+2*maxDescr;  // lunghezza in bytes di un record

    private int giorno, mese, anno, ore, minuti;
    private String descrizione;

    public static Appuntamento creaDaInput() {  // crea un nuovo appuntamento
        Appuntamento app = new Appuntamento();  // leggendo i dati dall'input
        Scanner input = new Scanner(in);
        out.print("DIGITA GIORNO MESE E ANNO: ");
        app.giorno = input.nextByte();
        app.mese = input.nextByte();
        app.anno = input.nextShort();
        out.print("DIGITA ORA E MINUTI: ");
        app.ore = input.nextByte();
        app.minuti = input.nextByte();
        out.print("DESCRIZIONE (MAX "+maxDescr+"): ");
        input.nextLine();
        app.descrizione = input.nextLine();
        return app;
    }
        // crea un nuovo appuntamento leggendo i dati dal file
    public static Appuntamento creaDaFile(RandomAccessFile f, long indice)
                                          throws IOException {
        cursore(f, indice);                   // porta il cursore del file in posizione
        Appuntamento app = new Appuntamento();  // crea un nuovo oggetto
        app.giorno = f.readByte();              // inizializza i campi del
        app.mese = f.readByte();                // nuovo oggetto leggendoli
        app.anno = f.readShort();               // dal file
        app.ore = f.readByte();
        app.minuti = f.readByte();
        int lungD = f.readShort();              // lunghezza della descrizione
        app.descrizione = "";
        for (int i = 0 ; i < lungD ; i++)
            app.descrizione += f.readChar();
        return app;
    }
        // metodo privato che eventualmente riposiziona il cursore del file
    private static void cursore(RandomAccessFile f, long indice) throws IOException {
        if (f.getFilePointer() != indice*size) f.seek(indice*size);
```

```
    }
        // scrive l'appuntamento nel file
    public void scriviInFile(RandomAccessFile f, long indice) throws IOException {
        cursore(f, indice);             // porta il cursore del file in posizione
        f.writeByte(giorno);            // scrive i campi nel file
        f.writeByte(mese);
        f.writeShort(anno);
        f.writeByte(ore);
        f.writeByte(minuti);
        f.writeShort(descrizione.length());
        for (int i = 0 ; i < maxDescr ; i++)
            f.writeChar(i < descrizione.length() ? descrizione.charAt(i) : ' ');
    }
        // stampa l'appuntamento
    public void stampa() {
        out.print("DATA: "+giorno+"/"+mese+"/"+anno);
        out.println("  ORARIO: "+ore+(minuti > 0 ? ":"+minuti : ""));
        out.println("    DESCRIZIONE: "+descrizione);
    }
}
```

Si noti che non c'è un costruttore esplicito perchè gli oggetti `Appuntamento` sono creati o leggendo i dati dalla console (metodo statico `creaDaInput()`) o leggendo un record dal file (metodo statico `creaDaFile()`). Questo genere di metodi che creano oggetti della classe, senza essere però dei costruttori, sono chiamati metodi fabbrica (*factory methods*). Sono quasi sempre metodi statici.

Vediamo ora la deinizione della classe `GestioneAppuntamenti`.

```
import static java.lang.System.*;
import java.util.*;
import java.io.*;

public class GestioneAppuntamenti {
    static private RandomAccessFile file = null;      // mantiene il file degli appuntamenti
    static private Scanner input = new Scanner(in);
        // apre il file se non è ancora aperto e ritorna il numero di appuntamenti
    static private long preparaFile() throws IOException {
        if (file == null) {
            try {                           // il file si trova nella directory di default
                file = new RandomAccessFile("appuntamenti", "rws");
            } catch (FileNotFoundException ex) {}
        }
        return file.length()/Appuntamento.size;
    }

    static private enum Menu {
        VIS("1. VISUALIZZA APPUNTAMENTI"),
        INS("2. INSERISCI APPUNTAMENTO"),
        CAN("3. CANCELLA APPUNTAMENTO"),
        ESC("4. ESCI");

        public final String voce;
        Menu(String voce) { this.voce = voce; }
    }
        // visualizza gli appuntamenti contenuti nel file
    static private void visualizza() throws IOException {
         n = preparaFile();
        for (int i = 0 ; i < n ; i++) {
            out.printf(" %-3d", i + 1);
            Appuntamento.creaDaFile(file, i).stampa();    // crea un oggetto appuntamento
        }                                                 // leggendolo dal file e lo stampa
    }
        // inserisce un nuovo appuntamento nel file
    static private void inserisci() throws IOException {
        long n = preparaFile();                           // crea un oggetto appuntamento
        Appuntamento.creaDaInput().scriviInFile(file, n); // leggendolo dall'input e poi
    }                                                     // lo scrive nel file
        // cancella un appuntamento dal file
    static private void cancella() throws IOException {
        long n = preparaFile();
        out.print("DIGITA L'INDICE DELL'APPUNTAMENTO: ");
        long indice = input.nextLong();
        if (indice >= 1 && indice <= n) {  // "sposta" l'ultimo appuntamento nella
                                           // posizione di quello cancellato
```

```
            Appuntamento.creaDaFile(file, n - 1).scriviInFile(file, indice - 1);
            file.setLength(file.length() - Appuntamento.size);  // accorcia il file
        } else out.println("INDICE ERRATO");
    }

    static public void main(String[] args) throws IOException {
        Menu[] voci = Menu.values();
        int scelta;
        do {
            out.println("\nNUMERO APPUNTAMENTI: "+preparaFile());
            for (int i = 0 ; i < voci.length ; i++)    // stampa il menù delle
                out.println(voci[i].voce);             // operazioni
            scelta = input.nextInt();
            out.println();
            switch (scelta) {
                case 1: visualizza(); break;
                case 2: inserisci(); break;
                case 3: cancella(); break;
            }
        } while (scelta < 4);
        file.close();
    }
}
```

Si noti come grazie alla `enum Menu` è possibile gestire il menù delle operazioni. In particolare, la stampa del menù è effettuata in modo del tutto uniforme e indipendente dal numero delle voci proprio grazie al metodo `values()` delle `enum`. Purtroppo questo livello di uniformità e indipendenza non è stato possibile implementarlo anche per la scelta della voce dal menù. Però fra non molto vedremo che, grazie ad alcune caratteristiche di Java che ancora non abbiamo introdotto, è possibile ottenere questo grado di uniformità e indipendenza anche in questi casi.

Per mantenere le implementazioni semplici entrambe le classi non effettuano controlli né sugli argomenti forniti nelle invocazioni dei vari metodi né sui dati letti dalla console. Ecco ora un esempio di esecuzione del programma (supponiamo che siano già stati inseriti tre appuntamenti).

```
NUMERO APPUNTAMENTI: 3
1. VISUALIZZA APPUNTAMENTI
2. INSERISCI APPUNTAMENTO
3. CANCELLA APPUNTAMENTO
4. ESCI
1

 1  DATA: 15/3/2009  ORARIO: 10:30
    DESCRIZIONE: visita museo
 2  DATA: 16/3/2009  ORARIO: 9
    DESCRIZIONE: riunione importante
 3  DATA: 16/3/2009  ORARIO: 17:30
    DESCRIZIONE: dentista

NUMERO APPUNTAMENTI: 3
1. VISUALIZZA APPUNTAMENTI
2. INSERISCI APPUNTAMENTO
3. CANCELLA APPUNTAMENTO
4. ESCI
2

DIGITA GIORNO MESE E ANNO: 18 3 2009
DIGITA ORA E MINUTI: 19 0
DESCRIZIONE (MAX 80): incontro con Laura

NUMERO APPUNTAMENTI: 4
1. VISUALIZZA APPUNTAMENTI
2. INSERISCI APPUNTAMENTO
3. CANCELLA APPUNTAMENTO
4. ESCI
1

 1  DATA: 15/3/2009  ORARIO: 10:30
    DESCRIZIONE: visita museo
 2  DATA: 16/3/2009  ORARIO: 9
    DESCRIZIONE: riunione importante
 3  DATA: 16/3/2009  ORARIO: 17:30
    DESCRIZIONE: dentista
```

```
 4  DATA: 18/3/2009  ORARIO: 19
    DESCRIZIONE: incontro con Laura

NUMERO APPUNTAMENTI: 4
1. VISUALIZZA APPUNTAMENTI
2. INSERISCI APPUNTAMENTO
3. CANCELLA APPUNTAMENTO
4. ESCI
3

DIGITA L'INDICE DELL'APPUNTAMENTO: 2

NUMERO APPUNTAMENTI: 3
1. VISUALIZZA APPUNTAMENTI
2. INSERISCI APPUNTAMENTO
3. CANCELLA APPUNTAMENTO
4. ESCI
1

 1  DATA: 15/3/2009  ORARIO: 10:30
    DESCRIZIONE: visita museo
 2  DATA: 18/3/2009  ORARIO: 19
    DESCRIZIONE: incontro con Laura
 3  DATA: 32/3/2009  ORARIO: 17:30
    DESCRIZIONE: dentista

NUMERO APPUNTAMENTI: 3
1. VISUALIZZA APPUNTAMENTI
2. INSERISCI APPUNTAMENTO
3. CANCELLA APPUNTAMENTO
4. ESCI
4
```

Esercizi

[Modifica] Aggiungere alla classe `GestioneAppuntamenti` una operazione MODIFICA APPUNTAMENTO per modificare un appuntamento: l'utente digita l'indice dell'appuntamento da modificare, il programma stampa l'appuntamento e chiede se si vuole modificare la data o l'orario o la descrizione. Per implementare questa operazione conviene aggiungere anche qualche metodo alla classe `Appuntamento`?

[Appuntamenti_del_giorno] Modificare le classi `GestioneAppuntamenti` e `Appuntamento` così da poter implementare una operazione che permette di visualizzare tutti gli eventuali appuntamenti relativi ad una data fornita dall'utente.

[Ordine_cronologico] Modificare le classi `GestioneAppuntamenti` e `Appuntamento` in modo tale che nel file gli appuntamenti siano sempre mantenuti in ordine cronologico.

[Rubrica] Definire una o più classi per gestire una rubrica su file. Ogni elemento della rubrica è costituito da un nome (una stringa che può mantenere il nome e cognome di una persona, la denominazione di una società, un negozio, ecc.), uno o più numeri telefonici (si può prevedere per questi una limitazione, ad esempio, al massimo 5 numeri telefonici) e un eventuale indirizzo. Le classi dovrebbero permettere le usuali operazioni di visualizzazione, inserimento, modifica e cancellazione.

[Rubrica+] Modificare le classi dell'esercizio precedente in modo da poter implementare due operazioni di ricerca. Entrambe le operazioni permettono di digitare una stringa. La prima operazione visualizza tutti gli elementi della rubrica il cui campo nome inizia con la stringa data. La seconda invece visualizza tutti gli elementi il cui campo nome contiene come sottostringa la stringa data.

[Contabilità] Definire una o più classi per gestire un archivio di operazioni contabili. L'archivio potrebbe essere utile per mantenere la contabilità di una famiglia o di una piccola impresa o negozio. Ogni operazione contabile è rappresentata da un record che contiene un campo data (la data dell'operazione), un campo descrizione (ad esempio, "bolletta telefonica", "acquisto cancelleria", "stipendio", ecc.) e un campo importo (questo è un numero negativo per una operazione in uscita ed è positivo per una operazione in entrata). Implementare gli usuali servizi di visualizzazione, inserimento, modifica e cancellazione delle operazione contabili.

[Contabilità+] Aggiungere al programma dell'esercizio precedente un servizio che calcola e visualizza un riepilogo contabile: l'utente digita due date e il programma stampa l'importo totale delle uscite, l'importo totale delle entrare e il saldo relativamente a tutte le operazioni contabili avvenute nel periodo tra le due date.

[Appuntamenti+rubrica] Usare le classi per la gestione degli appuntamenti e quelle per la gestione della rubrica (esercizi [Rubrica] e [Rubrica+]) per implementare una versione migliorata dell'operazione dell'esercizio [Appuntamenti del giorno]: per ogni appuntamento nella data specificata e per ogni parola nella sua descrizione visualizzare gli eventuali elementi della rubrica il cui campo nome contiene tale parola.

Accesso sequenziale

A differenza dell'accesso random che dispone di una sola classe (`FileRandomAccess`), l'accesso sequenziale dispone di molte classi. Inoltre queste sono distinte in classi che permettono solamente la lettura e in quelle che permettono solamente la scrittura.

Lettura Una delle classi più convenienti e potenti per l'accesso sequenziale di file di testo in lettura è la classe `Scanner`. Questa l'abbiamo già vista in azione applicata al flusso relativo allo standard input. Ma invero può essere applicata ad ogni flusso di caratteri. L'unica differenza sta nel costruttore usato. Nel caso di un file si può usare il seguente:

`Scanner(File source)`
: Costruisce un nuovo `Scanner` che interpreta i bytes letti dal file `source`. I bytes letti sono convertiti in caratteri tramite una specifica di default della soggiacente piattaforma. Generalmente, questa conversione fa corrispondere un byte per carattere. Il costruttore può lanciare una eccezione controllata di tipo `FileNotFoundException`.

Oltre ai metodi che abbiamo già visto (`next()`, `nextInt()`, ecc.) sono molto utili, soprattutto quando si legge da un file, i metodi che dicono se il prossimo token da leggere è del tipo richiesto (una parola, un intero, ecc.).

`boolean` `hasNext()`, **`boolean`** `hasNextByte()`, **`boolean`** `hasNextInt()`, **`boolean`** `hasNextShort()`, **`boolean`** `hasNexLong()`, **`boolean`** `hasNextFloat()`, **`boolean`** `hasNextDouble()`
: Tutti questi metodi ritornano `true` se e solo se il prossimo token esiste (cioè il flusso contiene ancora qualche carattere non ancora letto) ed è del tipo richiesto. Ad esempio, `hasNextInt()` ritorna `true` se e solo se esiste un prossimo token e questo può essere interpretato come un intero di tipo `int`. Questi metodi non fanno avanzare la scansione del flusso.

Ogni metodo che legge un token fa avanzare la scansione al primo byte non ancora scandito. Purtroppo non c'è un metodo che riporta la scansione all'inizio del file. L'unico modo per riportare all'inizio la scansione è di chiudere l'attuale oggetto `Scanner`, tramite il metodo `void close()` (la chiusura non è obbligatoria, ma permette di risparmiare risorse), e di crearne uno nuovo sempre relativo allo stesso file.

Consideriamo ora una classe molto semplice che, sfruttando `Scanner`, permette di leggere e stampare le linee di un file di testo e di fare alcune altre semplici operazioni.

```
import java.util.*;
import java.io.*;
import static java.lang.System.*;

public class TextFile {
    private File srcFile;       // mantiene la localizzazione del file
    private Scanner fileScan;   // lo Scanner che interpreta il contenuto del file

    public TextFile(String pathname) throws FileNotFoundException {
        srcFile = new File(pathname);
        fileScan = new Scanner(srcFile);
    }
        // salta le prossime skipL linee, poi stampa le successive nL linee e
    public int printNextLines(int skipL, int nL) {    // ritorna il numero di linee
        while (skipL > 0 && fileScan.hasNextLine()) { // effettivamente stampate
            fileScan.nextLine();
            skipL--;
        }
        int count = 0;
        while (count < nL && fileScan.hasNextLine()) {
            out.println(fileScan.nextLine());
            count++;
        }
        return count;
```

```
    }
        // stampa le prossime nL linee di testo del file e ritorna il numero di linee
    public int printNextLines(int nL) {    // effettivamente lette e stampate
        return printNextLines(0, nL);
    }
        // riporta la scansione all'inizio del file
    public void rewind() throws FileNotFoundException {
        fileScan.close();
        fileScan = new Scanner(srcFile);
    }
        // stampa l'intero file (linea per linea) e ritorna il numero di linee
    public int printFile() throws FileNotFoundException {
        rewind();
        int count = 0;
        while (fileScan.hasNextLine()) {
            out.println(fileScan.nextLine());
            count++;
        }
        return count;
    }
        // cerca la parola word nella parte di file non ancora scandita e ritorna
    public long find(String word) {    // il numero di occorrenze trovate
        long count = 0;
        while (fileScan.hasNext()) {
            String w = fileScan.next();
            if (word.equals(w)) count++;
        }
        return count;
    }
        // chiude la scansione. Dopo la chiusura l'unico metodo che può essere
    public void close() {    // usato è rewind()
        fileScan.close();
        fileScan = null;
    }
}
```

Ecco un semplice programma che usa la classe `TextFile`.

```
import java.io.*;
import static java.lang.System.*;

public class FileTest {
    public static void main(String[] args) throws FileNotFoundException {
        TextFile f = new TextFile("testo.txt");
        out.println("L'INTERO FILE:");
        f.printFile();
        out.println("\nLE PRIME TRE LINEE DEL FILE:");
        f.rewind();
        f.printNextLines(3);
        out.println("\nLE SUCCESSIVE DUE LINEE DEL FILE:");
        f.printNextLines(2);
        out.println("\nLE TRE LINEE DOPO LE PRIME CINQUE:");
        f.rewind();
        f.printNextLines(5, 3);
        f.rewind();
        String word = "linea";
        out.println("\nOCCORRENZE DELLA PAROLA \""+word+"\": "+f.find(word));
        f.rewind();
        word = "(VII)";
        out.println("\nOCCORRENZE DELLA PAROLA \""+word+"\": "+f.find(word));
        f.close();
    }
}
```

Infine, ecco una esecuzione del programma:

```
L'INTERO FILE:
1  prima linea (I)
2  seconda linea (II)
3  terza linea (III)
4  quarta linea (IV)
5  quinta linea (V)
6  sesta linea (VI)
7  settima linea (VII)
```

```
8  ottava linea (VIII)
9  nona linea (IX)
10 decima linea (X)
11 undicesima linea (XI)
12 dodicesima linea (XII)

LE PRIME TRE LINEE DEL FILE:
1  prima linea (I)
2  seconda linea (II)
3  terza linea (III)

LE SUCCESSIVE DUE LINEE DEL FILE:
4  quarta linea (IV)
5  quinta linea (V)

LE TRE LINEE DOPO LE PRIME CINQUE:
6  sesta linea (VI)
7  settima linea (VII)
8  ottava linea (VIII)

OCCORRENZE DELLA PAROLA "linea": 12

OCCORRENZE DELLA PAROLA "(VII)": 1
```

Scrittura Per scrivere in un file (di testo) con accesso sequenziale una delle classi più convenienti della piattaforma Java è `PrintStream`. Questa classe l'abbiamo già incontrata perchè è la classe del campo `out` di `System`. Quindi `PrintStream` dispone di tutti quei metodi che già conosciamo e che facilitano la scrittura in forma testuale su un flusso di output (`print()`, `println()`, `printf()`). Tuttavia, se usata direttamente per aprire e scrivere un file, nel caso il file è esistente, questo viene troncato a lunghezza zero. Quindi non permette di aggiornare il contenuto del file appendendo ad esso altre linee di testo, invece lo riscrive dall'inizio. Per avere la possibilità di aggiornare il contenuto di un file esistente (senza quindi cancellarne il contenuto precedente) è necessario aprire preliminarmente il file tramite la classe `FileOutputStream`.

Le due classi `FileOutputStream` e `PrintStream` hanno parecchi costruttori. Noi ne useremo solamente due, uno per classe. La descrizione dei due costruttori è la seguente.

`FileOutputStream(String pathname, boolean append)`
: Crea un flusso di output per scrivere nel file specificato da `pathname`. Se `append` è `true` la scrittura avviene alla fine del file, altrimenti avviene all'inizio del file (cancellando l'eventuale contenuto esistente). Se il file non esiste tenta di crearne uno nuovo. Può lanciare l'eccezione controllata `FileNotFoundException`

`PrintStream(OutputStream out, boolean autoFlush)`
: Crea un flusso di stampa "attaccato" al flusso di output `out`. Il tipo `FileOutputStream` è un sotto-tipo del tipo `OutputStream`. Vedremo più avanti il significato preciso di ciò, per ora è sufficiente dire che un oggetto di tipo `FileOutputStream` può essere usato ovunque è richiesto un oggetto di tipo `OutputStream`. Se `autoFlush` è `true` il buffer di output è svuotato non appena la scrittura di una linea è completata.

La classe `PrintStream` ha un metodo `close()` che chiude il flusso di stampa, svuota il buffer e chiude anche il soggiacente flusso di output (`OutputStream`). La classe `PrintStream` a differenza di molte altre classi che manipolano flussi non lancia eccezioni controllate (come `IOException`). Invece ha un metodo `boolean checkError()` che svuota il buffer e controlla se si è verificato qualche errore. Ritorna `true` se un errore si è verificato, altrimenti ritorna `false`.

Per mostrare come la classe `PrintStream` può essere usata (in congiunzione con `FileOutputStream`) per scrivere in un file di testo, definiamo una semplice classe per gestire un file di log. Vale a dire un file che mantiene annotazioni di eventi di un qualche genere. La nostra semplice classe prevede solamente le operazioni fondamentali: apertura di un file di log (esistente o da creare), aggiunta di una annotazione al file di log (una linea di testo con indicazione automatica della data e l'orario), stampa dell'intero file di log e una operazione di chiusura del file.

```
import java.io.*;
import java.util.*;

public class LogFile {
    private PrintStream wLog;     // per scrivere il log
```

```
    private TextFile rLog;        // per leggere il log

    public LogFile(String pathname) throws FileNotFoundException {
            // apre il flusso di output in modalità append
        FileOutputStream fos = new FileOutputStream(pathname, true);
        wLog = new PrintStream(fos, true);
        rLog = new TextFile(pathname);
    }
        // appende una linea al log
    public void append(String line) {
        GregorianCalendar calendar = new GregorianCalendar();
        wLog.print(calendar.get(Calendar.DAY_OF_MONTH));
        wLog.print("/"+(calendar.get(Calendar.MONTH)+1));
        wLog.print("/"+calendar.get(Calendar.YEAR));
        wLog.print(" "+calendar.get(Calendar.HOUR_OF_DAY));
        wLog.print(":"+calendar.get(Calendar.MINUTE));
        wLog.print(":"+calendar.get(Calendar.SECOND));
        wLog.print(":"+calendar.get(Calendar.MILLISECOND));
        wLog.println("   "+line);
    }
        // stampa l'intero log
    public void print() throws FileNotFoundException {
        rLog.printFile();
    }

    public void close() {
        wLog.close();
        rLog.close();
    }
}
```

La classe `GregorianCalendar` è nel package `java.util`. Il costruttore senza parametri costruisce un oggetto con data e orario inizializzati con la data e l'orario del momento in cui l'oggetto è creato. Ecco un semplice programma che mostra l'uso della classe `LogFile`:

```
import java.io.*;
import static java.lang.System.*;

public class LogFileTest {
    public static void main(String[] args) throws FileNotFoundException {
        String path = "log.txt";
        LogFile log = new LogFile(path);
        out.println("LOG \""+path+"\"");
        log.print();
        log.append("prima linea");
        log.append("seconda linea");
        out.println("\nLOG \""+path+"\"");
        log.print();
        log.append("terza linea");
        out.println("\nLOG \""+path+"\"");
        log.print();
        log.close();
    }
}
```

Il risultato dell'esecuzione del programma iniziando con il file di log `log.txt` vuoto è il seguente:

```
LOG "log.txt"

LOG "log.txt"
13/2/2009 22:17:19:680   prima linea
13/2/2009 22:17:19:681   seconda linea

LOG "log.txt"
13/2/2009 22:17:19:680   prima linea
13/2/2009 22:17:19:681   seconda linea
13/2/2009 22:17:19:682   terza linea
```

Esercizi

[Linee] Aggiungere alla classe `TextFile` un metodo che ritorna il numero di linee del file e un'altro metodo che stampa le ultime *n* linee del file.
[Parole] Aggiungere alla classe `TextFile` un metodo `String nextWord()` che ritorna la prossima parola. Per parola si intende una sequenza massimale di caratteri alfabetici minuscoli o maiuscoli di lunghezza almeno 1. Se la prossima parola non esiste allora ritorna la stringa vuota. Si noti che il metodo `next()` della classe `Scanner` non ritorna una parola seconda la definizione data (ritorna un token che è una sequenza di caratteri, anche non alfabetici delimitata da whitespaces). Il comportamento del metodo `nextWord()` dovrebbe essere tale che se il file di testo contiene:

```
L'indirizzo è il seguente: via Verdi n. 22. La città è Roma (RM).
```

allora il metodo, se invocato ripetutamente, ritorna le seguenti parole:

```
L  indirizzo  il  seguente  via  Verdi  n  La  citt  Roma  RM
```

Suggerimento: Si può operare con i token di default tenendo presente che ogni token può contenere zero, una, o più parole. In alternativa, si può modificare la definizione dei token usando il metodo `useDelimiter()` della classe `Scanner`. Tale metodo permette infatti di impostare come sono delimitati i token. Per default i token sono delimitati da caratteri whitespace (spazio, a-capo, tab, e similari). Se si invoca il metodo nel seguente modo `useDelimiter("[^\\p{Alpha}]+")`, i delimitatori dei token diventano tutti i caratteri non alfabetici. Così i token sono esattamente le parole.
[Log] Aggiungere alla classe `LogFile` un metodo `void extract(int d, int m, int y, String path)` che scrive nel file specificato da `path` tutte le linee di log che hanno la data `(d, m, y)`.
[Log+] Aggiungere alla classe `LogFile` un metodo `void delPrev(int d, int m, int y)` che riscrive il file di log eliminando tutte le linee con data uguale o antecedente alla data `(d, m, y)`.
[Elenchi_di_parole] Definire una classe `ElencoParole` che permette di lavorare con elenchi di parole. Per elenco di parole intendiamo un file che contiene una parola per linea. Sul sito http://gilda.it/giochidiparole/elenchi.htm si possono trovare parecchi di questi elenchi per l'italiano. La classe dovrebbe definire almeno i seguenti costruttore e metodi:

`ElencoParole(String pathname)`
: Costruisce un oggetto di tipo `ElencoParole` relativo al file specificato da `pathname` (che si suppone contenga un elenco di parole).

`int numeroParole()`
: Ritorna il numero di parole dell'elenco.

`boolean trova(String par)`
: Cerca la parola `par` nell'elenco e se la trova ritorna `true`, altrimenti ritorna `false`.

`String[] trova(String[] parArray)`
: Ritorna in un array tutte le parole dell'array `parArray` che compaiono nell'elenco. Se nessuna parola compare nell'elenco ritorna un array con zero componenti.

[Ortografia] Aggiungere alla classe `ElencoParole` dell'esercizio precedente un metodo `String[] ortografia(String par)` che se la stringa `par` non compare nell'elenco allora ritorna in un array tutte le parole dell'elenco che differiscono da `par` solamente in un carattere. Se invece la stringa `par` è presente nell'elenco, allora ritorna `null`. Ecco alcuni esempi:

```
STRINGA DI INPUT            PAROLE RITORNATE
alvero                      albero  altero  alzero  avvero
orologie                    omologie orologio urologie
proglamma                   programma
porgramma                   ARRAY VUOTO
programma                   NULL
ostografia                  optografia ortografia
```

[Anagrammi] Aggiungere alla classe `ElencoParole` dell'esercizio [Elenchi di parole] un metodo `String[] anagrammi(String par)` che ritorna in un array tutte le parole dell'elenco che sono anagrammi della parola `par`. Ecco alcuni esempi:

```
PAROLA DI INPUT             ANAGRAMMI
albero                      albore lobare
torta                       ratto rotta trota
parole                      polare
programma                   NESSUN ANAGRAMMA
giornale                    algerino laringeo regalino rigelano rilegano
```

[Unire_elenchi] Implementare un metodo `uniElenchi(String path1, String path2, String unipath)` che unisce in un nuovo elenco (scritto nel file specificato da `unipath`) tutte le parole degli elenchi contenuti nei file specificati da `path1` e `path2`. Si assume che ogni linea dei file contiene esattamente una parola e che le parole sono ordinate alfabeticamente (come negli elenchi che si trovano

sul sito http://gilda.it/giochidiparole/elenchi.htm). Il nuovo file deve contenere anch'esso le parole una per linea, ordinate alfabeticamente e senza ripetizioni. Ecco un esempio con elenchi molto piccoli:

```
ELENCO 1          ELENCO 2                ELENCO UNIONE
albero            alba                    alba
borsa             albero                  albero
dirottare         botte                   borsa
programma         programma               botte
zuppa             zoppa                   dirottare
                  zuppa                   programma
                                          zoppa
                                          zuppa
```

[Elenchi_ad_accesso_random] Definire una nuova versione della classe `ElencoParole` dell'esercizio [Elenchi di parole] basata su elenchi di parole mantenuti in file ad accesso random così da rendere più efficienti le operazioni di ricerca. Il costruttore, se il file specificato non è ad accesso random (lo potrebbe riconoscere da una particolare estensione, ad esempio `".rndacc"`), lo converte creandone uno nuovo ad accesso random. Questo significa che ogni parola è mantenuta in un record e tutti i record hanno la stessa lunghezza (una lunghezza di 30 bytes è più che sufficiente). Avendo a disposizione l'elenco con le parole memorizzate in ordine alfabetico e in record, tutti della stessa lunghezza, le operazioni di ricerca (come `trova()`) si possono implementare in modo molto più efficiente usando la ricerca binaria.