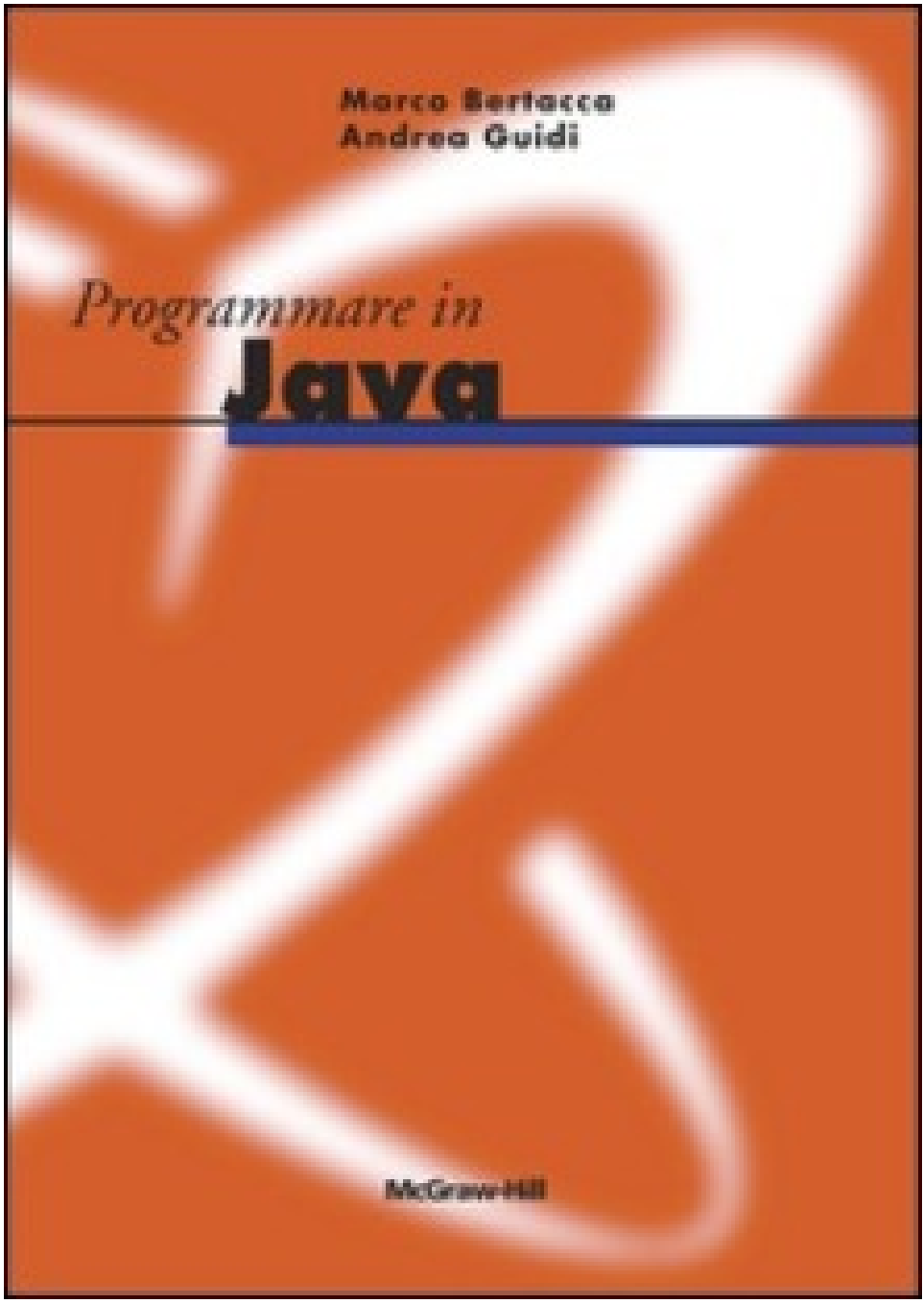
Marco Bertacca
Andrea Guidi

Programmare in Java

McGraw-Hill

Milano • New York • San Francisco • Washington D.C. • Auckland
Bogota • Lisboa • London • Madrid • Mexico City • Montreal
New Delhi • San Juan • Singapore • Sydney • Tokyo • Toronto


Marco Bertacca
Andrea Guidi
Programmare in
Java
McGraw-Hill

Indice

Copyright © 2007 The McGraw-Hill Companies, S.r.l.
Publishing Group Italia
via Ripamonti, 89 – 20139 Milano

McGraw-Hill
A Division of The McGraw-Hill Companies

I diritti di traduzione, di riproduzione, di memorizzazione elettronica e di adattamento totale e parziale con qualsiasi mezzo (compresi i microfilm e le copie fotostatiche) sono riservati per tutti i Paesi.

Date le caratteristiche intrinseche di Internet, l'Editore non è responsabile per eventuali variazioni negli indirizzi e nei contenuti dei siti Internet riportati.

Nomi e marchi citati nel testo sono generalmente depositati o registrati dalle rispettive case produttrici.

Editor: Paolo Roncoroni
Produzione: Donatella Giuliani
Realizzazione editoriale: Carmelo Giarratana
Stampa: Arti Grafiche Murelli, Fizzonasco di Pieve Emanuele (MI)

Printed in Italy
123456789AGMLIL0987
ISBN 978-88-386-6400-7

Indice

Prefazione

Questo libro è una guida graduale all'apprendimento della programmazione con il linguaggio Java, senza che sia richiesto alcun prerequisito in termini di conoscenze ed esperienze di programmazione. L'obiettivo è far conoscere e sperimentare Java nella sua completezza, illustrando le basi della programmazione ad oggetti e applicandone i principi, in modo da mettere in grado il lettore non soltanto di programmare, ma anche di comprendere il perché delle scelte adottate nel linguaggio e far suo un metodo di lavoro.

Gli argomenti trattati vengono presentati in modo graduale e sistematico e sono corredati da numerosi esempi illustrativi. Lo sforzo è stato quello di introdurre ogni proposizione sottolineando l'aspetto pratico e nondimeno mantenendone una descrizione rigorosa e priva di ambiguità. La stessa teoria degli oggetti viene presentata come una naturale evoluzione della programmazione strutturata mostrando a quali esigenze pratiche risponde.

Il libro tratta sostanzialmente del linguaggio in sé, ma approfondisce anche alcuni argomenti correlati, come le librerie per l'interfaccia grafica e per l'utilizzo su Internet, che sono necessari per comprendere come utilizzare il linguaggio in applicazioni complesse. In ogni caso, dato che le librerie sono destinate a mutare nel tempo, non è mai stato dato peso al nozionismo, ma sempre ai principi di base e alle tecniche usate, in modo tale da mettere il lettore in grado di apprendere semplicemente l'utilizzo di nuove librerie.

L'obiettivo non è solo illustrare teoricamente il linguaggio, ma anche fornire indicazioni pratiche d'utilizzo, per tale ragione sono considerate ed evidenziate anche le nuove caratteristiche dell'ultima versione Java 1.6 o, per usare il nome ufficiale "Java™ Platform, Standard Edition 6" o Java™ SE 6. Il motivo delle due differenti numerazioni è brevemente illustrato nel prossimo paragrafo.

7. Vincere giocando per primi al gioco degli undici. Si gioca in due con undici oggetti sul tavolo, per esempio stecchini da denti. Ciascun giocatore, al suo turno, può raccogliere 1, 2 o 3 oggetti. Perde chi raccoglie l'ultimo oggetto. [Il segreto sta nel lasciare all'avversario con la prima mossa un numero di fiammiferi della forma $4i + 1$, dove i è un numero intero].

8. Se nel gioco degli undici (Esercizio 7) il primo giocatore non raccoglie due oggetti nella prima mossa, è possibile realizzare un algoritmo che permetta al secondo giocatore di vincere sempre?

9. L'algoritmo che risolve l'Esercizio 7 può essere generalizzato in modo che il primo giocatore risulti vincente qualsiasi sia il numero di oggetti con cui si gioca? [Si consideri il suggerimento dato all'Esercizio 7.] In caso negativo indicare quando è possibile scrivere un algoritmo che risolve il problema.

10. Calcolare le radici di un'equazione di secondo grado $ax^2 + bx + c = 0$ dati i coefficienti a, b e c.

11. Determinare il minimo comune multiplo (mcm) di due numeri interi positivi.

Capitolo 2

Programmazione

Obiettivi didattici

- Linguaggio
- Programma
- Variabili
- Sequenza, selezione, iterazione
- Blocco d'istruzioni
- Approccio top-down
- Programmazione modulare
- Programmazione orientata agli oggetti

2.1 Linguaggi di programmazione

Un algoritmo scritto in un *linguaggio naturale* come l'italiano, lo spagnolo, il danese o una qualsiasi altra lingua non può essere eseguito direttamente da un elaboratore automatico, un computer. Per poter raggiungere tale scopo occorre esprimere l'algoritmo in un *linguaggio formale* che l'elaboratore sia in grado di interpretare ed eseguire. Un *programma* è appunto un algoritmo trascritto in un linguaggio formale, che per questa ragione è detto anche *linguaggio di programmazione*.

I linguaggi non formali, come il linguaggio naturale, generano ambiguità. La frase "ho visto un uomo con un cannocchiale" può significare "ho visto un uomo che teneva in mano un cannocchiale" oppure "guardando attraverso un cannocchiale ho visto un uomo" (che si presume si trovasse molto lontano). Le due interpretazioni sono accet-

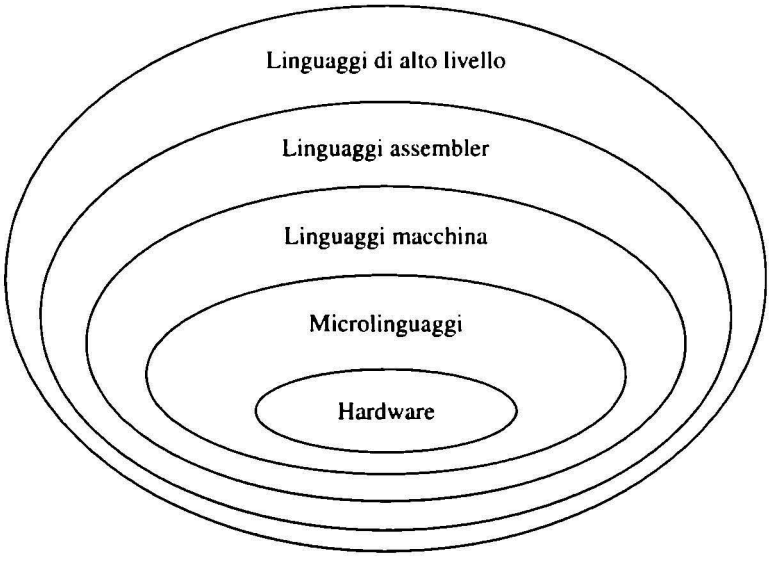


Figura 2.1 Una classificazione dei linguaggi di programmazione

tabili e l'interlocutore umano sceglie quella corretta in base al contesto in cui la frase si inserisce. Un sistema di elaborazione può invece interpretare univocamente solo istruzioni espresse in linguaggi governati da regole grammaticali precise e privi di ogni ambiguità: questi linguaggi sono appunto i linguaggi di programmazione.

I linguaggi di programmazione possono essere classificati in base alla loro natura attraverso una stratificazione in livelli come in Figura 2.1. Negli strati interni subito al di sopra dei microlinguaggi troviamo i *linguaggi di basso livello*: linguaggi macchina e linguaggi assembler, che dipendono dalla struttura fisica del tipo di computer per cui sono stati esplicitamente progettati. Nello strato più esterno troviamo invece i *linguaggi di alto livello* come il C, che sono più vicini alla *forma mentis* dell'uomo, ma i cui programmi per essere interpretati dall'elaboratore devono essere prima tradotti in codice di basso livello.

2.2 Linguaggio macchina

Il linguaggio formale che un computer è in grado di interpretare ed eseguire *senza mediazioni* è il *linguaggio macchina* in cui i programmi, il *codice oggetto*, sono rappresentati da una sequenza di cifre binarie che codificano le istruzioni e i dati su cui lavora la CPU. Dato che alle istruzioni in linguaggio macchina corrispondono operazioni direttamente eseguibili dall'hardware dell'elaboratore, le istruzioni sono strettamente correlate all'architettura dell'elaboratore, cioè alla struttura fisica delle sue

unità e delle loro interconnessioni. Ogni istruzione disponibile è identificata da un codice operativo e dagli operandi su cui agire. Quello che segue è un esempio di istruzione con un solo operando:

Codice operativo dell'istruzione	*Operando*
00000010	000000000011100

Vediamo una porzione di *programma in linguaggio macchina* composto da tre istruzioni. A destra riportiamo il corrispondente codice mnemonico, di cui parleremo più avanti.

Codice operativo	*Operando*	*Codice mnemonico*
00000010	000000011011100	LOAD 220
00000110	000000011111100	SUM 252
00000100	000000011011100	MEM 220

Tutte e tre le istruzioni utilizzate agiscono su un particolare registro della CPU detto *accumulatore*. I registri sono singole unità di memoria interne alla CPU, preposte a contenere i dati iniziali, provenienti dalla memoria centrale, e i risultati delle elaborazioni. La prima istruzione, il cui codice operativo è 00000010, legge il valore contenuto nella cella di memoria specificata dall'operando e lo *carica* nell'accumulatore. Nell'esempio, l'indirizzo binario della cella è 000000011011100, in decimale 220 (per il sistema di numerazione binario si rimanda all'Appendice D): si veda la Figura 2.2*a*.

La seconda istruzione calcola la *somma* del contenuto dell'accumulatore e del valore della cella di memoria d'indirizzo 000000011111100, paria a 252 in decimale, e la scrive nell'accumulatore (Figura 2.2*b*). L'ultima istruzione memorizza il contenuto dell'accumulatore nella locazione di memoria d'indirizzo 000000011011100, la stessa cella da dove era stato caricato il primo addendo (Figura 2.2*c*).

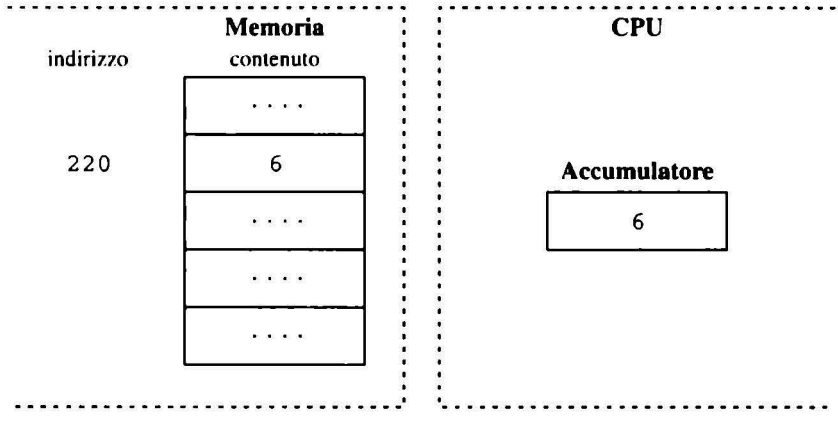


Figura 2.2*a* Stato della macchina dopo la prima istruzione (LOAD 220)

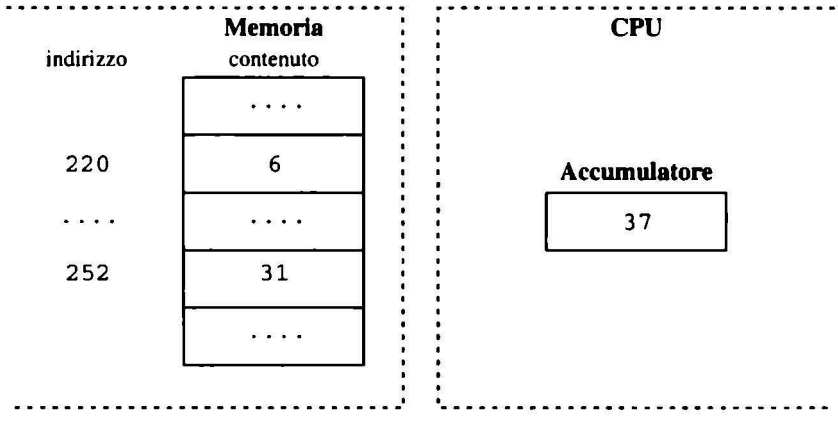


Figura 2.2*b* Stato della macchina dopo la seconda istruzione (`SUM 252`)

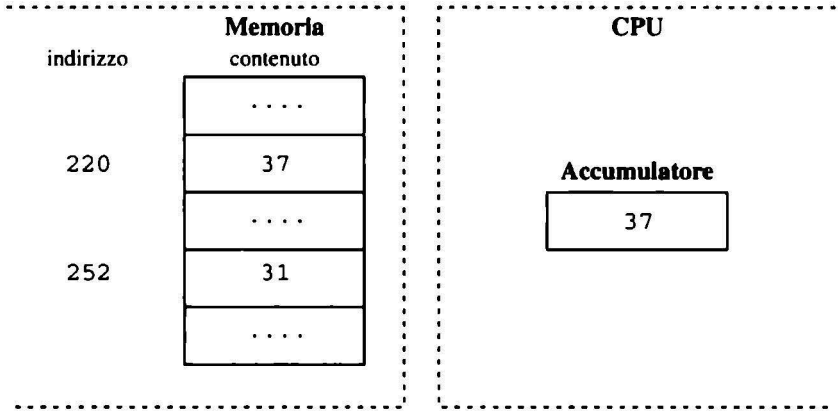


Figura 2.2*c* Stato della macchina dopo la terza istruzione (`MEM 220`)

Un primo, parziale passo verso la semplificazione della programmazione si è compiuto con l'uso dei linguaggi *assembler* in cui le singole istruzioni binarie sono rappresentate con un codice mnemonico più facilmente comprensibile per l'uomo. In assembler il programma precedente potrebbe essere espresso come segue:

```
LOAD 220
SUM  252
MEM  220
```

dove `LOAD` è il termine inglese che sta per "carica", `SUM` per "somma" e `MEM` per "memorizza". Da quando i programmatori hanno potuto iniziare a utilizzare un codice

mnemonico, come quello dell'esempio, la velocità per scrivere, correggere e far evolvere i programmi è aumentata incredibilmente. Dato che l'elaboratore comprende solo il linguaggio macchina, i programmi assembler devono essere comunque tradotti in linguaggio macchina; questo lavoro viene però svolto automaticamente dai programmi traduttori, detti *assemblatori*, forniti dalle case costruttrici degli elaboratori.

✔ **NOTA**

Al di sotto dei linguaggi assembler e macchina, al livello di gerarchia più basso, nel caso dei microprocessori *esiste un ulteriore strato di software: i* microlinguaggi *(si veda la Figura 2.1) in cui l'unità di controllo è in realtà una memoria in cui il costruttore ha cablato le* microistruzioni *per l'esecuzione delle istruzioni in linguaggio macchina. A ogni istruzione in linguaggio macchina corrisponde un microprogramma composto da microistruzioni: sequenze di bit che costituiscono i segnali di controllo che vanno a pilotare i componenti del microprocessore e consentono l'esecuzione delle istruzioni. Ciò significa che uno stesso microprocessore può essere programmato dal costruttore in differenti modi a seconda degli usi che se ne vuole fare: costituire l'unità centrale di un personal computer, realizzare un computer particolarmente adatto a svolgere complesse operazioni matematiche, far parte di una cabina di pilotaggio di un aereo o controllare il funzionamento di un frigorifero. Dunque, in questo caso, esiste un'ulteriore mediazione tra i programmi in linguaggio macchina e la macchina: il microlinguaggio. Al di sotto dei microlinguaggi non si può andare, lì c'è davvero solo il "ferro" (a onor del vero dovremmo scrivere: c'è davvero solo plastica e silicio).*

2.3 Linguaggi di alto livello

A partire dagli anni '50, con la realizzazione dei primi computer commerciali, sono stati sviluppati particolari linguaggi di programmazione il cui impiego risulta ancora più semplice dei linguaggi assembler. Questi linguaggi, sebbene sottoposti come tutti i linguaggi formali a ferree regole di composizione, utilizzano simboli matematici e parole tipiche delle lingue naturali (soprattutto dell'inglese) e sono di facile interpretazione. Essi sono più simili al linguaggio naturale non solo delle successioni apparentemente senza senso di 0 e 1 del linguaggio macchina, ma anche dei programmi assembler; per questo motivo vengono detti *linguaggi evoluti* o di *alto livello*. Appositi software, detti *compilatori*, provvedono quindi a tradurre le istruzioni scritte nei linguaggi evoluti, che compongono il *codice sorgente*, nel loro equivalente in codice eseguibile dalla macchina; vedi il paragrafo successivo.

Fortran, Cobol, Basic e Pascal sono alcuni dei linguaggi di alto livello che per varie ragioni sono hanno raggiunto una grande popolarità. Il Fortran (*FORmula TRANslator*), nato nel 1957 e progettato per lo sviluppo di programmi di tipo scientifico ricchi di formule e procedure matematiche, è oggi utilizzato soprattutto sui grandi computer dedicati alla ricerca scientifica. Il Cobol (*COmmon Business Oriented Language*), nato nel 1960 e pensato per applicazioni di tipo gestionale, benché per

certi aspetti obsoleto riveste ancora oggi importanza, soprattutto per l'eredità del passato di grandi moli di programmi che continuano a servire la gestione di banche, istituzioni pubbliche e grandi aziende. Il Basic (*Beginners All-purpose Symbolic Instruction Code*), nato nel 1964 per avvicinare gli studenti alla programmazione, è di uso generale ed è stato il primo linguaggio di alto livello ad essere applicato sui personal computer. Il Pascal, dal nome del matematico e filosofo francese Blaise Pascal che inventò per primo una macchina che eseguiva meccanicamente addizione e sottrazione, vede la luce nel 1971 per opera di Niklaus Wirth, un professore del politecnico di Zurigo. Il Pascal si caratterizza per la chiarezza e la precisione con cui è possibile definire i tipi di dati e controllare la sequenza delle istruzioni. È particolarmente adatto alla didattica per l'insegnamento della programmazione strutturata.

Nel 1972 Dennis Ritchie progettava e realizzava la prima versione del linguaggio C presso i Bell Laboratories. Ritchie e Ken Thompson, l'autore del sistema operativo Unix, riscrissero in C il codice di Unix. Il linguaggio C pur essendo un linguaggio di alto livello, permette operazioni tipiche del linguaggio macchina: si può, per esempio, indirizzare la memoria in modo assoluto, funzionalità fondamentale per lo sviluppo di applicazioni di basso livello. È un linguaggio apparentemente povero: non possiede istruzioni di entrata/uscita né istruzioni per operazioni matematiche. Ogni funzione diversa dai costrutti di controllo o dalle operazioni elementari sui tipi dati è affidata a un insieme di librerie esterne. In questo modo, Ritchie riuscì a raggiungere due obiettivi: da una parte mantenere compatto il linguaggio, dall'altra poter estenderne le funzionalità semplicemente aggiungendo nuove librerie o ampliando quelle esistenti. Il C è stato talvolta definito come "il linguaggio di più basso livello tra i linguaggi di alto livello". In effetti nasce per lo sviluppo di sistemi operativi, quindi per software di basso livello, ma preservando la semplicità d'uso dei linguaggi della terza generazione. I nuovi linguaggi che si sono presentati sulla scena dell'informatica, quali C++, Java e Perl, gli devono molto.

In Java, come in C, l'istruzione di somma vista nei paragrafi precedenti potrebbe diventare semplicemente:

```
a = a+b;
```

dove con `a` e `b` non stiamo facendo diretto riferimento a locazioni di memoria né a registri della CPU, ma a variabili identificate da un nome e alle quali possono essere assegnati dei valori.

A ogni specifica variabile presente nell'istruzione del linguaggio ad alto livello, con la traduzione in linguaggio macchina e il successivo caricamento in memoria del codice oggetto per l'esecuzione, sono fatte corrispondere le celle di memoria necessarie a ospitarne i valori. Il programmatore Java manipola comodamente delle variabili ciascuna identificata da un nome, mentre l'elaboratore lavora su spazi di memoria identificati da indirizzi di locazioni fisiche.

Una *variabile*, oltre al nome che la identifica, è caratterizzata dal *tipo* di appartenenza che ne definisce l'insieme dei valori che può assumere, come vedremo dettagliatamente a partire dal prossimo capitolo. Per adesso diciamo che le variabili con nome `a` e `b` sono di tipo intero e quindi possono assumere solo valori interi.

```
a = 6;
b = 31;
```

La prima istruzione – in Java e in molti altri linguaggi ogni istruzione termina con il carattere di punto e virgola – *assegna* ad a il valore 6: inserisce il valore 6 che segue l'operatore = nella variabile a che lo precede. Analogamente, la seconda istruzione assegna a b il valore 31:

L'istruzione a=a+b assegna alla variabile a la somma dei valori contenuti nelle variabili a e b:

```
a = a+b;
```

Dopo la sua esecuzione, a conterrà 37:

Uno dei vantaggi dei linguaggi evoluti è che il programma diventa portabile su ogni macchina che abbia un compilatore per il linguaggio che si sta utilizzando; un altro vantaggio è che non ci si deve affatto preoccupare degli indirizzi di memoria. Di più: se vogliamo ottenere l'area di un rettangolo possiamo scrivere

```
area = base*altezza;
```

utilizzando per le variabili nomi mnemonici e rendendo così il programma più intuitivo (il simbolo * indica l'operatore di moltiplicazione). Riprendendo l'algoritmo per determinare se un numero è pari o dispari, potremmo scrivere:

```
prendi numero;
resto = numero % 2;
se resto è zero scrivi "pari" altrimenti scrivi "dispari"
```

Dove numero e resto sono variabili e l'operatore di modulo % consente di ottenere il resto della divisione intera tra l'operando che lo precede e quello che lo segue.

✔ NOTA

Con il termine file *(archivio) viene indicato un qualsiasi insieme di dati memorizzato dal computer su una memoria di massa, per esempio un hard disk. Può quindi essere memorizzato in un file un programma sorgente, in modo da poterlo modificare in seguito; oppure il suo codice oggetto, così da poterlo poi eseguire direttamente quando lo si desidera e senza dover compilare il programma originario. I file possono contenere testi, immagini, suoni, filmati in formato digitale e anche insiemi di dati strutturati, per esempio i nominativi degli studenti di una classe o i valori contabili di una società.*

2.4 Compilatori e interpreti

I programmi per essere eseguiti devono essere scritti in linguaggio macchina (codice oggetto), per rendere il processo di trasformazione automatico sono disponibili appositi *programmi traduttori*. Un programma assembler è tradotto in linguaggio da programmi detti *assemblatori*.

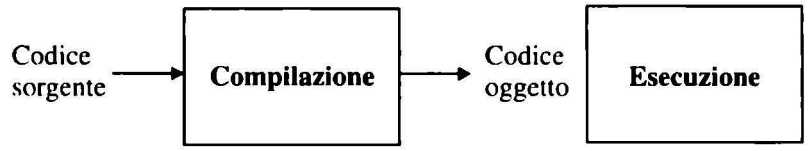
Codice mnemonico (assembler) → Assemblatore → Codice oggetto (eseguibile)

Un programma in un linguaggio di alto livello (codice sorgente) è tradotto in linguaggio macchina dai programmi detti *compilatori*.

Codice sorgente → Compilatore → Codice oggetto (eseguibile)

Una volta prodotto il codice oggetto il programma può essere eseguito.

In molti linguaggi, come C, C++, Fortran e Cobol, le due fasi di compilazione ed esecuzione sono distinte. Prima è generato il codice oggetto dal compilatore, poi il programma può essere caricato in memoria ed eseguito quante volte si desidera.



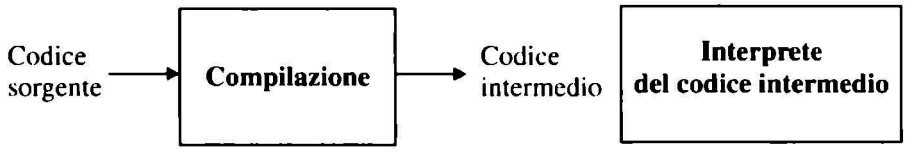
Nella messa a punto del programma è necessario ripetere le due fasi, in modo da verificare l'effetto delle modifiche apportate al codice. Successivamente il programma è utilizzato ed eseguito direttamente in codice oggetto. Una nuova traduzione è necessaria soltanto se sono effettuate modifiche al programma originario. In questo caso si parla di *linguaggio compilato*.

Alcuni linguaggi, come il BASIC, seguono un principio di funzionamento diverso: in pratica il codice sorgente viene letto direttamente da un programma apposito, detto *interprete*, che comprende le istruzioni del programma e si occupa di eseguirle tramite opportune istruzioni alla CPU: in questo caso si parla di *linguaggi interpretati*.

L'interprete dunque fa le veci di una CPU virtuale in grado di capire direttamente il codice sorgente, sollevando il programmatore dal compito di dover eseguire la fase di compilazione. Un altro vantaggio di questo tipo di linguaggio è che l'interprete isola i programmi dal sistema operativo sottostante, permettendo ai programmi di poter funzionare su sistemi operativi diversi risentendo meno delle caratteristiche peculiari di ciascuno di essi.

Naturalmente i linguaggi interpretati presentano anche degli svantaggi, primo fra tutti il fatto di essere meno efficienti: la fase di interpretazione dei sorgenti infatti è un processo relativamente laborioso che viene eseguito ogni volta che il programma viene eseguito e che è invece del tutto assente nei linguaggi compilati. Un altro svantaggio rispetto ai linguaggi compilati è che questi ultimi durante la compilazione vengono controllati globalmente, per cui eventuali errori formali vengono individuati prima dell'esecuzione mentre in un linguaggio interpretato se un'istruzione non è sintatticamente corretta, l'errore viene normalmente rilevato solo quando l'interprete prova a eseguirla.

Per ovviare agli svantaggi dei due approcci, alcuni linguaggi interpretati come Java prevedono una fase di compilazione che produce un codice intermedio, formalmente verificato e più facilmente e rapidamente interpretabile.

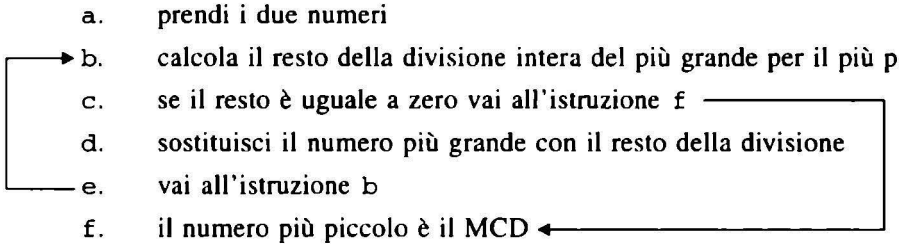


Il codice intermedio di Java è chiamato *bytecode* perché ciascuna istruzione è composta da un singolo byte, in modo simile a quanto si è visto per il linguaggio macchina. La Sun, che ha "inventato" Java, ha perfino realizzato dei microprocessori che eseguono direttamente il bytecode di Java. Dunque si può pensare all'interprete di Java come a una CPU virtuale capace di eseguire il bytecode di Java, ed è per questo motivo che ci si riferisce a esso col termine *Java Virtual Machine (JVM)*.

Un programma Java quindi può essere eseguito su qualsiasi computer a patto che su di esso sia presente una JVM e oggi esistono JVM praticamente per tutti i sistemi operativi: da qui il motto lanciato da Sun "Write once, run everywhere" ("scrivi una volta, esegui dovunque").

2.5 Programmi strutturati

Abbiamo visto negli esempi e nella descrizione della macchina di Turing del primo capitolo come un algoritmo sia costituito, oltre che dalle operazioni da applicare ai dati per trasformarli, dalla distinzione dei differenti stati di computazione, dalla capacità di scegliere in base a dei valori come proseguire l'esecuzione ed eventualmente di saltare a un altro stato. Riprendiamo il problema del calcolo del MCD tra due numeri e scriviamo un algoritmo equivalente a quello già visto.



La c è un'istruzione di controllo decisionale: "se il resto è uguale a zero ..." (si vedano i Paragrafi 1.2 e 1.3). In questo caso se il controllo dà esito positivo l'istruzione provoca un *salto* nella sequenza d'esecuzione, che per tale ragione viene detto *salto condizionato*. Il salto è condizionato dal verificarsi di un certo evento in fase di esecuzione: "se il resto è uguale a zero vai all'istruzione `f`", altrimenti il flusso proseguirà alla prossima istruzione della sequenza, la `d`. L'istruzione `e` presenta un *salto incondizionato* ("vai all'istruzione `b`"): in ogni caso il flusso d'esecuzione dopo l'istruzione `e` passerà all'istruzione indicata dal salto, la `b`.

Vediamo il flusso di esecuzione per valori in ingresso 924 e 120.

passo 1	924 e 120
passo 2	84 è il resto della divisione intera di 924 per 120
passo 3	il resto è diverso da zero (prosegui in sequenza)
passo 4	120 e 84
passo 5	vai all'istruzione `b`
passo 6	36 è il resto della divisione intera di 120 per 84
passo 7	il resto è diverso da zero (prosegui in sequenza)
passo 8	84 e 36
passo 9	vai all'istruzione `b`
passo 10	12 è il resto della divisione intera di 84 per 36
passo 11	il resto è diverso da zero (prosegui in sequenza)
passo 12	36 e 12
passo 13	vai all'istruzione `b`
passo 14	0 è il resto della divisione intera di 36 per 12
passo 15	il resto è uguale a zero, vai all'istruzione `f`
passo 16	12 è il MCD

Schematizziamo il flusso dell'algoritmo, così come avevamo fatto per la precedente versione nella Figura 1.1 del Capitolo 1, inserendo all'interno di ogni cerchio, che indicheremo come *nodo*, l'istruzione corrispondente e all'esterno i passi in esecuzione, sempre nel caso in cui i numeri siano 924 e 120 (Figura 2.3).

Oltre alle istruzioni che consentono di selezionare il cammino (se il resto è uguale a zero ...), esistono quelle che permettono di effettuare il salto a un altro nodo. La più famosa di queste in molti linguaggi è l'istruzione *goto* ("vai a") seguita dall'indicazione dell'istruzione/nodo cui saltare, che può essere un'etichetta come quelle usate nel nostro esempio (`b`, `f`) o proprio il numero della riga da cui si desidera far proseguire l'esecuzione. Abbiamo utilizzato questo esempio allo scopo di poter far riferimento con cognizione di causa all'istruzione di salto. In realtà questa libertà di *salto a piacimento* porta rapidamente, affrontando problemi via via più complessi, a programmi contorti, ottenendo l'effetto comunemente conosciuto come codice o programmi a "spaghetti" (Figura 2.4).

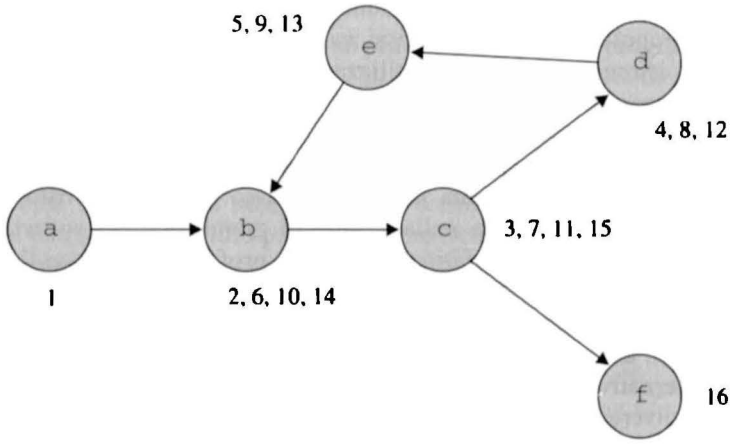


Figura 2.3 Sequenza di stati nel flusso di un algoritmo per il calcolo del MCD

Si parte dal nodo iniziale, eseguendo le operazioni specificate, e si passa al nodo successivo seguendo una delle frecce, scegliendo o meglio *selezionando* il cammino in base allo stato che si è determinato. A ogni nodo si eseguono le operazioni e si seleziona il successivo, finché si trova l'istruzione di termine del calcolo. Potremo quindi avere continui spostamenti, cicli e ricicli tra i nodi.

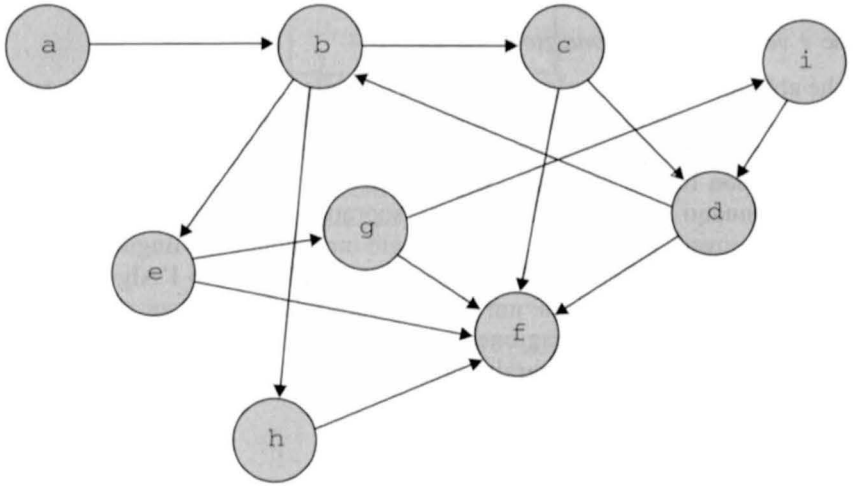


Figura 2.4 Sequenza di stati nel flusso di esecuzione di un programma non strutturato

Per anni questo approccio fu considerato corretto e forse l'unico possibile: il vantaggio evidente era di poter eseguire a piacimento parti di programma saltando in avanti e indietro senza regola. Gli sviluppatori si vantavano di scrivere programmi con il minor numero di istruzioni possibili, riutilizzando parti di codice senza preoccuparsi che questo significasse inserire continuamente dei salti tra le istruzioni. Si è poi verificato senza ombra di dubbio come questo modo di procedere porti a programmi difficili da correggere, manutenere ed estendere. Il tutto si aggrava nei casi in cui lo sviluppatore chiamato a modificare il programma non sia lo stesso che lo ha scritto, o in cui il lavoro è di gruppo, come succede nella stesura di grandi progetti software. Questo modo di pensare con il tempo e a fatica è cambiato profondamente con l'arrivo della *programmazione strutturata*. Obiettivo di quest'ultima è di rendere un flusso ordinato il passaggio tra le istruzioni dall'inizio alla fine dei programmi. L'ideale sarebbe di poter ridurre tutto a una sequenza di operazioni da eseguire una dopo l'altra in modo lineare, senza alternative possibili, ma ciò limiterebbe drasticamente la potenza degli algoritmi. Per risolvere il problema sono state definite delle regole, che del resto vanno incontro al nostro stesso modo di pensare in linguaggio naturale.

Nella programmazione strutturata è possibile utilizzare oltre alla *sequenza*:

fai *questo*
fai *quello*

la *selezione* tramite le *strutture di controllo decisionali*:

se *è verificata una condizione* fai *questo*

prevedendo anche il caso di *alternativa* doppia che abbiamo usato nella terza istruzione dell'esempio per determinare se il numero intero era pari o dispari:

se *è verificata una condizione* fai *questo altrimenti* fai *quello*

e le *ripetizioni* cicliche tramite le *strutture di controllo iterative*:

finché *è verificata una condizione* fai *questo*

Quelle che abbiamo citato sono le strutture di *controllo del flusso*: sequenza, selezione e ripetizione, che andremo di seguito ad approfondire e che tratteremo in tutto il testo. Avendo a disposizione queste strutture, come avviene nei moderni linguaggi di alto livello, il *goto* non ha più senso di esistere e in Java non è presente, anche se altri linguaggi continuano a renderlo disponibile soprattutto per ragioni di compatibilità con il passato; è invece necessario nel linguaggio macchina e nei linguaggi assembler.

Il primo linguaggio che seguiva il pensiero naturale è stato l'Algol che per tale ragione fu adottato all'epoca nelle università d'informatica e da cui i suoi successori hanno ereditato molto. Per una ragione evidente, i linguaggi di alto livello utilizzano termini inglesi del linguaggio naturale; per esempio, le strutture di controllo appena viste spesso sono nella forma:

`if` *è verificata una condizione* fai *questo*

`if` *è verificata una condizione* fai *questo* `else` fai *quello*

`while` *è verificata una condizione* fai *questo*

2.6 Sequenza, selezione, iterazione

Con la tecnica strutturata i programmi sono più chiari (rispetto ai programmi spaghetti), facili da mettere a punto e modificare e con maggiori probabilità di essere esenti da errori. Rimane una domanda: l'utilizzo di sequenza, selezione e iterazione senza l'uso del salto *goto* dà la stessa potenza di calcolo oppure si perde qualcosa? La risposta è venuta nel 1966 da due studiosi italiani, Corrado Böhm e Giuseppe Jacopini, che hanno dimostrato come i programmi possano essere scritti senza usare nemmeno un'istruzione *goto*. Il teorema che da loro prende il nome (teorema di Jacopini-Böhm) afferma che tutti i programmi possono essere scritti in termini di due sole strutture di controllo del flusso, oltre alla sequenza: la selezione e l'iterazione.

La Figura 2.5 mostra un esempio di *diagramma di flusso* a struttura sequenziale.

✔ **NOTA**

Nella stesura degli algoritmi, invece di andare a codificare il programma direttamente nel linguaggio di programmazione, spesso si preferisce utilizzare una pseudocodifica, che pur non essendo eseguibile dall'elaboratore permette di rendere meglio quel che desideriamo esprimere. La pseudocodifica di un algoritmo, se pensata con cura, consentirà successivamente di tradurre rapidamente il programma in un linguaggio di programmazione vero e proprio.

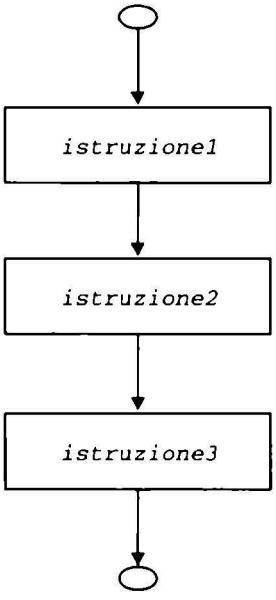


Figura 2.5 Diagramma di flusso con una struttura di controllo del flusso sequenziale

Come la pseudocodifica, anche i diagrammi di flusso servono a rappresentare algoritmi. Per esempio, mostrano con immediatezza la funzione delle strutture di controllo, e noi li utilizzeremo esclusivamente a tale scopo. I blocchi rettangolari che hanno una o più frecce in ingresso da cui proviene il flusso e una sola in uscita dove prosegue il flusso d'esecuzione contengono istruzioni imperative: leggi, addiziona, visualizza. *I blocchi romboidali che hanno una o più frecce in ingresso e una sola in uscita contengono una* condizione logica *che, a seconda se è verificata o meno, farà proseguire il flusso d'esecuzione su una o l'altra delle due frecce in uscita. I circoli li abbiamo utilizzati per indicare da dove parte e dove finisce il flusso di esecuzione. Nel caso di un intero algoritmo rappresentato con un diagramma di flusso, il primo circolo dovrebbe contenere la parola* INIZIO *e l'ultimo la parola* FINE.

La struttura di sequenza è implicita in Java: sempre che non venga indicato diversamente vengono eseguite in sequenza le istruzioni nell'ordine in cui appaiono. Il Capitolo 3 introduce proprio questo argomento.

Per la selezione abbiamo le strutture di controllo decisionali (Figura 2.6)

```
if(espressione-logica) istruzione
```

In esse si ha la valutazione di una condizione, `espressione-logica`, che potremo indicare anche come `espressione-booleana`. Se questa è vera si esegue `istruzione`; successivamente il flusso di esecuzione proseguirà con la prossima istruzione in sequenza. Un'*espressione logica*, come vedremo precisamente a partire dal Capitolo 4, in Java può restituire solamente uno dei due *valori booleani*: *vero* (*true*) o *falso* (*false*). Un esempio di espressione è "il resto della divisione è uguale a zero?":

```
prendi numero;
resto = numero % 2;
if(resto==0) scrivi "è pari";
```

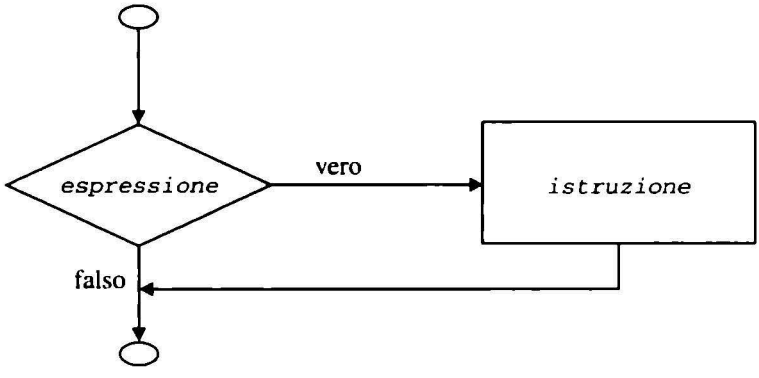


Figura 2.6 Diagramma di flusso con una struttura di controllo del flusso decisionale

L'operatore == ha il significato di *uguale a*: restituisce vero se l'operando che lo precede e quello che lo segue sono uguali, falso altrimenti. Nelle esemplificazioni utilizziamo il corsivo courier per indicare quelle parti che devono essere sostituite con elementi che le specifichino maggiormente: nell'ultimo esempio `resto==0` ha sostituito *espressione-logica*.

In altri linguaggi come il Pascal il costrutto è più esplicativo:

```
if espressione-logica then istruzione
```

con aggiunta dopo la condizione di `then` ("allora").

Vediamo un altro esempio di struttura di controllo decisionale, un'alternativa che possiamo definire *doppia* (Figura 2.7):

```
if(espressione-logica) istruzione1 else istruzione2
```

Qui si ha la valutazione di una condizione, *espressione-logica*; se questa è vera si esegue *istruzione1*, altrimenti si esegue *istruzione2*. Riprendendo l'esempio per determinare se un numero è pari o dispari, avremmo:

```
prendi numero;
resto = numero % 2;
if(resto==0) scrivi "è pari" else scrivi "è dispari";
```

Il Capitolo 4 introduce i costrutti decisionali e gli operatori del linguaggio Java. Vediamo ora le strutture di controllo iterative (Figura 2.8):

```
while(espressione-logica) istruzione
```

Con questo costrutto viene verificato che *espressione-logica* sia vera, nel qual caso viene eseguita *istruzione*. Il ciclo si ripete fintantoché *espressione* risulta essere vera.

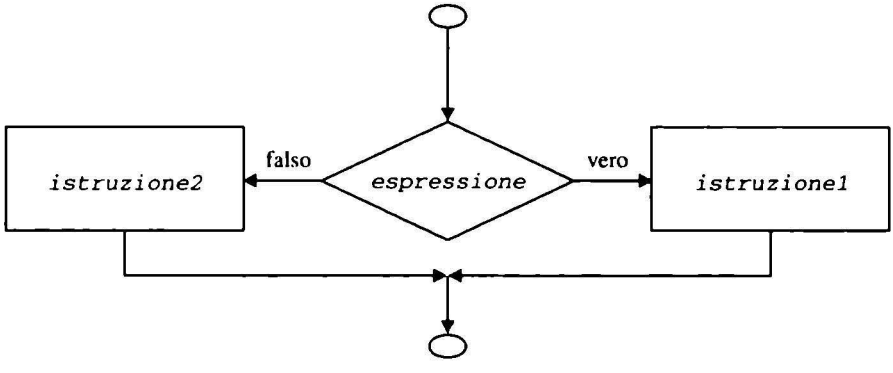


Figura 2.7 Diagramma di flusso con una struttura di controllo del flusso alternativa doppia

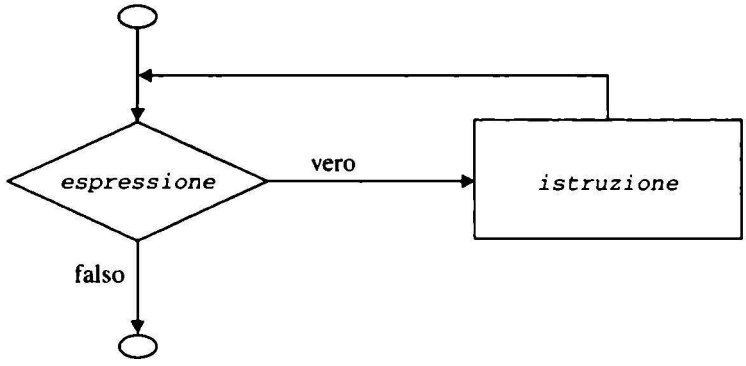


Figura 2.8 Diagramma di flusso con una struttura di controllo del flusso iterativa

Vediamo una iterazione che fa assumere a `i` valori interi da 1 a 100:

```
i=0;
while(i<100) i=i+1;
```

L'operatore < ha il significato di *minore di*: restituisce vero se l'operando che lo precede è minore di quello che lo segue, falso altrimenti.

Esiste anche una struttura di controllo iterativa in cui si ha la possibilità di eseguire una prima volta *istruzione* prima di valutare *espressione* (Figura 2.9):

```
do
  istruzione
while(espressione-logica);
```

Con essa viene eseguita *istruzione* e successivamente viene controllato se *espressione-logica* risulta essere vera, nel qual caso il ciclo è ripetuto. Riprendiamo per esempio l'iterazione che fa assumere a `i` valori interi da 1 a 100:

```
i=0;
do
  i=i+1;
while(i<100);
```

Con il costrutto `do-while` il controllo avviene dopo l'esecuzione dell'istruzione: `i=i+1`. Con il costrutto `while` può succedere che *istruzione* non venga mai eseguita (è il caso in cui *espressione-logica* già la prima volta risulti falsa), mentre, con il costrutto `do-while`, *istruzione* viene sempre eseguita almeno una volta.

In realtà spesso gli sviluppatori utilizzano di più il costrutto iterativo `for`, rispetto al `while` e al `do-while`. Introdurremo tutte le strutture di controllo iterative con esempi anche comparativi nel Capitolo 5.

Abbiamo proposto alcune possibilità. Ce ne sono altre. L'importante è che tutte fanno riferimento a tre uniche strutture: sequenza, selezione, iterazione. Ci manca però un ultimo elemento non trascurabile della programmazione strutturata: i *blocchi*.

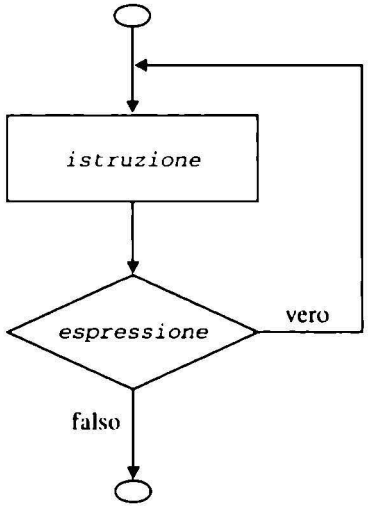


Figura 2.9 Diagramma di flusso con una seconda struttura di controllo del flusso iterativa

Abbiamo aperto i diagrammi di flusso con un piccolo cerchio, ugualmente li abbiamo chiusi con un ultimo cerchio: tutti i canali partono da quel primo cerchio e terminano nell'ultimo. In altre parole, c'è un solo punto d'inizio e un solo punto di fine. Un programma può essere composto da più strutture di controllo, e i cerchi sono i connettori, i punti in cui diverse strutture possono essere collegate le une con le altre. Non è possibile collegare due strutture utilizzando altri punti di attacco che non siano quelli indicati. In questo modo tutti i diagrammi che potremo comporre avranno una composizione lineare con un solo inizio e una sola fine.

2.7 Blocco d'istruzioni

Se mettiamo insieme più strutture formiamo un *blocco d'istruzioni*, come abbiamo implicitamente fatto in Figura 2.5 per visualizzare la sequenza, che è appunto un insieme d'istruzioni che ha la caratteristica di avere un solo inizio, da dove inizia sempre l'elaborazione, e una sola uscita, dove termina sempre l'elaborazione del blocco (Figura 2.10).

Se ci fossero delle uscite laterali tramite delle istruzioni di salto, non si tratterebbe più di un blocco e non verrebbero rispettati i canoni della programmazione strutturata. Dunque il blocco ha una propria autonomia e può essere visto come una *scatola nera* utilizzata per eseguire un determinato compito, mentre non possono essere utilizzate singolarmente delle sue sottoparti. Si tratta di una caratteristica della programmazione strutturata da sottolineare: il "tubo" che porta dall'inizio alla fine del blocco è a tenuta stagna, una sola entrata e una sola uscita, nessuna infiltrazione o perdita.

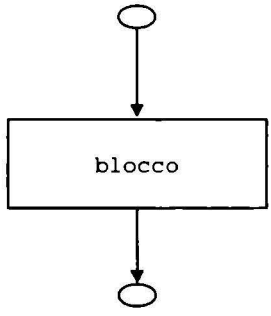
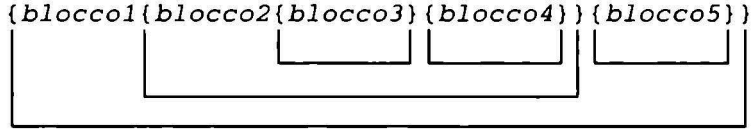


Figura 2.10 Rappresentazione di un blocco d'istruzioni

In Java i blocchi iniziano con una parentesi graffa aperta e finiscono con una parentesi graffa chiusa:

{blocco}

Le caratteristiche che abbiamo enunciato fanno sì che i blocchi possono essere contenuti uno nell'altro ma non possono mai intersecarsi, esattamente quello che succede con le parentesi nelle espressioni matematiche, la prima parentesi chiusa chiude l'ultima parentesi aperta.



Riprendiamo l'algoritmo per il calcolo del MCD esaminato nel Capitolo 1 e trasformiamolo in un codice "quasi Java", utilizzando quanto abbiamo introdotto fino a questo punto. Avremo le variabili `grande`, `piccolo` e `resto`, l'operatore `%` per trovare il resto della divisione intera tra `grande` e `piccolo`, l'operatore = per assegnare il risultato dell'operazione a `resto` e successivamente spostare il valore di `piccolo` in `grande` e di `resto` in `piccolo`. La struttura `do-while` ci permetterà di ripetere il ciclo fino a che `resto` non assumerà valore zero:

```
prendi grande e piccolo;
do {
  resto = grande % piccolo;
  grande = piccolo;
  piccolo = resto;
}
while(resto!=0);
scrivi grande "è il MCD";
```

L'operatore != significa *diverso da*: restituirà vero quando resto sarà diverso da zero, falso altrimenti. Quando le istruzioni formano un blocco devono essere racchiuse tra parentesi graffe:

```
{
  resto = grande % piccolo;
  grande = piccolo;
  piccolo = resto;
}
```

In questo modo la struttura di controllo iterativa do-while a ogni ciclo ripeterà in sequenza le istruzioni comprese nel blocco. Seguiamo l'esecuzione delle sole istruzioni operative osservando l'effetto che hanno sulle variabili. La prima istruzione *prendi* costituisce l'ingresso dei valori iniziali nelle variabili grande e piccolo. Supponiamo che tali valori siano 924 e 120 (Figura 2.11). L'ultima istruzione, *scrivi*, rappresenta l'uscita del risultato verso l'esterno e visualizzerà o stamperà il valore della variabile grande, seguito da "è il MCD":

```
12 è il MCD
```

Si deve avere l'accortezza di dare inizialmente in pasto al programma il più grande dei due numeri, poi il più piccolo, in modo che vadano a riempire rispettivamente le variabili grande e piccolo. Per fare in modo che ciò non sia necessario dobbiamo aggiungere all'inizio del programma la possibilità di scambiare i valori:

```
prendi grande e piccolo;
if(grande<piccolo) {
  aus = grande;
  grande = piccolo;
  piccolo = aus;
}
do {
  resto = grande % piccolo;
  grande = piccolo;
  piccolo = resto;
}
while(resto!=0);
scrivi grande "è il MCD";
```

Con l'assegnamento a grande del valore di piccolo il precedente valore di grande viene perduto, per cui per poterlo correttamente assegnare a piccolo tale valore deve essere stato salvato in una variabile di appoggio che abbiamo chiamato aus (Figura 2.12).

In una nota nel Capitolo 5, dopo che avremo potuto sperimentare sequenze, diramazioni e iterazioni, nonché l'utilizzo dei blocchi e delle variabili, torneremo brevemente sul teorema di Jacopini-Böhm.

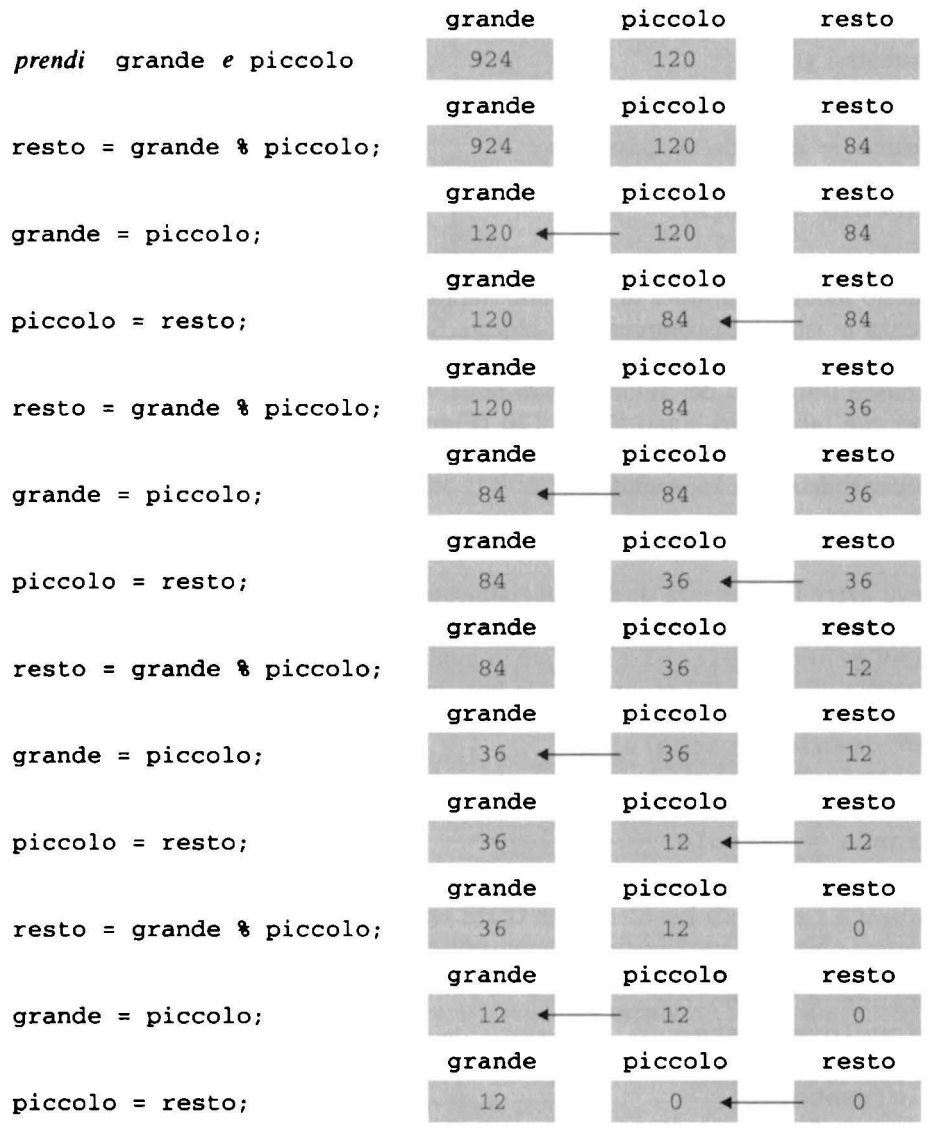


Figura 2.11 Valori delle variabili durante l'esecuzione dell'algoritmo per determinare il MCD

2.8 Approccio top-down e bottom-up

La metodologia *top-down* parte dall'alto considerando l'intero problema da risolvere e procede a cascata verso il basso fino a ottenere un insieme di sotto-problemi elementari. Infatti per affrontare un problema complesso il modo migliore è di scomporlo

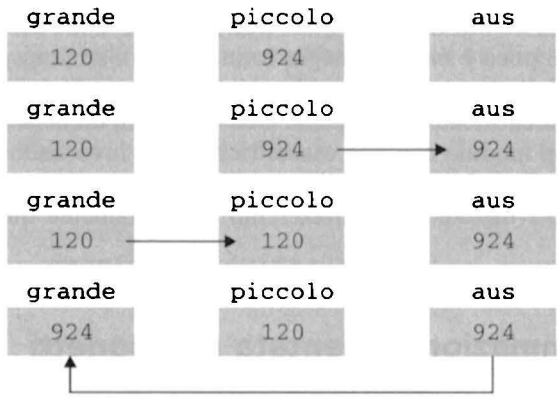


Figura 2.12 Scambio dei valori di due variabili

in più problemi e affrontare ciascuno di essi. Ognuno di questi sotto-problemi sarà più semplice e potrà essere ulteriormente suddiviso fino ad arrivare a risolvere facilmente ognuno dei sotto-problemi identificati. A ognuno di questi si fa corrispondere un modulo applicativo, sotto-programma, che lo risolve.

L'approccio complementare a questo è detto *bottom-up* ("dal basso verso l'alto"), in cui ci si occupa prima di risolvere singole parti del problema, senza averne necessariamente una visione d'insieme, per poi risalire procedendo per aggiustamenti e componendo la soluzione stessa. Nei progetti software i due approcci possono convivere; per esempio si può utilizzare come linea guida la metodologia top-down, mentre in casi particolari parte dei progettisti si mettono a lavorare allo sviluppo di programmi che risolvono specifici aspetti del problema particolarmente spinosi.

2.9 Programmazione modulare

Nella *programmazione modulare* il progettista software affida a ogni *modulo*, che possiamo pensare come un blocco d'istruzioni, un determinato compito, per esempio ordinare una sequenza di numeri. Tale modulo può essere riutilizzato ogni volta che sia necessario svolgere quel compito. Realizzato il modulo per ordinare dei numeri, ogni volta che avremo tale necessità, il nostro lavoro si ridurrà a chiamare l'apposito modulo e applicarlo ai valori d'interesse. Il concetto di *riusabilità* è molto importante nella progettazione del software, perché fa risparmiare una grande quantità di tempo e migliora la qualità dei sistemi.

La programmazione modulare appare a prima vista piuttosto lineare, ma non è proprio così. I moduli per risolvere l'intero problema devono gioco forza interagire gli uni con gli altri, il che aggiunge complessità alla soluzione. Per questa ragione e anche per poterli riutilizzare efficacemente nella soluzione di problemi differenti, i

moduli devono essere progettati in modo da avere validità in se stessi, il più possibile indipendenti e scambiare informazioni l'uno con l'altro in modo semplice e chiaro. Se un modulo fa troppo poco è meglio integrarlo in un altro, se fa troppo è meglio suddividerlo in più moduli. A questo scopo i linguaggi strutturati mettono a disposizione il *sottoprogramma*, identificato da un nome e costituito da un insieme di istruzioni. Il sottoprogramma è un modulo che può essere richiamato invocandone il nome quante volte lo si desidera. Introdurremo esplicitamente la programmazione modulare nel Capitolo 7 trattando i metodi, ma utilizzeremo implicitamente questo approccio in tutto il testo.

2.10 Programmazione orientata agli oggetti

La programmazione strutturata offre un metodo di descrizione degli algoritmi molto valido, ma non affronta un problema che è molto comune nello sviluppo di programmi reali. Difficilmente si potrà trovare un modo più comprensibile della programmazione strutturata per descrivere l'algoritmo appena visto per il calcolo del MCD, ma i programmi reali normalmente non trattano solo numeri, ma insiemi d'informazioni che hanno essi stessi una struttura più o meno complessa.

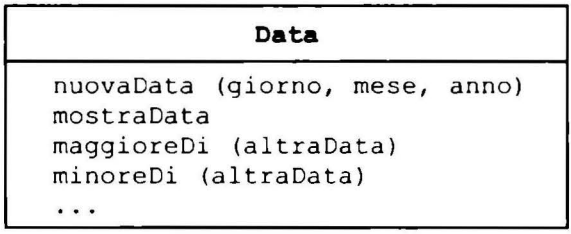
Per esemplificare il problema supponiamo di dover sviluppare un programma che esegue delle elaborazioni con le date di nascita di una o più persone. Una data può essere rappresentata in modi diversi: il più immediato è quello di usare tre numeri, uno per l'anno, uno per il mese e uno per il giorno, ma si possono usare altre rappresentazioni usando, per esempio, solo due numeri, uno per l'anno e il secondo che indica il numero di giorni passati dal capodanno, o ancora si può usare anche un solo numero che definisce o il numero di giorni a partire da una data fissata o il numero di secondi (o millisecondi) a partire da un 'momento' prefissato.

Normalmente non si fanno calcoli sulle date, ma alcune volte è possibile che qualche piccola elaborazione si renda necessaria come, per esempio, sapere se una data è posteriore a un'altra, calcolare quanti giorni intercorrono tra due date e così via.

Per fissare le idee, supponiamo di dover sviluppare un programma in cui risulta spesso necessario confrontare delle date e che sia deciso che le date siano memorizzate usando tre numeri interi, il che ci costringe a eseguire un seppur piccolo algoritmo per fare i confronti. Ripetere un algoritmo più volte, oltre che essere un evidente spreco di tempo (del programmatore) e memoria (del computer), ha le seguenti controindicazioni:

- se l'algoritmo contiene un errore, magari che si presenta solo in casi poco frequenti, è più facile individuarlo se esso compare una volta sola; una volta individuato il problema, la correzione vale per tutto il programma.
- se un giorno, per qualche motivo, si decide di rappresentare una data invece che con 3 numeri interi (anno, mese e giorno) con un singolo numero intero corrispondente al numero di giorni trascorsi dal 1/1/1900, l'intero programma dovrebbe essere rivisto per modificare tutti i confronti tra date.

Ecco quindi che per il programmatore diventerebbe comodo poter usare le date in modo analogo a quanto fa con i numeri, senza preoccuparsi cioè di come sono memorizzati i dati, ma conoscendo solo le operazioni possibili su di essi. Una data allora può essere schematizzata come segue:

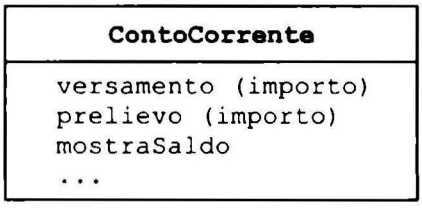


La data è diventata così un *tipo astratto* o *classe di oggetti*, cioè un'associazione tra dati, normalmente invisibili all'esterno, e operazioni (o *metodi*) che agiscono su di essi: il programmatore agisce sui dati solo attraverso le operazioni disponibili, affidando a esse l'implementazione degli algoritmi necessari e la garanzia della congruenza dei dati. In questo modo si possono trattare le date nello stesso modo semplice con cui vengono trattati i numeri e se ne può centralizzare la gestione evitando tutti i problemi anzidetti. Ogni volta che il programmatore ha bisogno di una data, crea un'*istanza* della classe (o *oggetto*), vale a dire un'area di memoria organizzata secondo i criteri descritti nella classe e sulla quale si può operare tramite i metodi implementati nella classe stessa.

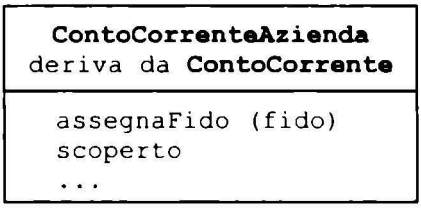
Nella precedente schematizzazione semplificata, accanto al nome dell'operazione sono stati riportati tra parentesi tonda gli eventuali operandi aggiuntivi (*argomenti*) necessari alla corretta esecuzione dell'operazione. Ecco quindi che il confronto tra due date si può esprimere nel modo seguente, dove `unaData` e `unAltraData` sono di tipo `Data`:

```
Data unaData, unAltraData;
.
.
  if (unaData.maggioreDi (unAltraData)) ...
.
.
```

Questo approccio può essere esteso dalle strutture dati più semplici, come la data, a strutture via via più complesse. Un intero programma può quindi essere un'istanza di una classe che contiene al proprio interno altri oggetti, e così via in una struttura gerarchica. Per esempio, un programma di gestione bancaria potrebbe avere una classe `ContoCorrente` fatta nel modo seguente:



Le banche però hanno molti tipi di conto corrente, sempre più specializzati secondo le caratteristiche della clientela. Fare una classe per ogni tipo comporterebbe una duplicazione di algoritmi per tutte le operazioni comuni a tutti i tipi; d'altro canto fare un'unica classe con tutte le tipologie possibili la renderebbe molto complessa, difficile da manutenere e quindi poco adatta a futuri adattamenti. Per superare questo genere di problemi è stato allora introdotto il concetto di *ereditarietà*, la possibilità cioè di fare in modo che una classe sia del tutto equivalente a un'altra (detta *classe genitrice*) ma abbia dei dati e delle operazioni in più per affrontare meglio problemi particolari. Nel nostro esempio allora potremmo avere:



La classe `ContoCorrenteAzienda` viene definita *sottoclasse* di `ContoCorrente` e i suoi oggetti possono essere usati come gli oggetti della sua classe genitrice, ma permettono di fare anche altre operazioni più specializzate. Se necessario è possibile specializzare ulteriormente una classe in sottoclassi senza limite.

La programmazione orientata agli oggetti modella più facilmente la realtà, permettendo di comprendere e risolvere più semplicemente i problemi. La sua potenza risiede soprattutto nel fatto che si adatta particolarmente bene al modo di pensare degli esseri umani. Per tale ragione si è diffusa ed è destinata a essere utilizzata sempre più ampiamente. Cominceremo a lavorare in modo consapevole con un approccio orientato agli oggetti a partire dal Capitolo 8.

Domande di verifica

1. Fare degli esempi di frasi in lingua italiana che contengano delle ambiguità.
2. Rileggere con attenzione le proprie soluzioni date agli esercizi del Capitolo 1 e verificare se non siano presenti delle istruzioni che possono essere interpretate in modo ambiguo da un'altra persona. [Quando siamo conviti che non siano presenti ambiguità possiamo effettuare un'ultima verifica, dando il foglio di carta con l'algoritmo a un amico e chiedendogli di eseguire le istruzioni senza fantasia e senza nostri consigli.] Nel caso, riscrivere tali frasi in modo da renderle maggiormente precise.
3. Cos'è un linguaggio di programmazione?
4. Quali sono le differenze tra linguaggio macchina e assembler?
5. Cosa caratterizza i linguaggi di alto livello?
6. Quale compito svolgono i programmi detti assemblatori? E i compilatori? E gli interpreti?

7. Riflettere sulla seguente definizione di "potenza": "la capacità di calcolare un certo insieme di funzioni calcolabili". Quali sono i computer più potenti e quali quelli meno potenti? [Si noti che questa definizione non ha a che vedere con la velocità di calcolo o il numero di istruzioni al secondo.]
8. Cosa asserisce il teorema di Jacopini-Böhm?
9. Descrivere compiutamente cosa si intende per programmazione strutturata.
10. Fare degli esempi di strutture di controllo decisionali e iterative.
11. Cosa si intende per blocco d'istruzioni?
12. Cosa si intende per approccio top-down riferendosi alla risoluzione dei problemi?
13. Qual è un vantaggio della programmazione modulare?
14. Dare una prima descrizione del significato di "programmazione orientata agli oggetti".

Esercizi

Descrivere, utilizzando strutture di controllo e variabili, le istruzioni necessarie per realizzare quanto specificato dai seguenti esercizi, molti dei quali avevamo già affrontato nel precedente capitolo.

1. Verificare se un numero è divisibile per 3.
2. Scrivere la successione dei numeri interi positivi da 1 a 100.
3. Scrivere la successione dei numeri interi positivi dispari da 1 a 99.
4. Scrivere la successione dei numeri negativi pari da -2 a -50.
5. Scrivere la successione dei numeri negativi partendo da -150 e arrivando a -1, escluso i numeri divisibili per 4.
6. Verificare se un numero è divisibile per 5 e per 30.
7. Vincere giocando per primi al gioco degli undici. Si gioca in due con undici oggetti sul tavolo, per esempio stecchini da denti. Ciascun giocatore, al proprio turno, può raccogliere 1, 2 o 3 oggetti. Perde chi raccoglie l'ultimo oggetto. [Il segreto sta nel lasciare all'avversario con la prima mossa un numero di stecchini della forma $4i+1$, dove i è un numero intero.]
8. Se nel gioco degli undici (Esercizio 7) il primo giocatore non raccoglie due oggetti nella prima mossa, è possibile realizzare un algoritmo che permetta al secondo giocatore di vincere sempre?
9. L'algoritmo che risolve l'Esercizio 7 può essere generalizzato in modo che il primo giocatore risulti vincente qualsiasi sia il numero di oggetti con cui si gioca? [Si consideri il suggerimento dato all'Esercizio 7.] In caso negativo, indicare quando è possibile e scrivere un algoritmo che risolve il problema.

10. Calcolare le radici di un'equazione di secondo grado $ax^2 + bx + c = 0$ dati i coefficienti a, b e c.

11. Determinare il minimo comune multiplo (mcm) di due numeri interi positivi.

Capitolo 3

Introduzione al linguaggio

Obiettivi didattici

- Primi programmi completi in linguaggio Java
- Visualizzare informazioni sul video (output)
- Variabili e assegnamenti
- Tipi di dati primitivi: interi, a virgola mobile, booleani e caratteri
- Costanti, sequenze di escape
- Primo approccio pratico agli oggetti
- Accettare valori da tastiera (input)

In alcuni linguaggi a oggetti esiste un'unica entità, l'oggetto, e un unico costrutto fornito dal linguaggio, l'assegnamento di un oggetto a un altro oggetto. È una scelta progettuale, propria per esempio di Smalltalk, che li rende molto coerenti dal punto di vista architetturale, ma penalizza le prestazioni e impone una notazione dissimile da quella usata nei linguaggi di programmazione più diffusi. Per evitare questi problemi Java comprende istruzioni e tipi di dati del tutto simili a quelli dei linguaggi strutturati, in particolare del linguaggio C, e su di essi innesta i costrutti per la gestione degli oggetti.

Nei primi capitoli la nostra attenzione sarà dedicata principalmente a tale insieme di istruzioni e tipi di dati; dal Capitolo 8 in poi ci concentreremo sulla programmazione orientata agli oggetti.

3.1 Il primo programma

L'approccio scelto è permettere al lettore di provare direttamente sul computer tutti i concetti illustrati. Iniziamo quindi con un programma che esegue una semplice visualizzazione.

```
class IlProgramma {                                    // linea 1
   public static void main (String argv[]) {           // linea 2
      System.out.println ("Il primo programma");        // linea 3
   }                                                    // linea 4
}                                                       // linea 5
```

Per illustrare compiutamente il programma precedente, sarebbe necessario spiegare in modo astratto un numero notevole di concetti, quali per esempio quelli di classe e metodo, che invece preferiamo introdurre gradualmente vedendone l'utilizzo pratico.

Un programma è composto da un insieme di dichiarazioni di *classi*; la dichiarazione di una classe inizia con la parola riservata `class` seguita da un nome che la identifica e da una parentesi graffa aperta; la corrispondente parentesi graffa chiusa indica il termine della classe. Nel nostro esempio abbiamo la classe `IlProgramma` la cui dichiarazione inizia a linea 1 e termina a linea 5.

Java distingue sempre tra lettere maiuscole e minuscole, per cui la classe `IlProgramma` è totalmente distinta da un'eventuale altra classe `ilprogramma`. Il testo compreso tra le doppie barre `//` e la fine della riga è considerato un commento.

Una classe non può contenere direttamente delle istruzioni, ma può contenere la dichiarazione di *metodi* che a loro volta contengono le istruzioni. Un metodo a sua volta è tipicamente composto da dichiarazioni di variabili e istruzioni eseguibili. La nostra classe contiene la dichiarazione di un solo metodo che inizia a linea 2.

```
public static void main (String argv[]) {          // linea 2
   System.out.println ("Il primo programma");       // linea 3
}                                                   // linea 4
```

Un metodo è identificato da un nome, nel nostro caso `main`, e racchiude tutte le istruzioni e dichiarazioni tra una parentesi graffa aperta e una chiusa, dunque il nostro metodo termina a linea 4. Sulla linea 2 sono presenti prima del nome del metodo alcune parole riservate e dopo, tra parentesi tonde, un parametro d'invocazione di cui parleremo più avanti.

Un metodo può essere attivato solo dall'interno di un altro metodo. Al momento dell'attivazione è possibile fornire al metodo invocato dei parametri racchiusi tra parentesi tonde e separati da virgole e ricevere da questo un valore di ritorno. Poiché un metodo può essere invocato solo dall'interno di un altro metodo, deve essercene uno da cui partire. Java considera il metodo il cui nome è `main`, come quello del nostro esempio, il punto di partenza dell'elaborazione.

La linea 3 contiene l'unica istruzione eseguibile di tutto il programma e consiste nell'invocazione del metodo `println`, contenuto in una classe fornita con il linguaggio, che visualizza il parametro passato tra parentesi tonde seguito da un carattere di ritorno a capo.

```
System.out.println ("Il primo programma");          // linea 3
```

`println` è la contrazione di *print line* (stampa linea); il significato dei termini che precedono il nome separati dai punti verrà illustrato in seguito. Ogni istruzione eseguibile deve essere chiusa con il carattere `;` (punto e virgola).

Il programma deve essere contenuto in un file il cui nome deve corrispondere al nome della classe seguito dal postfisso `.java` e deve essere compilato. Supponendo di disporre dell'ambiente di sviluppo *JDK* o di altri simili, la compilazione si ottiene con il comando `javac` seguito dal nome della classe con estensione.

```
javac IlProgramma.java
```

Il compilatore produce un file il cui nome corrisponde a quello della classe seguito dal postfisso `.class`. Attenzione: il compilatore confronta il nome del file con quello della classe considerando i caratteri maiuscoli e minuscoli anche su sistemi operativi che normalmente non fanno una tale distinzione.

Per eseguire il nostro programma `IlProgramma.class` è necessario richiamare l'interprete, che nell'ambiente JDK o in altri simili è `java`, seguito dal nome del file senza estensione.

```
java IlProgramma
```

Il risultato dell'elaborazione è la visualizzazione della seguente riga:

```
Il primo programma
```

✔ NOTA

Abbiamo fatto riferimento all'ambiente di sviluppo JDK (Java Development Kit) in quanto è il più diffuso, è stato prodotto dalla Sun che ha realizzato e lanciato Java ed è inoltre attualmente distribuito gratuitamente. Gli altri ambienti disponibili sul mercato sono tipicamente sistemi di sviluppo integrati (IDE Integrated Development Environment), il che significa che l'utente interagisce con essi tramite interfaccia grafica a finestre per mezzo del mouse. Con tali strumenti quando si lancia l'esecuzione, viene normalmente aperta una console virtuale che simula il comportamento di un terminale a caratteri. In alcuni di questi ambienti, questa console virtuale viene chiusa appena cessa l'esecuzione del programma, non permettendo di controllare i risultati. In tali casi si consiglia di porre in fondo al programma l'istruzione

```
try {
   System.in.read();
} catch (Exception e) {
}
```

che ferma l'esecuzione fino a che non si preme il tasto Invio.

3.2 Variabili e assegnamenti

Supponiamo di voler calcolare l'area di un rettangolo i cui lati hanno valori interi; il programma potrebbe essere il seguente.

```
class AreaRettangolo {

   public static void main (String argv[]) {
      int base;
      int altezza;
      int area;

      base = 3;
      altezza = 7;
      area = base * altezza;

      System.out.println ("Area = ");
      System.out.println (area);
   }
}
```

Dopo la dichiarazione del metodo `main` e la parentesi graffa aperta, sono presenti le dichiarazioni delle variabili intere necessarie:

```
int base;
int altezza;
int area;
```

La parola chiave `int` specifica che l'identificatore che lo segue si riferisce a una variabile di tipo intero, per cui `base`, `altezza` e `area` sono variabili di questo tipo. Le dichiarazioni, come le istruzioni eseguibili, devono terminare con un punto e virgola. Nel nostro esempio, alla dichiarazione della variabile corrisponde anche la sua *definizione* che fa sì che le sia riservato uno spazio in memoria.

Il nome di una variabile la identifica, il suo tipo ne definisce la dimensione e l'insieme delle operazioni che vi si possono eseguire. In particolare, il tipo `int` definisce una variabile che occupa 32 bit e che può contenere un numero intero compreso tra -2147483648 e 2147483647. Tra le operazioni permesse fra `int` vi sono le quattro operazioni aritmetiche, cioè la somma (+), la sottrazione (-), il prodotto (*) e la divisione (/). L'istruzione

```
base = 3;
```

assegna alla variabile `base` il valore 3; inserisce cioè nello spazio di memoria riservato a tale variabile il valore indicato. Effetto analogo avrà `altezza=7`. L'operatore = realizza dunque l'assegnamento. L'istruzione

```
area = base * altezza;
```

assegna alla variabile `area` il prodotto dei valori di `base` e `altezza`.

Compilando ed eseguendo questo programma, si otterrà il seguente risultato

```
Area =
21
```

Se vogliamo che il risultato sia visualizzato su una sola riga, anziché su due, si può usare il metodo `print` anziché `println`: le due ultime istruzioni del programma diventano così

```
System.out.print ("Area = ");
System.out.println (area);
```

Il programma modificato genera la seguente visualizzazione

```
Area = 21
```

In un programma Java le variabili possono essere dichiarate in una posizione qualsiasi del programma ma naturalmente prima di essere utilizzate. Variabili dello stesso tipo possono essere dichiarate in sequenza, separate da una virgola.

```
int base, altezza, area;
```

Dopo la dichiarazione di tipo sono specificati gli identificatori di variabile, che possono essere in numero qualsiasi, separati da virgola e chiusi da un punto e virgola. È possibile assegnare un valore a una variabile direttamente in fase di dichiarazione, anche nel caso in cui tale valore derivi da un'espressione. Nel nostro esempio è possibile usare la notazione

```
int base = 3;
int altezza = 7;
int area = base * altezza;
```

oppure, in modo ancora più conciso

```
int base = 3, altezza = 7, area = base * altezza;
```

Esistono delle regole da rispettare nella costruzione degli *identificatori*: essi devono iniziare con una lettera o un carattere di sottolineatura _ o un simbolo del dollaro $ e possono contenere un numero qualsiasi di lettere, cifre, caratteri di sottolineatura e caratteri dollaro. Le lettere maiuscole sono considerate diverse dalle rispettive minuscole. Il carattere di sottolineatura e il carattere dollaro solitamente sono usati solo in codice generato da strumenti automatici.

È importante notare che Java è stato sviluppato tenendo conto dello standard *Unicode 2.0*. Ciò significa che un identificatore può essere scritto con ideogrammi giapponesi o, facendo un esempio più pratico per le nostre esigenze, utilizzando caratteri accentati. Sono identificatori validi `città` e `òpera` e non coincidono rispettivamente con gli identificatori `citta` e `opera`. Un identificatore non può essere una parola riservata per cui, per esempio, la dichiarazione

```
int int;
```

genera un errore in fase di compilazione.

La forma grafica data al programma è del tutto opzionale; una volta rispettata la sequenzialità e la sintassi, la scrittura del codice è libera. In particolare più istruzioni possono essere scritte su una stessa linea oppure un'istruzione può essere scritta su più righe.

```
class AreaRettangolo { public static void main (String
argv[]) {
int base;
int altezza; int area; base = 3; altezza = 7; area = base *
altezza; System
.out.println("Area = ");
System
.
out
.
println
(
area
)
;
}
}
```

Questo programma viene compilato senza errori e genera gli stessi risultati di quello riportato all'inizio del paragrafo, ma è notevolmente meno leggibile.

Per migliorare la leggibilità, è invece opportuno inserire dei commenti nei punti dove il codice è meno comprensibile. Un commento è un brano di testo che viene ignorato dal compilatore e che può essere utilizzato appunto per commentare il codice vero e proprio. Un modo per inserire un commento l'abbiamo già visto e consiste nell'usare due barre affiancate (`//`) prima del testo di commento, che termina alla fine della riga. Commenti su più righe possono essere racchiusi tra una sequenza barra asterisco (`/*`) e asterisco barra (`*/`). Va sottolineato che dopo la sequenza `/*`, un'altra `/*`, antecedente la chiusura del commento con `*/`, viene considerata alla stregua di qualsiasi altra sequenza di caratteri, per cui non sono permessi commenti annidati.

3.3 Tipi primitivi

Nel paragrafo precedente abbiamo introdotto il tipo `int` che permette di memorizzare dei numeri interi. Esso è definito *primitivo* in quanto non ha nessuna delle proprietà che caratterizzano gli oggetti. Come vedremo nel seguito, si può pensare a un oggetto come a un tipo definito dal programmatore che è generalmente composto dall'aggregazione di più tipi primitivi e/o altri oggetti.

Gli altri tipi primitivi li trattiamo in sequenza:

- logici: `boolean`
- numerici interi: `byte`, `short`, `long` oltre a `int` già esaminato

- numerici floating point: `float` e `double`
- caratteri: `char`

Per la rappresentazione dell'informazione si veda anche l'Appendice appositamente dedicata.

3.3.1 Booleani

Il tipo primitivo più semplice è il `boolean`, può contenere solo due valori, `true` o `false`, che sono parole riservate. Ogni volta che in un programma appare un confronto tra due variabili, a esso corrisponde un valore `boolean`. Vedremo poi che tali tipi sono utilizzati nelle istruzioni di controllo decisionali.

Va notato che mentre in altri linguaggi il valore di falso corrisponde al numero 0, mentre il valore vero corrisponde a un numero diverso da 0 (tipicamente 1 o -1), in Java viceversa i valori vero e falso sono totalmente distinti dai numeri e non è ammessa nessuna commistione con i tipi numerici. Il frammento di programma seguente

```
boolean a = 2 > 1;
boolean b = false;
System.out.println (a);
System.out.println (b);
System.out.println (2 < 1);
```

produce la visualizzazione

```
true
false
false
```

3.3.2 Numerici interi

Esistono differenti tipi che consentono di memorizzare dei numeri interi e si distinguono l'uno dall'altro in base alla diversa occupazione di memoria e alla conseguente differente capacità di contenimento. Il tipo più piccolo è il `byte` che occupa per l'appunto 8 bit e può contenere numeri compresi tra -128 e 127. Questo tipo di variabili è utilizzato generalmente per analizzare sequenze di byte, per esempio letti da un file o inviati verso un file, che necessitano di un'interpretazione da parte del programma. È comunque sempre possibile utilizzare queste variabili in espressioni che comprendono le quattro operazioni.

Il tipo `short` occupa 16 bit e può contenere valori compresi tra -32768 e 32767. Normalmente non viene utilizzato in quanto a esso si preferisce il tipo `int`; l'eventuale spreco di due byte, infatti, non rappresenta quasi mai un problema e dal punto di vista delle prestazioni sui computer a 32 bit risulta più efficiente il trattamento di quest'ultimo tipo.

Quando il tipo `int` non è più sufficiente a contenere i numeri che ci interessano, si può utilizzare il tipo `long` che occupa 64 bit e quindi può contenere numeri compresi tra -9'223'372'036'854'775'808 e 9'223'372'036'854'775'807.

3.3.3 Numerici floating point

Per elaborare numeri con decimali, Java mette a disposizione due tipi a *virgola mobile* (floating point), il `float` che occupa 32 bit e può contenere numeri positivi e negativi compresi tra $1.40129846432481707*10^{-45}$ e $3.40282346638528860*10^{38}$, e il `double` che occupa 64 bit e può contenere numeri positivi e negativi compresi tra $4.94065645841246544*10^{-324}$ e $1.79769313486231570*10^{308}$. Anche se questi tipi possono contenere valori molto grandi, o molto piccoli, le cifre significative su cui si può fare affidamento sono non più di 8 per i `float` e non più di 16 per i `double`. Calcoli accurati meritano quindi l'utilizzo del tipo `double` e, infatti, tutte le funzioni matematiche che trattano numeri reali fornite con il linguaggio utilizzano numeri di questo tipo. Dal punto di vista delle prestazioni, alcuni moderni processori eseguono più velocemente i calcoli a 64 bit che a 32.

3.3.4 Promozioni e casting

Nella stesura dei programmi capita spesso di dover spostare dei valori numerici fra variabili di tipo diverso. Nel caso in cui un valore contenuto in una variabile di un certo tipo sia assegnato a una variabile di un tipo più capace, come nell'esempio seguente

```
byte b = 100;
int i = b;
```

Java esegue una conversione automatica. Questo tipo di conversione è detto *widening* o *promozione*. L'operazione contraria, cioè lo spostamento del contenuto di una variabile di un certo tipo in una variabile di un tipo meno capace, non è automatica in quanto potrebbe portare alla perdita di cifre significative. Il compilatore Java esegue dei controlli accurati sugli assegnamenti dei numeri, verificando che la variabile che riceve un valore abbia la capacità sufficiente. L'esempio seguente

```
byte b = 128;
```

provoca un errore in fase di compilazione. Se in questo caso il compilatore segnala una situazione realmente anomala, in altri casi il controllo può risultare inopportuno. Il codice seguente, per esempio

```
int i = 10;
byte b = i;
```

provoca un errore in fase di compilazione, anche se in effetti le operazioni sono del tutto legittime. Per ovviare a quest'inconveniente, è possibile effettuare un *cast*, cioè forzare una variabile di un certo tipo a diventare di un tipo diverso. L'operatore di cast, che s'indica racchiudendo tra parentesi tonde il tipo da forzare, va premesso al valore da forzare. Il codice precedente per essere compilato senza errori deve diventare quindi

```
int i = 10;
byte b = (byte)i;
```

Nell'esempio un tipo viene forzato a un tipo meno capace; ci si riferisce a quest'operazione col termine di *narrowing* (restringimento). Se, nell'ultimo esempio, `i` contenesse un valore che eccede la capacità di un `byte`, i bit più significativi sarebbero ignorati. Con le istruzioni

```
int i = 257;
byte b = (byte)i;
```

alla variabile `b` viene assegnato il valore 1. Infatti la rappresentazione binaria a 16 bit (ampiezza di un `int`) del decimale 257 è

0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1

la conversione nel tipo `byte` considera solo i primi 8 bit a destra:

0	0	0	0	0	0	0	1

che in decimale corrisponde a 1.

Nel calcolo di espressioni, i risultati parziali delle operazioni sono memorizzati in variabili implicite di tipo `int`, a meno che nell'espressione non siano coinvolte variabili di tipo più ampio, cioè in ordine crescente `long`, `float` e `double`, nel qual caso i risultati intermedi sono memorizzati in variabili intermedie di quel tipo. Nell'esempio riportato

```
byte b1 = 100;
byte b2 = 120;
int i = b1 * b2 / 2;
```

viene utilizzata una variabile implicita di tipo `int` per contenere il risultato del prodotto `b1*b2`, per cui a `i` viene assegnato il valore corretto. Questo tipo di strategia può portare però a risultati scorretti quando i termini di un'espressione sono di tipo `int` e i calcoli parziali eccedono la capacità di questo tipo. L'esempio seguente

```
int i1 = 2000000000;
int i2 = 2000000000;
long l = i1 * i2 / 2000000000;
```

viene compilato senza errori, ma al termine dell'elaborazione la variabile `l` contiene inopinatamente il valore 0. Per rimediare a questo comportamento bisogna ricorrere nuovamente a un cast che effettui la promozione di almeno uno dei termini dell'espressione. Dopo l'elaborazione del codice

```
int i1 = 2000000000;
int i2 = 2000000000;
long l = (long)i1 * i2 / 2000000000;
```

la variabile `l` contiene il valore 2'000'000'000 come atteso.

La promozione automatica vista precedentemente può dar luogo a errori in fase di compilazione che possono apparire oscuri. Compilando il codice seguente

```
byte b = 1;
b = b + 1;
```

si ottiene infatti un errore. Il motivo ormai l'abbiamo spiegato e risiede nel fatto che il risultato della somma viene memorizzato in una variabile implicita di tipo `int`, per cui è necessario un cast esplicito per superare la compilazione.

```
byte b = 1;
b = (byte) (b + 1);
```

3.3.5 Caratteri

Per contenere i caratteri alfanumerici da usare in testi, messaggi ecc. si ha a disposizione il tipo `char`. Poiché Java adotta lo standard Unicode, che come abbiamo detto codifica i più diffusi alfabeti del mondo, il `char` occupa 16 bit. Questo tipo è assimilabile perfettamente ai tipi numerici interi già visti, con l'unica peculiarità di non accettare valori negativi: esso, infatti, può contenere numeri compresi tra 0 e 65535. Nella codifica Unicode, i primi 256 caratteri corrispondono al set di caratteri denominato ISO-Latin-1 che, a sua volta, ha i primi 128 caratteri corrispondenti al set di caratteri ASCII. Questo permette a Java di funzionare correttamente anche su sistemi che non prevedono ancora il set di caratteri Unicode.

3.3.6 Costanti

Abbiamo visto che in Java è presente un discreto numero di tipi numerici e che il compilatore esegue dei controlli sulla validità degli assegnamenti. Per questo motivo esistono diversi tipi di *costanti letterali* o *literal*. Le costanti letterali sono la rappresentazione esplicita di un valore di un determinato tipo; per esempio nell'espressione seguente

```
int i = 123;
```

123 è una costante letterale. Java considera un numero scritto in questo modo come un tipo intero la cui dimensione sia la minima necessaria per contenere il valore e non superiore a 32 bit. Gli esempi seguenti illustrano la situazione

```
byte b0 = 127;              // Corretto
byte b1 = 128;              // Errore in compilazione
long l0 = 2147483647;       // Corretto
long l1 = 2147483648;       // Errore in compilazione
```

Quando si ha la necessità di riportare in un programma una costante letterale intera che non può essere contenuta in 4 byte, è necessario posporre alla costante una lettera L, maiuscola o minuscola; è preferibile utilizzare la lettera maiuscola per evitare confusioni con il numero 1.

L'esempio seguente

```
long l1 = 2147483648L;      // Corretto
```

viene compilato correttamente. Oltre alla notazione decimale, è possibile usare le notazioni ottale ed esadecimale per la rappresentazione di numeri interi. Queste notazioni sono generalmente utilizzate nei casi in cui un intero sia utilizzato come maschera di bit o per altri usi particolari. Per questo motivo è possibile specificare solo numeri positivi, come se i tipi non avessero segno. In notazione esadecimale, per esempio, è possibile assegnare a una variabile di tipo `byte` valori compresi tra 0 e 7F (127) oppure, utilizzando un cast, fino a FF (255), anche se poi i numeri superiori a 7F sono di fatto negativi. Vedremo nel capitolo successivo come sono rappresentati gli interi negativi.

Un numero in notazione ottale inizia con un carattere 0 e può contenere cifre comprese tra 0 e 7. Nell'esempio

```
int i = 010 - 8;
```

la variabile `i` assume il valore 0, infatti 10 in ottale equivale al valore decimale 8. La notazione ottale aveva un'utilità pratica quando i computer lavoravano a 7 bit ed è supportata in Java per venire incontro a quei programmatori C che sono abituati a utilizzarla. Sui computer moderni che lavorano con un numero di bit multiplo di 8 è più comodo usare la notazione esadecimale. Un numero espresso in questa notazione inizia con i caratteri 0x o 0X e può contenere le cifre da 0 a 9 e le lettere, maiuscole o minuscole, comprese tra A e F. Le lettere sono utilizzate per rappresentare le cifre esadecimali da 10 a 15. Nell'esempio seguente

```
int i = 0x10 - 010;
```

la variabile `i` assume il valore 8, infatti viene sottratto 8 all'esadecimale 10 che equivale al valore decimale 16.

Una costante numerica contenente un punto decimale (.), il corrispondente anglosassone della nostra virgola, viene considerata di tipo `double` e in base 10. Osserviamo i seguenti esempi

```
long l = 0.0;               // Errore in fase di compilazione
float f = 0.0;              // Errore in fase di compilazione
double d0 = 010.0 - 8;      // Alla variabile d è assegnato 2
double d1 = 0x10.0 - 8;     // Errore di compilazione
```

Le prime due righe causano un errore di compilazione in quanto si tenta di assegnare dei valori `double` a delle variabili meno capaci. Nella terza riga il numero `010.0` non è espresso in notazione ottale, anche se inizia con il carattere 0, in quanto il punto decimale lo rende di tipo `double` e quindi espresso in notazione a base 10. La quarta riga non viene compilata perché non è permesso rappresentare i numeri a virgola mobile in notazione esadecimale.

I tipi `float` o `double`, sono spesso usati per trattare numeri molto grandi o molto piccoli, che espressi esplicitamente possono richiedere decine se non centinaia di cifre. Per esprimere in forma più leggibile questo tipo di numeri, è possibile utilizzare

la cosiddetta *notazione esponenziale* o scientifica. Essa consiste di due numeri separati dalla lettera E, maiuscola o minuscola, il primo dei quali, la *mantissa*, può essere un numero intero o con un punto decimale mentre il secondo, l'*esponente*, deve essere intero. Il valore del numero è calcolato moltiplicando la mantissa per 10 elevato alla potenza indicata dall'esponente. Per esempio, il numero 1230 può essere scritto in notazione esponenziale come `1.230e3` ($1{,}230*10^3$) oppure come `123e1` ($123*10^1$). Anche le costanti in notazione esponenziale sono considerate di tipo `double`, indipendentemente dal fatto che la mantissa contenga o no il punto decimale.

È possibile fare in modo che una costante numerica con un punto decimale o in notazione esponenziale sia considerata di tipo `float` anziché `double` posponendo una lettera F, maiuscola o minuscola, alla costante stessa. La riga seguente

```
float f = 0.0F;          // Corretto
```

viene compilata correttamente. Se invece della F si pospone una D, maiuscola o minuscola, si ottiene di nuovo una costante di tipo `double` come se non avessimo posposto nulla.

Per inserire in un programma dei caratteri costanti, è possibile utilizzare le costanti numeriche intere ma questo renderebbe i programmi poco leggibili. L'istruzione seguente

```
char c = 115;
```

non evidenzia in modo chiaro che si sta assegnando la lettera `s` (esse minuscola) alla variabile `c`. Per questo motivo è possibile rappresentare un carattere costante racchiudendolo tra apici singoli. Riscrivendo l'istruzione precedente nel modo seguente

```
char c = 's';
```

l'intenzione del programmatore appare sicuramente più chiara. Una costante carattere corrisponde a un tipo `char` e non può essere assegnata a una variabile di tipo `short` o `byte` senza un cast. Nell'esempio

```
short s0 = 115;          // Corretto
short s1 = 's';          // Errore in compilazione
```

la seconda riga provoca un errore in fase di compilazione.

La rappresentazione esplicita è comoda per i caratteri visibili con l'editor che si utilizza per scrivere i programmi. Per i caratteri che non sono visibili esistono alcune sequenze convenzionali, riconoscibili per il fatto che iniziano tutte con il *carattere di escape*: la barra rovesciata `\`. Per esempio il carattere che inserisce una linea nuova in una stampa (new line) può essere rappresentato con `\n`, quello di tabulazione con `\t` e quello di pagina nuova con `\f`.

La barra rovesciata può essere utilizzata anche per inserire il codice ottale o esadecimale del carattere. Un carattere in notazione ottale è formato dalla barra rovesciata seguita da tre cifre ottali. In questo modo però si possono rappresentare solo caratteri il cui codice sia compreso tra 000 e 377 (255 decimale). Un carattere in notazione esadecimale è formato dalla barra rovesciata seguita da una `u` (per Unicode) e 4 cifre esadecimali (vedi Tabella 3.1).

Tabella 3.1 Sequenze di escape utilizzabili in Java

Sequenze di escape	Descrizione
\OOO	Carattere in notazione ottale
\uXXXX	Carattere Unicode in notazione esadecimale
\'	Apice singolo
\\	Barra rovesciata
\b	Backspace (codice ASCII 8)
\t	Carattere di tabulazione (codice ASCII 9)
\n	Nuova linea (codice ASCII 10)
\f	Nuova pagina (codice ASCII 12)
\r	Ritorno a capo (codice ASCII 13)
\"	Doppi apici

Infine la barra rovesciata permette di togliere il significato speciale a caratteri come l'apice che viene rappresentato come `\'`, per cui

```
char c = '\'';
```

assegna a `c` il carattere apice; `\\` rappresenta semplicemente il carattere di barra rovesciata e non l'inizio di una sequenza di escape.

Fra i tipi primitivi non ne esiste uno che sia adatto a contenere stringhe, cioè insiemi di più caratteri. Questo perché le stringhe sono implementate in Java come oggetti, vale a dire strutture complesse non definite nel linguaggio, ma implementate utilizzando il linguaggio. Ciononostante esiste il modo di rappresentare delle costanti stringa, come del resto abbiamo visto sin dal primo esempio del libro, racchiudendo i caratteri che le compongono tra doppi apici. Costanti stringa sono, per esempio, `"Il primo programma "` e `"Area = "`.

Poiché le costanti stringa sono l'unione di più caratteri, è possibile operarvi con le stesse sequenze di escape, comprese le notazioni ottali e esadecimali, che si utilizzano per le costanti carattere. L'istruzione seguente

```
System.out.println ("Riga \"1\"\nRiga \"2\"");
```

esegue la seguente visualizzazione

```
Riga "1"
Riga "2"
```

Si può notare che la sequenza `\"` in realtà non è indispensabile nelle costanti carattere poiché i doppi apici possono essere rappresentati come carattere con `'"'`

```
char c = '"';
```

mentre la sequenza `\'` non è indispensabile per lo stesso motivo nelle costanti stringa.

Ciononostante entrambe le sequenze sono supportate in entrambi i tipi di costanti per uniformare la notazione.

Non esiste una sequenza per spezzare una costante stringa su più linee di testo, ma è possibile concatenare più stringhe mediante l'utilizzo dell'operatore +. L'istruzione seguente

```
System.out.println ("Stringa" +
                    " " +
                    "concatenata");
```

esegue la seguente visualizzazione

```
Stringa concatenata
```

✔ **NOTA**

In alcune implementazioni di Java, i caratteri Unicode sono riconosciuti e convertiti prima dell'analisi lessicale e quindi al di fuori del contesto in cui si trovano. Questo comporta che se si riportano dei caratteri speciali con la notazione Unicode, questi possono generare degli errori in fase di compilazione. Riprendendo un esempio precedente, se avessimo utilizzato la seguente istruzione

```
System.out.println ("Riga \"1\"\u000aRiga \"2\"");
```

dove `\u000a` *corrisponde al carattere di ritorno a capo, essa verrebbe trasformata prima dell'analisi lessicale in*

```
System.out.println ("Riga \"1\"
Riga \"2\"");
```

causando un errore di stringa non terminata in compilazione. In questi casi è quindi opportuno non usare le sequenze Unicode.

Per terminare la trattazione delle costanti letterali, manca il tipo `boolean`. Come abbiamo già accennato, esistono solo due costanti possibili di questo tipo che sono rappresentate dalle parole riservate `true` (vero) e `false` (falso). Tali costanti possono essere utilizzate in tutti i casi in cui può essere utilizzato un tipo `boolean` e in nessun altro contesto. L'operatore di cast `(boolean)` è sintatticamente accettato, ma può essere applicato solo a variabili o costanti di tipo `boolean`, per cui risulta assolutamente inutile.

3.4 Oggetti

Introduciamo in questo paragrafo il concetto di oggetto: l'argomento verrà approfondito più in avanti, ma al momento torna comodo sapere cosa siano e come si usano perché tutte le interazioni fra programmi e sistema operativo avvengono in Java tramite il loro utilizzo e quindi solo usando gli oggetti si possono sviluppare dei programmi interessanti.

Possiamo pensare a un *oggetto* come a un *tipo definito dal programmatore* (*tipo astratto*) che può essere manipolato in modo simile a un tipo primitivo. Così come le operazioni sui tipi primitivi si fanno tramite gli operatori, le operazioni sugli oggetti si fanno tramite dei metodi, ciascuno dei quali realizza un compito specifico. Una classe descrive come è fatto un oggetto e quali operazioni è possibile fare su di esso: ecco quindi che un oggetto è anche detto *esemplare* (o *istanza)* di una classe.

Abbiamo detto che gli oggetti, o meglio le classi, sono definiti dal programmatore, ma Java è fornito di una nutrita libreria di sistema per affrontare i problemi più comuni; nella libreria si trova la classe `BigInteger` il cui scopo è permettere l'esecuzione di calcoli aritmetici tra numeri interi molto grandi, comprendenti cioè un numero qualsiasi di cifre. Vediamo allora come si può trasformare la classe vista precedentemente in modo da usare oggetti di questa classe.

```
import java.math.BigInteger;

class AreaRettangoloOggetti {
   public static void main (String argv[]) {
      BigInteger base;
      BigInteger altezza;
      BigInteger area;

      base =  new BigInteger("12345678901234567890");
      altezza = new BigInteger("98765432109876543210");
      area = base.multiply(altezza);

      System.out.print ("Area = ");
      System.out.println (area);
   }
}
```

La prima riga serve per informare il compilatore che abbiamo intenzione di usare oggetti della classe `BigInteger`, per il resto si può notare che l'utilizzo di un oggetto è abbastanza simile a quello di un tipo primitivo. Le differenze sostanziali sono:

- gli oggetti debbono essere creati tramite l'utilizzo dell'operatore `new`; fortunatamente non dobbiamo però preoccuparci della loro distruzione perché questa operazione viene fatta automaticamente dall'interprete;
- per eseguire la moltiplicazione non abbiamo usato l'operatore `*` ma abbiamo invocato il metodo `multiply` di `base` fornendo come argomento tra parentesi l'altro numero da moltiplicare. Il risultato è un terzo oggetto contenente il risultato dell'operazione. L'associazione tra oggetto e metodo avviene tramite l'operatore punto (`.`).

Compilando ed eseguendo questo programma, si otterrà il seguente risultato

```
Area = 1219326311370217952237463801111263526900
```

A questo punto ci si può rendere conto che un oggetto era già comparso negli esempi precedenti e precisamente `System.out` (sorvoliamo per il momento sul significato del punto tra le due parole), del quale abbiamo usato due metodi `print` e `println`. È questo un oggetto, creato e messo a disposizione automaticamente dall'interprete, che consente di visualizzare messaggi sulla console. Ne esiste un altro simmetrico, che consente di leggere da tastiera o da un file o da un altro dispositivo, il cui nome è `System.in`. Dalla tastiera però arrivano soltanto caratteri, mentre normalmente i programmi necessitano di numeri e stringhe. Per questo motivo dalla versione 5 di Java è disponibile una classe di utilità che permette di leggere i caratteri che arrivano dalla tastiera e di trasformarli in numeri e stringhe a seconda delle necessità. Tale classe si chiama `Scanner` e fra i molti metodi di cui dispone ne ha uno, `nextInt`, che può tornarci utile immediatamente. Esso infatti restituisce un valore, di tipo `int`, che rappresenta un numero digitato sulla tastiera. Ecco allora che possiamo trasformare i precedenti programmi che calcolano l'area di un rettangolo in modo che eseguano i calcoli sulla base di dati inseriti di volta in volta da tastiera e non cablati direttamente nel programma.

```
import java.util.Scanner;

class AreaRettangoloDaUtente {
   public static void main (String argv[]) {
      int base;
      int altezza;
      int area;
      Scanner in = new Scanner(System.in);

      System.out.print ("Base: ");
      base =  in.nextInt();
      System.out.print ("Altezza: ");
      altezza = in.nextInt();
      area = base * altezza;

      System.out.print ("Area = ");
      System.out.println (area);
   }
}
```

Si noti che all'atto della creazione dell'oggetto `Scanner` gli abbiamo fornito come argomento l'oggetto `System.in` per indicare la volontà di ricevere dati da tastiera. Si noti inoltre che al metodo `nextInt` non viene passato alcun argomento, ma le parentesi rimangono comunque necessarie. Compilando ed eseguendo questo programma, esso rimarrà in attesa fintanto che non digiteremo almeno due numeri seguiti da un invio. Naturalmente il risultato dipenderà dai valori digitati.

Domande di verifica

1. Quale sequenza di istruzioni permette di visualizzare i versi di una poesia, rispettandone gli a capo?
2. In che modo si riesce a far eseguire in sequenza le istruzioni di un programma?
3. Qual è la struttura generale di un programma?
4. Nella stesura di un programma, dopo il punto e virgola si deve andare a capo?
5. Quando è obbligatorio andare a capo, nella stesura di un programma?
6. Come possono essere inseriti i commenti all'interno di un programma?
7. Come si dichiarano le variabili di tipo intero?
8. Qual è l'operatore che permette di assegnare un valore a una variabile?
9. Quali operatori possono essere applicati a valori numerici di tipo intero?
10. Esistono limiti ai valori assegnati a una variabile di tipo intero?
11. Cosa differenzia i tipi `int`, `byte`, `short` e `long`? Quando conviene utilizzare un tipo rispetto a un altro?
12. Scrivere un programma che effettui dei calcoli arbitrari che restituiscano dei risultati interi, assegni tali risultati a variabili di tipo `int`, `byte`, `short`, `long` e ne mostri i risultati.
13. È esatto dire che in Java una variabile di tipo carattere può contenere un carattere alfanumerico purché compreso nel set dei caratteri ASCII?
14. Quali valori può assumere una variabile di tipo `boolean`?
15. Cosa si intende per sequenze di escape?
16. Come si indica al programma che una costante è di tipo ottale o esadecimale?
17. Si possono utilizzare all'interno di espressioni matematiche costanti di tipo esadecimale?
18. Qual è il metodo che permette di acquisire valori immessi dall'utente?

Esercizi

1. Scrivere un programma che, invocando tre volte il metodo `print`, mostri sullo schermo:

```
Esercizio n.1, prove di visualizzazione
```

2. Scrivere un programma che visualizzi

```
Esercizio n.2, prima linea
Esercizio n.2, seconda linea
Esercizio n.2, terza linea
```

3. Codificare un programma che calcoli l'espressione `y = xa+b`, dove `x` è uguale a 5, `a` è uguale a 18 e `b` è uguale a 7 e dove `x`, `a`, `b` e `y` devono essere dichiarate come variabili intere. Si visualizzi infine il valore finale `y = 97`.

4. Modificare il programma dell'esercizio precedente in modo che `a` e `b` non siano variabili ma costanti.

5. Indicare tutti gli errori commessi nel seguente listato.

```
class TuttoSbagliato {
   public static void main (String argv[]) {
      int base;
      int altezza; area;

      base = 3, altezza = 7;
      area = base * altezza;

      System.out.println ("Area = ")
      System.out.println (area);
   }
}
```

6. Tradurre i seguenti numeri in rappresentazione decimale nella corrispondente notazione esponenziale:

   a) `123.456;`
   b) `2700000;`
   c) `0.99999999;`
   d) `0.07.`

7. Verificare con un apposito programma quale dei seguenti valori assume la variabile `i` di tipo `int` se gli viene assegnato:

   a) `-2147483648`
   b) `2`
   c) `(int)2147483648L`
   d) `(int)4294967296L`

8. Verificare con un apposito programma quale dei seguenti valori assume la variabile `b` di tipo `byte` se gli viene assegnato:

   a) `b = (byte)300;`
   b) `b = 44;`
   c) `b = 44+1;`
   d) `b = 44; b = b + 1;`
   e) `b = 44; b = (byte)(b+1);`

   E quale valore assume `b` se gli vengono assegnati 257, 256, 255, 128, 127, -128?

9. Quali valori vengono assegnati alla variabile `b` di tipo `byte` dai seguenti assegnamenti, quando non sopravvengano errori in compilazione:

```
long i = 100;
int j = 1000;
char c = 'a';
```

a) `b = (byte) i;`

b) `b = (byte) j;`

c) `b = (byte) c;`

d) `b = i;`

e) `b = j;`

f) `b = c;`

10. Codificare un programma che calcoli l'espressione `y = xa+b`, dove il valore di `x` è richiesto all'utente, `a` è uguale a 18 e `b` è uguale a 7 e dove `x`, `a`, `b` e `y` devono essere dichiarate come variabili intere. Si visualizzi infine il valore finale di `y`.

11. Realizzare un programma che calcoli l'area del rettangolo richiedendo all'utente i valori di base e altezza, dove tali valori sono di tipo `float`. [La classe `Scanner` possiede il metodo `nextFloat` (il cui utilizzo è analogo a `nextInt` visto nell'esempio del capitolo) che acquisisce dei caratteri da tastiera e restituisce un valore, di tipo `float`. Il valore deve essere scritto dall'utente in formato esponenziale, per esempio 300 va espresso come 3e2 o 3E2 e 0.2 va espresso come 2E-1 o 2e-1.]

Capitolo 4

Strutture di controllo decisionali

Obiettivi didattici

- `if-else`, `switch-case`
- Istruzioni composte
- Selezioni annidate
- Gerarchia e associatività degli operatori
- Espressioni
- Operatori aritmetici, di assegnamento, relazionali e logici
- Espressioni condizionali
- Operatori bit a bit

4.1 L'istruzione `if`

Quando si desidera eseguire un'istruzione al verificarsi di una certa condizione, si utilizza l'istruzione `if`. Per esempio, se si vuole visualizzare il messaggio '`minore di 100`' solamente nel caso in cui il valore della variabile intera `i` è minore di 100, si scrive:

```
if (i<100) System.out.println ("minore di 100");
```

L'istruzione `if` accetta come argomento un valore booleano racchiuso tra parentesi tonde, verifica se vale `true` e in caso affermativo esegue l'istruzione riportata di seguito, altrimenti salta all'istruzione successiva. L'operatore < è di tipo logico, mette in relazione cioè due valori confrontabili e restituisce un valore booleano in modo analogo a come l'operatore `*` mette in relazione due tipi numerici e restituisce un

valore numerico. Nei paragrafi successivi esamineremo tutti gli operatori logici e vedremo che ne esistono alcuni che mettono in relazione due valori booleani consentendo la scrittura di espressioni logiche. La sintassi dell'istruzione `if` è

```
if (valore-booleano) istruzione1 [ else istruzione2 ];
```

dove la valutazione di *valore-booleano* controlla l'esecuzione di *istruzione1* e *istruzione2*: se *valore-booleano* vale `true` viene eseguita *istruzione1*, altrimenti viene eseguita *istruzione2*.

Nell'esempio precedente è stato omesso il ramo `else`; questo fatto è del tutto legittimo poiché esso è opzionale, come evidenziato dalle parentesi quadre della forma sintattica. Modifichiamo ora l'esempio in modo da visualizzare un messaggio anche quando la variabile `i` non è minore di 100

```
if (i<100)
  System.out.println ("minore di 100");
else
  System.out.println ("maggiore o uguale a 100");
```

Abbiamo introdotto il ramo `else` dell'istruzione `if`, inserendo l'istruzione che visualizza `maggiore o uguale a 100`, da eseguire nel caso in cui il valore booleano sia `false`. Si noti che i rami `if-else` si escludono mutuamente.

4.2 Blocchi

Nell'istruzione `if` soltanto un'istruzione dipende dal valore booleano.

```
if (i < 100)
  System.out.println ("minore di 100");
  System.out.println ("istruzione successiva");
```

Altrettanto dicasi per la clausola `else`

```
if (i < 100)
  System.out.println ("minore di 100");
else
  System.out.println ("maggiore o uguale a 100");
System.out.println ("istruzione successiva alla clausola else");
```

L'ultima visualizzazione viene eseguita indipendentemente dal fatto che `i` risulti minore o maggiore uguale a 100. Supponiamo di avere una variabile di tipo `boolean` `min100` cui vogliamo assegnare il valore `true` nel caso in cui `i` sia minore di 100 e `false` altrimenti. Per fare questo dobbiamo utilizzare un blocco o *istruzione composta*, cioè un insieme di istruzioni inserite tra parentesi graffe che il compilatore tratta come un'istruzione unica. La scrittura

```
{
  System.out.println ("minore di 100");
  min100 = true;
}
```

è un blocco costituito da due istruzioni. La sintassi esposta nel paragrafo precedente è ancora valida se assumiamo che l'istruzione possa essere semplice o composta. Ecco come appare allora il codice dell'esempio aggiornato alle nuove esigenze:

```
boolean min100;
if (i < 100) {
  System.out.println ("minore di 100");
  min100 = true;
} else {
  System.out.println ("maggiore o uguale a 100");
  min100 = false;
}
```

Alcuni programmatori preferiscono come regola utilizzare i blocchi dopo una `if`, anche se da essa dipende un'unica istruzione, per aumentare la chiarezza del codice e per rendere più semplice un eventuale futuro inserimento di altre istruzioni. Il codice

```
boolean min100;
if (i < 100) {
  System.out.println ("minore di 100");
  min100 = true;
} else
  System.out.println ("maggiore o uguale a 100");
  min100 = false;
```

può apparire a prima vista analogo all'esempio precedente mentre in realtà non produrrà mai una variabile `min100` con un valore `true`. Utilizzando sempre le graffe è più difficile commettere sviste di questo tipo.

Un blocco può essere utilizzato nel programma dovunque possa comparire un'istruzione semplice e, quindi, indipendentemente dall'istruzione `if`. All'interno del blocco, dopo la parentesi graffa aperta, possono essere inserite dichiarazioni di *variabili locali* a quel blocco, che sono visibili, cioè utilizzabili, solo fino alla chiusura del blocco stesso.

4.3 `if` annidati

Il costrutto `if` è un'istruzione e quindi può comparire all'interno di un altro `if`, come si deduce dalla sintassi generale, nel ruolo di *`istruzione`*. Quando ciò si verifica, si parla di `if` *annidati*.

```
if (i < 100)
  if (i > 0)
    System.out.println ("minore di 100 e maggiore di zero");
```

Il secondo controllo `(i>0)` viene effettuato soltanto se il primo `(i<100)` ha dato esito positivo. Se anche il secondo `if` risulta vero, viene visualizzata la frase tra virgolette. Si osservi che, dopo il primo `if`, non è necessario inserire il secondo `if` e la

visualizzazione all'interno di parentesi graffe, in quanto queste costituiscono già un'unica istruzione semplice. Aggiungiamo ora al nostro esempio un ramo `else`:

```
if (i < 100)
  if (i > 0)
    System.out.println ("minore di 100 e maggiore di zero");
  else
    System.out.println ("minore o uguale a zero");
```

Come fa capire il messaggio della seconda visualizzazione, l'`else` che abbiamo aggiunto si riferisce al secondo `if`, cioè a quello più interno, il quale insieme alle due visualizzazioni e all'`else` costituiscono ancora un'unica istruzione. Per fare in modo che l'`else` si riferisca al primo `if` bisogna usare le parentesi graffe:

```
if (i < 100) {
  if (i > 0)
    System.out.println ("minore di 100 e maggiore di zero");
} else
  System.out.println ("maggiore o uguale a 100");
```

Se avessimo anche l'`else` dell'`if` più interno le parentesi graffe sarebbero superflue:

```
if (i < 100)
  if (i > 0)
    System.out.println ("minore di 100 e maggiore di zero");
  else
    System.out.println ("minore o uguale a zero");
else
  System.out.println ("maggiore o uguale a 100");
```

Questo è reso possibile dal fatto che il tutto viene considerato come un'unica istruzione. Modifichiamo ora il nostro esempio in modo che vengano visualizzati due messaggi diversi nel caso in cui la variabile `i` sia maggiore o uguale a 100, e non soltanto un messaggio generico come sopra:

```
if (i < 100)
  if (i > 0)
    System.out.println ("minore di 100 e maggiore di zero");
  else
    System.out.println ("minore o uguale a zero");
else
  if (i == 100)
    System.out.println ("uguale a 100");
  else
    System.out.println ("maggiore di 100");
```

Come si può osservare, anche la parte *`istruzione`* dell'`else` può essere un'istruzione `if`, la quale, a sua volta, può avere un proprio ramo `else`. L'operatore `==` ha il significato di *uguale a*, e restituisce `true` se i due operandi che mette in relazione

contengono lo stesso valore, `false` altrimenti. Un'ulteriore modifica del nostro esempio consiste nel dare più informazioni nel caso in cui `i` sia minore di 100

```
if (i < 100)
  if (i > 0)
    System.out.println ("minore di 100 e maggiore di zero");
  else
    if (i == 0)
      System.out.println ("uguale a zero");
    else
      System.out.println ("minore di zero");
else
  if (i == 100)
    System.out.println ("uguale a 100");
  else
    System.out.println ("maggiore di 100");
```

È importante notare che nell'esempio precedente non è richiesto l'uso di parentesi graffe.

4.4 Espressioni e operatori

Andiamo ora a esaminare gli operatori utilizzabili nelle espressioni: aritmetici, relazionali, logici, bit a bit, oltre all'operatore di assegnamento e a quello che realizza espressioni condizionate. Alla fine avremo un quadro d'insieme che ci permetterà di lavorare agevolmente con le espressioni e di apprezzare il ricco set di operatori e di controlli messi a disposizione dal linguaggio.

4.4.1 Espressioni aritmetiche

Si definisce *espressione aritmetica* un insieme di valori numerici, contenuti in costanti o variabili o restituiti da metodi, connessi da operatori aritmetici. Il risultato di un'espressione aritmetica è sempre un valore numerico. Gli operatori aritmetici presenti in Java sono riportati nella Tabella 4.1 in ordine decrescente di precedenza associativa.

Tabella 4.1 Operatori aritmetici

massima precedenza		
negazione (- operatore unario)	autoincremento (++)	autodecremento (--)
prodotto (*)	divisione (/)	modulo (%)
somma (+)	differenza (-)	
	assegnamento (=)	
minima precedenza		

Va osservato come non sia presente l'operatore di elevamento a potenza. Gli operandi degli operatori aritmetici precedenti possono essere solo tipi numerici primitivi, escluso l'operatore assegnamento che può essere utilizzato per qualsiasi tipo. Il simbolo + inoltre, come abbiamo già visto, inserito tra due valori stringa, assume il significato di operatore di concatenazione tra stringhe.

In un'espressione, sono valutati prima gli operatori che hanno una precedenza più alta. Questo consente di scrivere espressioni aritmetiche secondo le normali regole algebriche in modo analogo a come siamo abituati a calcolarle manualmente. L'espressione seguente, per esempio

```
3 + 4 * 2
```

restituisce il valore 11 poiché, come risulta dalla Tabella 4.1, il prodotto ha una precedenza maggiore della somma, per cui viene valutato prima 4*2 e al risultato viene aggiunto 3. Gli operatori +, -, *, / e % sono *associativi a sinistra* (left-associative), il che significa che operatori con uguale precedenza vengono raggruppati da sinistra a destra. Nell'espressione seguente, per esempio

```
3 - 4 + 2
```

viene valutata prima la sottrazione e successivamente l'addizione, visto che gli operatori + e - hanno la stessa precedenza. Se si racchiude una parte dell'espressione tra parentesi tonde, questa sarà valutata per prima, consentendo così di modificare la precedenza degli operatori. Per esempio l'espressione

```
(3 + 4) * 2
```

restituisce il valore 14 in quanto le parentesi tonde fanno in modo che sia valutata prima l'addizione e successivamente il prodotto. A differenza di quanto si fa usualmente quando si scrive un'espressione da calcolare manualmente, in Java non si possono utilizzare parentesi quadre e graffe, ma al posto di queste devono essere ancora utilizzate le parentesi tonde. L'espressione seguente

```
((3 + 4) * 2 + 1) * 3
```

restituisce 45 poiché viene prima calcolato 3+4, il cui totale viene moltiplicato per 2 e a esso viene aggiunto 1; il risultato viene infine moltiplicato per 3 ottenendo così il valore 45. Non esiste un limite al numero di parentesi tonde aperte contemporaneamente, a patto ovviamente che a ciascuna parentesi tonda aperta ne corrisponda una chiusa.

Gli operatori di negazione, autoincremento e autodecremento sono *unari*, si applicano cioè a un solo operando. Vediamo ora l'operatore negazione e rimandiamo a un capitolo successivo gli operatori di autoincremento e autodecremento. Se x ha un valore 5, l'espressione

```
-x
```

restituisce il valore -5, mentre

```
2 * -(x - 6)
```

restituisce 2. Si noti che il - anteposto alla parentesi tonda aperta corrisponde all'operatore unario di negazione, mentre l'altro rappresenta l'operatore di sottrazione.

L'operando di un operatore di negazione può essere un qualsiasi tipo primitivo numerico, compreso il `char`. L'esempio seguente

```
byte b = 35;
short s = -b;
```

genera un errore in compilazione poiché l'operazione di negazione è a tutti gli effetti un'espressione che, come detto nel capitolo precedente, utilizza dei tipi `int` per memorizzare i risultati intermedi. Per rendere compilabile il codice precedente, è perciò necessario utilizzare un cast nel modo seguente

```
byte b = 35;
short s = (short) -b;
```

L'operatore `%` di modulo consente di ottenere il resto della divisione intera fra l'operando di sinistra e quello di destra. Con l'operatore modulo è quindi sempre vera la seguente identità

```
n % d == n - (n / d) * d
```

dove del quoziente della divisione `n/d` viene considerata solo la parte intera, troncando gli eventuali decimali. Questa identità rimane valida anche utilizzando operandi di segni diversi. L'espressione

```
34 % 5
```

restituisce il valore 4 (34-6*5). L'operatore modulo può avere come operandi anche numeri a virgola mobile. L'espressione seguente

```
15.9415 % 3.99
```

restituisce il valore 3,9715 in quanto:

- 15,9415 / 3,99 dà come quoziente 3,995363408521
- la parte intera di 3,995363408521 è 3
- moltiplicando 3 * 3,99 si ottiene 11,97
- sottraendo 11,97 da 15,9415 si ottiene 3,9715

L'assegnamento è un operatore con precedenza minore di tutti gli altri; per questa ragione nell'espressione

```
y = 3 + 4 * 2;
```

prima viene valutata completamente la parte a destra dell'assegnamento e poi il risultato viene assegnato nella variabile `y`. L'operazione di assegnamento può essere multipla

```
x = y = z;
```

In questo caso il valore di `z` viene assegnato sia a `x` che a `y`. Va sottolineato che questo tipo di operazione è possibile poiché l'operatore `=`, a differenza di quelli già visti, è *associativo a destra* (right-associative), il che comporta che nell'espressione precedente sia valutato prima `y=z` e successivamente `x=y`. Si può avere quindi anche

```
a = b = c = d = f;
```

dove il valore di `f` viene assegnato ad `a`, `b`, `c` e `d` o anche

```
x = y = t = z * 2 / x;
```

dove il valore restituito da `z*2/x` viene assegnato a `x`, `y` e `t`. Poiché anche l'assegnamento `=` è un operatore, è possibile utilizzare le parentesi tonde per modificarne la precedenza. L'espressione seguente

```
x = (y = 3) * 2;
```

assegna il valore 3 alla variabile `y` e il valore 6 alla variabile `x`.

4.4.2 Espressioni logiche

Un'*espressione logica* genera come risultato un valore booleano. Le espressioni logiche possono contenere degli *operatori relazionali*, usati per confrontare fra loro i valori. Gli operatori relazionali hanno anch'essi una precedenza associativa; la Tabella 4.2 mostra gli operatori relazionali in ordine decrescente di precedenza.

Tabella 4.2 Operatori relazionali

massima precedenza			
maggiore di (>)	minore di (<)	maggiore uguale (>=)	minore uguale (<=)
	uguale a (==)	diverso da (!=)	
minima precedenza			

Si noti come l'operatore di uguaglianza == sia diverso dall'operatore di assegnamento.

Gli operatori relazionali riportati nella prima riga possono confrontare solo tipi numerici primitivi, mentre quelli della seconda possono confrontare anche booleani e referenze come, per esempio, gli oggetti. In quest'ultimo caso però non viene confrontato il contenuto, bensì se una referenza è uguale a un'altra, per esempio se entrambe referenziano uno stesso oggetto, oppure se una referenza è uguale o diversa da `null`. Il codice

```
Object a = System.out;
System.out.println (a != null);
```

visualizza

```
true
```

Accessoriamente si può notare che, se non avessimo assegnato un valore ad `a`, il codice precedente non verrebbe compilato in quanto Java non permette in generale di utilizzare variabili non inizializzate. Vedremo in seguito che il problema delle variabili non inizializzate si pone in un contesto più ampio.

Poiché gli operatori relazionali hanno precedenze differenti, nell'espressione

```
1 < 3 == 0 < 4
```

vengono valutati prima i confronti col < poiché questo operatore ha precedenza più alta di ==. L'utilizzo delle parentesi potrebbe modificare l'ordine di valutazione degli operatori relazionali, ma in realtà non ha alcuna praticità. L'espressione seguente, per esempio

```
1 < (3 == 0) < 4
```

non viene compilata, in quanto il confronto tra parentesi genera un tipo booleano e a esso non è possibile applicare l'operatore < né nessun altro operatore, esclusi == e != che però hanno la stessa priorità.

In una espressione logica gli operandi vengono promossi secondo le regole già viste per le espressioni aritmetiche, vale a dire che se gli operandi sono di tipo meno capace di `int`, essi vengono promossi a `int`, mentre se fra essi ne esiste uno di tipo più capace, gli operandi vengono promossi a questo tipo.

Gli *operatori logici* consentono di concatenare tra loro più valori booleani e di negare un valore booleano. La Tabella 4.3 mostra gli operatori logici in ordine decrescente di precedenza.

Tabella 4.3 Operatori logici

massima precedenza
NOT (!)
AND (&)
XOR (^)
OR (|)
AND short circuit (&&)
OR short circuit (||)
minima precedenza

L'operatore & (AND) ha come operandi due valori booleani e restituisce `true` se gli operandi hanno entrambi valore `true`, `false` altrimenti. L'espressione

```
x == y & a > b
```

restituisce `true` solo se `x` è uguale a `y` e contemporaneamente `a` è maggiore di `b`.

L'operatore ^ (XOR) ha anch'esso due operandi booleani e restituisce `true` se solo uno dei due ha valore `true`, `false` se entrambi hanno lo stesso valore. L'espressione

```
x == y ^ a > b
```

restituisce `true` se `x` è uguale a `y` oppure se `a` è maggiore di `b`, `false` se entrambi i confronti restituiscono `true` o `false`.

L'operatore | (OR) ha due operandi booleani come i precedenti e restituisce `true` se almeno uno di essi ha valore `true`. L'espressione

```
x == y | a > b
```

restituisce `false` solo se `x` è diverso da `y` e contemporaneamente `a` è minore o uguale a `b`.

L'operatore ! (NOT) ha un solo operando e restituisce `false` se l'operando ha valore `true` e `true` se l'operando ha valore `false`. La Tabella 4.4 è la *tavola di verità* degli operatori illustrati.

Tabella 4.4 Tavola di verità degli operatori logici

x	y	AND	XOR	OR	NOT x
false	false	false	false	false	true
true	false	false	true	true	false
false	true	false	true	true	true
true	true	true	false	true	false

Come si vede nella Tabella 4.3, esistono altri due operatori, che sono una variante di AND e di OR. Essi sono definiti di *corto circuito (short circuit)* e sono stati pensati basandosi sull'osservazione che mettendo in AND o in OR due valori booleani, spesso è possibile conoscere il risultato dell'espressione solo in base al primo valore. Se, infatti, due valori booleani sono in AND, quando il primo ha valore `false`, il risultato sarà comunque un valore `false` indipendentemente dal valore del secondo. Analogamente se il primo operando in un OR ha valore `true`, il risultato sarà comunque `true`. Gli operatori short circuit `&&` e `||` si comportano quindi rispettivamente come `&` e `|` ma evitano di valutare il secondo valore booleano nei casi in cui non è necessario. Questo comportamento è utile nei casi in cui il secondo operando dipende dal primo per un corretto funzionamento. Nell'esempio seguente

```
a != 0 && b / a > 100
```

l'utilizzo di un operatore short circuit evita che venga effettuata una divisione per 0 che genererebbe un'eccezione in esecuzione.

Nei programmi si utilizzano di solito questi operatori short circuit in quanto tipicamente si risparmiano operazioni e perché i programmatori C sono abituati a usarli, mancando quel linguaggio di operatori AND e OR effettivi. Questo di solito non ha controindicazioni, tranne in alcuni casi in cui la valutazione di entrambi gli operandi è richiesta. Nell'esempio seguente

```
a != 0 & (b = c) > 0
```

il valore della variabile `c` viene assegnato alla variabile `b` in ogni caso; se avessimo utilizzato un operatore short circuit, l'assegnamento sarebbe dipeso dal valore della variabile `a`.

Come abbiamo visto in questi ultimi esempi, un'espressione logica può contenere delle espressioni aritmetiche, per cui è opportuno riassumere nella Tabella 4.5 la precedenza degli operatori trattati in ordine decrescente.

Tabella 4.5 Scala di priorità degli operatori esaminati

massima precedenza				
		( )		
!	- (unario)		++	--
	*	/	%	
	+		-	
<	>		<=	>=
	==		!=	
		&		
		^		
		|		
		&&		
		||		
		?:		
		=		
minima precedenza				

Nella prima riga abbiamo messo le parentesi tonde per evidenziare che con esse è sempre possibile assegnare la massima precedenza a un qualsiasi operatore. L'operatore di assegnamento ha la precedenza più bassa ed è per questo che è possibile effettuare assegnamenti del tipo seguente

```
b = i < 100
```

dove b è una variabile di tipo booleano. Aggiungendo un punto e virgola si trasforma l'espressione in un'istruzione

```
b = i < 100;
```

Esaminiamo un altro esempio:

```
x = a == b;
```

dove x è una variabile di tipo booleano che prende il valore `true` se a è uguale a b, `false` altrimenti. Dunque è lecito anche questo assegnamento, la cui interpretazione è lasciata al lettore:

```
x = (a == b) && (z != t || m >= n);
```

✔ **NOTA**

Da quanto detto risulta evidente gli operatori `^` *e* `!=` *con operandi booleani danno gli stessi risultati ma hanno precedenze diverse. L'espressione seguente, per esempio*

```
a == b ^ c == d;
```

è del tutto equivalente a

```
(a == b) != (c == d);
```

Vedremo in seguito che l'operatore `^` *può essere utilizzato per compiere un'operazione di XOR bit a bit (bitwise XOR).*
Va sottolineato che non è il simbolo che viene usato per due funzioni diverse, ma l'operatore con la propria precedenza. In altri linguaggi a oggetti, come per esempio il C++, lo stesso operatore con la propria precedenza può essere ridefinito dall'utente per trattare con qualsiasi tipo di dato. Un operatore quindi non ha un significato di per sé, ma ne assume uno solo in relazione ai tipi di dati a cui è applicato. In Java non è permesso all'utente di ridefinire gli operatori, perlomeno fino a oggi, ma il principio è lo stesso, come mostra il caso eclatante dell'operatore `+`.
Analogamente `==` *può essere considerato come un NOT XOR.*

4.4.3 Espressioni condizionate

Degli operatori illustrati nella Tabella 4.5 rimane da vedere quello nella terzultima riga, `?:`, che è un operatore ternario e ha la seguente sintassi:

valore-booleano `?` *valore1* `:` *valore2*

Se *valore-booleano* vale `true` l'intera espressione vale *valore1*, altrimenti vale *valore2*. Per esempio l'espressione seguente

```
x == y ? a : b
```

vale `a` se `x` è uguale a `y`, `b` altrimenti. Questo operatore è comodo quando si debbono fare degli assegnamenti condizionati dal valore di una variabile. Nell'espressione seguente, per esempio

```
v = x == y ? a * c  + 5 : b - d;
```

alla variabile `v` viene assegnato il valore `a*c+5` se `x` è uguale a `y`, altrimenti le viene assegnato il valore `b-d`. Notare che questo tipo di comportamento è dettato dal fatto che l'operatore `?:` ha una precedenza appena superiore a quella dell'assegnamento, il che significa che viene valutata dopo tutte le espressioni logiche e aritmetiche e giusto prima dell'assegnamento. Un'espressione condizionata non è un'istruzione assolutamente necessaria, in quanto può essere rimpiazzata mediante l'uso di istruzioni `if`. L'esempio precedente, potrebbe essere, infatti, scritto nel modo seguente

```
if (x == y)
  v = a * c  + 5;
else
  v = b - d;
```

Un'espressione condizionata va usata tipicamente al posto di un `if` in quelle situazioni in cui la maggiore concisione porta una migliore leggibilità del codice. Riportiamo di seguito alcuni utilizzi comuni di espressioni condizionate.

```
x > y ? x : y          // Valore massimo tra x e y
x >= 0 ? x : -x        // Valore assoluto di x
y != 0 ? x / y : 0     // Vale 0 se y è 0, x/y altrimenti
```

Le espressioni condizionate restituiscono un valore e possono essere inserite in qualsiasi punto in cui il valore restituito sia ammesso. È perfettamente lecito scrivere per esempio

```
a = (x + y) / (x > y ? x : y);
```

che assegna ad `a` il valore della somma di `x+y` diviso per il massimo valore tra i due. Da notare che l'espressione condizionata è stata messa tra parentesi tonde a causa della bassa precedenza dell'operatore `?:`. Lo stesso risultato può essere ottenuto mediante l'uso di `if` con il codice seguente

```
if (x > y)
  a = (x + y) / x;
else
  a = (x + y) / y;
```

Un'espressione condizionata può essere presente anche all'interno di un'altra espressione condizionata o, come si usa dire, può essere annidata. Nell'esempio precedente si ottiene un'eccezione in fase di esecuzione se il valore massimo tra `x` e `y` vale 0, evitiamolo scrivendo

```
a = (x + y) / (x > y ? (x == 0 ? 1 : x) : (y == 0 ? 1 : y));
```

che corrisponde a

```
if (x > y)
  if (x == 0)
    a = (x + y) / 1;
  else
    a = (x + y) / x;
else
  if (y == 0)
    a = (x + y) / 1;
  else
    a = (x + y) / y;
```

Per finire, notiamo che in un'espressione condizionata viene valutato sempre uno solo dei valori dopo il punto interrogativo. Il brano di codice seguente

```
int a = 0, b = 0, c = 0;
int x = 6, y = 8;
a = x > y ? (b = x) : (c = y);
```

```
System.out.println (a);
System.out.println (b);
System.out.println (c);
```

visualizza

```
8
0
8
```

poiché l'espressione `x>y` restituisce il valore `false` e di conseguenza ad `a` viene assegnato il valore di `(c=y)`, il quale comporta l'assegnamento del valore di `y` a `c`. Il termine `(b=x)` non viene valutato, per cui la variabile `b` contiene il valore con cui è stata inizializzata, cioè 0.

4.4.4 Operatori sui bit

Su valori numerici interi, appartenenti cioè ai tipi `byte`, `char`, `short`, `int` e `long`, è possibile applicare degli operatori che consentono di agire sulla rappresentazione del dato in termini di bit. Per prima cosa vediamo dunque come tali tipi sono rappresentati in termini di bit.

In Java la rappresentazione dei numeri è indipendente dal tipo di processore su cui il programma viene eseguito. Negli interi formati da più di un byte, il byte più significativo si trova a sinistra e man mano che si procede verso destra si trovano i byte meno significativi. Ci si riferisce a un tale tipo di rappresentazione con il termine *big endian*. Alcuni processori, come quelli della serie x86 che equipaggiano i personal computer IBM compatibili, funzionano al loro interno utilizzando una rappresentazione opposta, detta *little endian*; su processori di questo tipo la macchina virtuale di Java si incarica di effettuare le opportune conversioni in modo del tutto trasparente ai programmi. Per fare un esempio, il numero 1'000'000'001 (un miliardo e uno) è formato da $59 * 256^3 + 154 * 256^2 + 202 * 256 + 1$ per cui in Java viene rappresentato come `int` in termini di bit nel modo seguente:

```
00111011 10011010 11001010 00000001
   =        =        =        =
  59       154      202        1
```

I numeri interi negativi sono rappresentati in *complemento 2*. La regola per ottenere un numero negativo partendo dal suo valore assoluto è la seguente: si prende il numero positivo e, procedendo da destra verso sinistra, si copiano i bit a 0 nel numero negativo, nella stessa posizione, finché si trova il primo bit a 1; si copia anche tale bit, ma da questo punto in avanti i restanti bit vengono *complementati*, vale a dire quando nel numero positivo c'è un 1 nel numero negativo si mette uno 0 e viceversa. Per fare un esempio, il valore 12 ($2^3 + 2^2$) viene rappresentato in un tipo `byte` nel modo seguente

```
00001100
```

Il numero -12 verrà rappresentato come

```
11110100
```

In questo modo il numero -1 viene rappresentato come

```
11111111
```

mentre il massimo negativo in un tipo `byte`, cioè -128, viene rappresentato come

```
10000000
```

Analogamente il numero -1 viene rappresentato in uno `short` come

```
1111111111111111
```

mentre il massimo negativo, cioè -32768, viene rappresentato come

```
1000000000000000
```

La rappresentazione in complemento 2 fa in modo che in un numero negativo il bit più significativo del byte più significativo sia sempre a 1, mentre nei numeri positivi sia a 0: perciò ci si riferisce a esso come al *bit del segno*. Vediamo ora gli operatori che ci consentono di agire sui singoli bit dei valori interi. Essi sono riportati in Tabella 4.6.

Tabella 4.6 Scala di priorità degli operatori bit a bit

massima precedenza		
NOT bitwise (~)		
scorrimento a sinistra (<<)	scorrimento a destra (>>)	scorrimento a destra senza segno (>>>)
AND bitwise (&)		
XOR bitwise (^)		
OR bitwise (|)		
minima precedenza		

Si può notare che alcuni di questi operatori sono già stati visti in precedenza con operandi booleani. In questo caso li vedremo con operandi interi, il che li fa comportare differentemente.

Ci sono due categorie di operazioni che possono essere fatte sui bit: il primo tipo permette di farli scorrere verso destra o verso sinistra, mentre il secondo consiste nel considerare ogni bit a 0 come falso e ogni bit a 1 come vero, dopodiché è possibile applicare su di essi le operazioni logiche NOT, AND, XOR e OR. Anche utilizzando questi operatori vengono applicate le regole di promozione viste precedentemente per le espressioni.

L'operatore `&` ha due operandi interi e restituisce un intero i cui bit valgono 1 nelle posizioni in cui gli operandi hanno entrambi un bit 1, e 0 nelle posizioni rimanenti. Nell'esempio

```
byte a = 41;     // 0x29 o 00101001
byte b = 97;     // 0x61 o 01100001
int c = a & b;   // 0x21 o 00100001
```

alla variabile c viene assegnato il valore 33.

L'operatore ^ ha due operandi interi e restituisce un intero i cui bit valgono 1 nelle posizioni in cui uno solo degli operandi ha valore 1, e 0 nelle posizioni in cui gli operandi hanno lo stesso valore. Nell'esempio

```
byte a = 41;     // 0x29 o 00101001
byte b = 97;     // 0x61 o 01100001
int c = a ^ b;   // 0x48 o 01001000
```

alla variabile c viene assegnato il valore 72.

L'operatore | ha due operandi interi e restituisce un intero i cui bit valgono 1 nelle posizioni in cui uno qualsiasi degli operandi, o anche tutti e due, ha valore 1, e 0 nelle posizioni in cui entrambi gli operandi hanno valore 0. Nell'esempio

```
byte a = 41;     // 0x29 o 00101001
byte b = 97;     // 0x61 o 01100001
int c = a | b;   // 0x69 o 01101001
```

alla variabile c viene assegnato il valore 105.

L'operatore ~ ha un operando intero e restituisce un intero i cui bit valgono 1 nelle posizioni in cui l'operando ha valore 0, e 0 nelle posizioni in l'operando ha valore 1. Nell'esempio

```
byte a = 41;    // 0x29 o 00101001
int c = ~a;     // 0xD6 o 11010110
```

alla variabile c viene assegnato il valore -42. La Tabella 4.7 riporta una tavola di verità per gli operatori appena visti.

Tabella 4.7 Tavola di verità degli operatori bit a bit

x	y	AND	XOR	OR	NOT x
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	0	1	0

✔ NOTA

Ci si potrebbe chiedere perché per il NOT bit a bit non è stato usato il simbolo ! in analogia a quanto fatto per gli operatori &, ^ e | ; questa stranezza è, in effetti, dovuta a un retaggio del linguaggio C nel quale, non esistendo tipi booleani, si è dovuto distinguere il NOT logico, che restituisce solo 0 o 1, dal NOT bitwise.

Qualcuno potrebbe inoltre osservare che se considerassimo i tipi booleani come interi di un bit e considerassimo il valore 0 come `false` *e 1 come* `true` *non sarebbe vero che esiste una distinzione tra gli operatori bitwise* `&`, `^` *e* `|` *e i loro omonimi booleani. Questa osservazione è scorretta: se fosse vera, infatti, sarebbe possibile utilizzare l'operatore* `~` *sui tipi booleani e l'operatore* `!` *sui tipi interi, il che invece causa un errore in fase di compilazione. Come siano implementati i booleani in realtà è un argomento che non ci riguarda, l'importante è che dal punto di vista logico i booleani non sono interi.*

L'operatore `<<` ha due operandi interi e restituisce un intero i cui bit corrispondono a quelli del primo operando spostati a sinistra di tante posizioni quanto indicato dal secondo operando. I bit che eccedono a sinistra vengono persi mentre i bit mancanti a destra vengono sostituiti da 0. Va notato però che, per effetto della promozione a `int` effettuata da Java, i bit si perdono a sinistra solo dopo la trentaduesima posizione. Nell'esempio

```
byte a = 41;        // 0x29  o 00101001
int b = a << 1;     // 0x52  o  01010010
int c = a << 3;     // 0x148 o   101001000
```

viene assegnato il valore 82 alla variabile `b` e 328 alla variabile `c`. Osserviamo come lo scorrimento a sinistra di n posizioni corrisponda alla moltiplicazione per 2^n. Se volessimo eliminare i bit oltre l'ottavo potremmo ricorrere a un cast o a un *mascheramento*: quest'ultimo consiste nel mettere il valore risultante in AND con un numero che ha il valore 1 solo nei bit che ci interessano, escludendo gli altri. Nell'esempio

```
byte a = 41;                // 0x29  o 00101001
int b = (byte)(a << 3);     // 0x48  o    01001000
int c = a << 3 & 0xFF;      // 0x48  o    01001000
```

viene assegnato il valore 72 sia alla variabile `b` che alla variabile `c`; nell'ultima istruzione si è sfruttato il fatto che l'operatore `<<` ha maggiore precedenza di `&`.

L'operatore `>>` ha due operandi interi e restituisce un intero i cui bit corrispondono a quelli del primo operando spostati a destra di tante posizioni quanto indicato dal secondo. I bit che eccedono a destra vengono persi mentre i bit mancanti a sinistra vengono sostituiti da 0 se il numero è positivo e da 1 se il numero è negativo. In altre parole, i bit mancanti a sinistra vengono riempiti con il valore del bit del segno. Questo comportamento consente a un numero di rimanere dello stesso segno indipendentemente da quanti scorrimenti a destra siano stati effettuati. Nell'esempio

```
byte a = 41;        // 0x29  o  00101001
int b = a >> 1;     // 0x14  o 00010100
byte c = -41;       // 0xD7  o  11010111
int d = c >> 1;     // 0xEB  o 11101011
```

alla variabile `b` viene assegnato il valore 20 mentre alla variabile `d` viene assegnato il valore -21. Si può notare come lo scorrimento a destra di n posizioni corrisponda alla divisione intera per 2^n arrotondata per difetto.

Supponiamo ora di dover manipolare un'immagine grafica in bianco e nero, memorizzata mediante numeri interi in cui ciascun bit a 1 rappresenta un punto (*pixel*) nero dell'immagine mentre ciascun bit a 0 rappresenta un punto bianco. In tale situazione il bit del segno non ha alcun significato particolare rispetto agli altri bit. In questo caso, come in altri, può rivelarsi utile un'operazione di scorrimento a destra *senza segno* che riempia i bit mancanti a sinistra sempre con 0, in modo simmetrico a quanto visto per lo scorrimento a sinistra.

L'operatore >>> ha due operandi interi e restituisce un intero i cui bit corrisponderanno a quelli dell'operando di sinistra spostati a destra di tante posizioni quanto indicato dall'operando di destra. I bit che eccedono a destra vengono persi mentre i bit mancanti a sinistra vengono sostituiti da 0. È evidente che sui numeri positivi non c'è differenza di comportamento tra >>> e >>. Nell'esempio

```
byte a = -41;              // 0xD7 o                                  11010111
int b = a >>> 1;           // 0x7FFFFFEB o 01111111111111111111111111101011
int c = (byte)(a >>> 1); // 0xEB o                                   11101011
```

alla variabile b viene assegnato il valore 2'147'483'627 in virtù della promozione a `int`. Alla variabile c viene assegnato il valore -21 poiché il cast esclude i bit più significativi, ma in tal modo si perde la differenza di comportamento tra gli operatori >>> e >>. In queste situazioni la soluzione migliore per non avere risultati inaspettati è dunque di lavorare con tipi `int` o `long`.

Concludendo con il trattamento delle espressioni, non si può fare a meno di notare che invece di parlare di operatori aritmetici e logici, sia più corretto parlare di operatori fra tipi numerici e operatori fra tipi booleani. La suddivisione fatta tra espressioni logiche e aritmetiche risulta utile solo per comprendere più facilmente il comportamento degli operatori, ma, di fatto, in un'espressione possono essere mescolati a piacimento operatori booleani e aritmetici, purché abbiano gli operandi del tipo corretto. Riportiamo dunque nella Tabella 4.8 il riassunto degli operatori di Java in ordine decrescente di precedenza.

Agli operatori è stata assegnata una precedenza per permettere la scrittura di espressioni più leggibili, senza utilizzare troppe parentesi e quindi in definitiva per semplificare il compito dei programmatori. Dobbiamo però prestare attenzione perché in alcuni casi si può essere tratti in inganno. Supponiamo di voler controllare se la variabile x di tipo `int` è dispari o pari. Il modo più veloce per effettuare un simile controllo è di osservare il bit meno significativo: se esso vale 1 il numero è dispari, altrimenti è pari. Per controllare il bit meno significativo di un valore è sufficiente metterlo in AND con il numero 1. L'espressione seguente quindi

```
x & 1
```

vale 1 se x è dispari, 0 se x è pari. Alla luce di ciò, potrebbe apparire corretto scrivere

```
if (x & 1 == 1)                    // Errore in compilazione
  System.out.println ("x è dispari");
else
  System.out.println ("x è pari");
```

Tabella 4.8 Scala di priorità di tutti gli operatori

massima precedenza				
		()		
!	- (unario)	~	++	--
	*	/	%	
	+		-	
	<<	>>	>>>	
<	>		<=	>=
	==		!=	
		&		
		^		
		|		
		&&		
		||		
		?:		
		=		
minima precedenza				

Il compilatore viceversa segnala un errore in quanto l'operatore == ha precedenza maggiore dell'operatore & per cui viene valutato per primo, restituendo un valore booleano che non può essere messo in AND con un operando intero. Il codice precedente, per essere compilato e funzionare correttamente, deve essere modificato nel modo seguente

```
if ((x & 1) == 1)                              // Ok
  System.out.println ("x è dispari");
else
  System.out.println ("x è pari");
```

In C avremmo lo stesso tipo di problema con l'aggravante di non ottenere la rilevazione dell'errore in fase di compilazione. Questo depone a favore della distinzione tra interi e booleani e dei controlli sui tipi effettuati da Java. In effetti l'esempio riportato in C funzionerebbe correttamente anche senza parentesi, ma solo perché si tratta di un caso particolare, il che rafforza piuttosto che sminuire la nostra osservazione a favore di Java.

4.5 Istruzione `switch`

Le decisioni a più vie possono essere risolte utilizzando più istruzioni `if-else` in cascata:

```
if (valore-booleano)
  istruzione1
else if (valore-booleano)
  istruzione2
else if (valore-booleano)
  istruzione3
 ...
else
  istruzioneN;
```

Come abbiamo visto, ogni `istruzione` può essere formata da più istruzioni racchiuse tra parentesi graffe (blocchi). In alcuni casi è possibile implementare una decisione a più vie sostituendo le `if-else` in cascata con il più efficiente e leggibile costrutto `switch-case`, il quale consente di trasferire il controllo a un insieme di istruzioni basandosi sul confronto di un'espressione, che può restituire solo valori di tipo `byte`, `char`, `short` o `int`, con un insieme di costanti. La sintassi completa dell'istruzione `switch-case` è

```
switch (espressione) {
case costante1:
  [istruzioni1;]
  [break;]
case costante2:
  [istruzioni2;]
  [break;]
case costante3:
  [istruzioni3;]
  [break;]
 ...
case costanteN:
  [istruzioniN;]
  [break;]
[default:
  [istruzioniDefault;]
  [break;]]
}
```

La parola `switch` è seguita da una `espressione`, racchiusa tra parentesi, il cui risultato deve essere, come detto, un valore di tipo `int` o promovibile a `int` (`byte`, `char`, `short`). Il resto del costrutto è formato da un blocco, costituito da un numero qualsiasi di sottoparti, ciascuna delle quali inizia con la parola chiave `case`, seguita da un'espressione costante di tipo `int` o promovibile a `int`, separata, tramite il sim-

bolo di due punti (:), da una o più istruzioni. In fase di esecuzione, viene valutata espressione e il risultato viene confrontato con costante1, se i due valori sono uguali allora il controllo passa alla prima istruzione che segue i due punti corrispondenti, altrimenti si prosegue confrontando il risultato dell'espressione con costante2 e così di seguito. Una volta che il controllo viene trasferito a una certa istruzione, vengono eseguite tutte le rimanenti istruzioni presenti nello switch-case.

Nell'utilizzo più comune del costrutto, il programmatore desidera che vengano eseguite solamente le istruzioni associate a un singolo case; a questo scopo è possibile utilizzare l'istruzione break che provoca l'uscita immediata dallo switch. Se l'espressione non corrisponde a nessuna delle costanti, il controllo del programma è trasferito alla prima istruzione che segue i due punti dopo la parola riservata default, quando presente. L'opzione default può apparire in qualsiasi posizione anche se in genere si mette prima del primo case o dopo l'ultimo. Si osservi che, anche se nell'utilizzo più comune si mette un'istruzione break al termine di ogni case, in alcuni casi è comodo che più case facciano riferimento alle stesse istruzioni. Se volessimo, per esempio, riconoscere se un intero x compreso tra 1 e 10 è un numero primo o meno, potremmo usare il seguente codice

```
switch (x) {
case 1: //Va al successivo
case 2: //Va al successivo
case 3: //Va al successivo
case 5: //Va al successivo
case 7:
  System.out.print (x);
  System.out.println (" è un numero primo");
  break;
case 4: //Va al successivo
case 6: //Va al successivo
case 8: //Va al successivo
case 9: //Va al successivo
case 10:
  System.out.print (x);
  System.out.println (" non è numero primo");
  break;
default:
  System.out.println ("x è minore di 1 o maggiore di 10");
  break;
}
```

Capita comunemente che il programmatore si dimentichi un break alla fine di un case, il che non provoca nessuna segnalazione da parte del compilatore, essendo questa una possibilità prevista dal linguaggio. Può quindi essere utile segnalare i punti in cui il break è omesso volutamente per distinguere a colpo d'occhio i punti in cui è stato semplicemente dimenticato. A tale scopo nell'esempio precedente abbiamo inserito un commento //Va al successivo. Il break prima della parentesi graffa

che chiude lo `switch` è di fatto inutile, ma è buona norma inserirlo ugualmente per chiarezza e per evitare possibili errori inserendo successivamente altri `case`. I valori costanti dopo un `case` possono essere anche delle espressioni costanti calcolate, come `3*2+5`; in tal caso esse vengono valutate direttamente dal compilatore.

4.6 Istruzione `assert`

Supponiamo che a un certo punto di un programma ci troviamo nella necessità di dover fare la seguente operazione:

```
r = n / d;
```

dove le variabili in gioco sono tutti numeri interi. In casi come questi ci sono due possibilità: o il valore di `d` può essere pari a zero e allora il programmatore deve prevedere questo caso in modo opportuno, oppure `d` deve avere necessariamente un valore diverso da zero, altrimenti significa che c'è un problema da qualche altra parte cui non si può porre rimedio a questo punto. Certo gli errori sono sempre in agguato, per cui un programmatore prudente può decidere di mettere un'istruzione `if` anche in questo secondo caso per evitare errori che possono apparire misteriosi, specialmente in fase di sviluppo. Questa soluzione ha però due inconvenienti: il primo è che l'istruzione `if` in ogni caso richiede una certa elaborazione e, se è inutile quando il programma è finalmente messo a punto, rappresenta uno spreco; il secondo è che leggendo il programma si può non capire che quell'istruzione è stata messa solo per prudenza in fase di sviluppo e da questo poi arrivare a una comprensione inesatta del flusso.

Una soluzione migliore è rappresentata dall'uso delle *asserzioni*, implementate nel linguaggio Java dalla versione 1.4 con l'istruzione `assert`, la cui sintassi è la seguente:

```
assert valore-booleano [ : valore ];
```

Questa istruzione ha la caratteristica di essere eseguita solo quando l'elaborazione è attivata con una particolare opzione, tipicamente `-ea`, e di essere totalmente ignorata altrimenti. Il suo funzionamento è molto semplice: viene valutato il valore booleano e nel caso che sia falso, l'elaborazione termina immediatamente informando l'utente che un'asserzione è fallita ed eventualmente visualizzando il valore dopo i due punti, nel caso sia specificato.

L'uso delle asserzioni rappresenta un valido modo, oltre che per rendere più sicuri i programmi, per documentare le condizioni che il programmatore si aspetta siano rispettate affinché un algoritmo esegua in modo corretto il proprio compito.

Domande di verifica

1. Qual è la sintassi dell'istruzione `if`?
2. Il ramo `else`, in un costrutto `if`, è obbligatorio?

3. Realizzare un programma di propria fantasia che utilizzi correttamente l'istruzione `if`.
4. Esistono variabili logiche in Java?
5. Fare un esempio in cui è necessario utilizzare più istruzioni `if` annidate.
6. Come si può fare in modo che più istruzioni semplici costituiscano un blocco (istruzione composta)?
7. Realizzare degli esempi di istruzioni `if` che dimostrino chiaramente le necessità dei blocchi.
8. Quale possibilità offre il costrutto `switch-case`? Predisporre almeno tre esempi di programmi in cui risulti chiara la sua utilità.
9. Quando è possibile utilizzare uno `switch-case` al posto di una serie di istruzioni `if` in cascata?
10. Quali possibilità si hanno a disposizione per diramare il flusso di esecuzione di un programma? Perché queste possibilità rientrano pienamente nelle regole della programmazione strutturata?
11. Quali sono gli operatori relazionali, e qual è il loro significato?
12. A quale scopo vengono utilizzati gli operatori logici `&&`, `||` e `!`?
13. Gli operatori relazionali possono essere utilizzati congiuntamente agli operatori logici e a quelli aritmetici in una stessa espressione?
14. Con quali altri operatori può essere utilizzato in una stessa espressione l'operatore di assegnamento?
15. Perché è stato scelto di indicare con `==` l'operatore di uguaglianza, anziché con il semplice `=`?
16. Indicare l'esatta gerarchia degli operatori presentati in questo capitolo.
17. Qual è l'associatività dell'operatore di assegnamento? È la stessa di qualcuno degli altri operatori presentati nel capitolo?
18. Predisporre un programma che utilizzi un espressione condizionale.
19. Da cosa può essere sostituita un'espressione condizionale?
20. Trovare degli esempi in cui la presenza o meno di una coppia di parentesi, pur mantenendo la correttezza sintattica dell'espressione, ne modifichi il valore restituito.
21. Oltre che per modificare la sequenza di valutazione di un'espressione, le parentesi possono essere utili per qualche altra ragione?
22. A quale scopo vengono usate le asserzioni?

Esercizi

1. Scrivere un programma che, dati due valori interi qualsiasi, segnali se i due numeri sono uguali; in caso contrario indichi il minore e il maggiore.

2. Scrivere un programma che dati quattro interi ne determini il maggiore.

3. Un negoziante per ogni spesa di importo superiore a € 100 effettua uno sconto del 5%, del 10% per ogni spesa superiore a € 300. Scrivere un programma che dato l'ammontare della spesa visualizzi l'importo effettivo da pagare.

4. Se le variabili intere `a`, `b` e `c` hanno rispettivamente valore 5, 35 e 7; quale valore viene assegnato alle variabili `ris` e `risLog` rispettivamente di tipo `int` e `boolean` dalle seguenti assegnazioni?

   a) `ris = a+b-c`

   b) `risLog = a>b`

   c) `ris = a/b`

   d) `risLog = a/b > c`

   e) `risLog = (a>b) | (a<c) | (c==b)`

   Scrivere un programma che verifichi le risposte date.

5. Se le variabili intere `a`, `b` e `c` hanno gli stessi valori dell'esercizio precedente, le seguenti espressioni restituirebbero vero o falso?

   a) `(a>b) | (c>a)`

   b) `(c>a) & (a>b)`

   c) `!(a>b) & (c>a)`

   d) `!(a>b) | !(c>a)`

   e) `(a==c) | (a<b) & (b<c)`

   f) `(a!=c) | (a<b) & (b<c)`

   Scrivere un programma che verifichi le risposte date.

6. Supponendo che le variabili intere `x` e `y` abbiano rispettivamente valore 12 e 45 e che le variabili carattere `a` e `b` abbiano valore `t` e `T`; le seguenti espressioni restituirebbero vero o falso?

   a) `(x>y) | (a!=b)`

   b) `(y>x) & (a==b)`

   c) `(a!=b) & !(x>y)`

   d) `true | (y<x)`

   e) `b == 't'`

   Scrivere un programma che verifichi le risposte date.

7. Scrivere un programma che dato il valore di una variabile `char`, stabilisca se si tratta di una lettera maiuscola e, in caso positivo, segnali se è una vocale o una consonante.

8. Scrivere un programma che memorizzi nelle variabili `a`, `b` e `c` tre valori interi e visualizzi:

- il volume del parallelepipedo di lati `a`, `b` e `c` se il valore di `a` al quadrato sommato a `b` è diverso da `c`;
- la somma di `a`, `b` e `c`, altrimenti.

9. Scrivere un programma che dato il numero MM, che rappresenta il valore numerico di un mese, visualizzi, se $1 \leq MM \leq 12$, il suo nome per esteso altrimenti la frase "`Valore numerico non valido`".

10. Scrivere un programma che dato il numero AA, che rappresenta un anno, verifichi se questo è bisestile.

11. Scrivere un programma che dati i numeri GG, MM, AA di una data verifichi se questa è valida.

12. Scrivere il programma che dati sei numeri che rappresentano due date nel formato GG, MM, AA determini la più recente.

13. Su una linea ferroviaria rispetto alla tariffa piena gli utenti pensionati usufruiscono di uno sconto del 10%, gli studenti del 15% e i disoccupati del 25%. Codificando i pensionati con una `P`, gli studenti con una `S` e i disoccupati con una `D`, scrivere un programma che, dati il costo di un biglietto e l'eventuale condizione particolare dell'utente, visualizzi l'importo da pagare.

14. Scrivere un programma che dati tre numeri interi, a seconda dei casi visualizzi una delle seguenti risposte:

```
Tutti uguali
Due uguali e uno diverso
Tutti diversi
```

15. Scrivere un programma che lette le misure dei lati di un triangolo visualizzi se il triangolo è equilatero, isoscele o scaleno.

16. Utilizzando l'espressione condizionale `? :` scrivere un programma che, dati tre valori interi memorizzati nelle variabili `a`, `b` e `c`, assegni a `d`:

- il volume del parallelepipedo di lati `a`, `b` e `c` se il valore di `a` al quadrato sommato a `b` è diverso da `c`;
- la somma di `a`, `b` e `c`, altrimenti.

17. Scrivere un programma che richieda in ingresso tre valori interi distinti e ne determini il maggiore facendo uso dell'espressione condizionale "`? :`".

18. Scrivere un programma che visualizzi il seguente menu:

```
        MENU DI PROVA
1) Per immettere dati
2) Per determinare il maggiore
3) Per determinare il minore
4) Per ordinare
5) Per visualizzare
Scelta: _
```

quindi attenda l'immissione di un intero da parte dell'utente e visualizzi una scritta corrispondente alla scelta effettuata, del tipo: "`In esecuzione l'opzione 1`". Se la scelta non è tra quelle proposte (`1, 2, 3, 4, 5`) deve venire visualizzata la scritta: "`Opzione inesistente`". Si utilizzi il costrutto `switch-case`.

19. Scrivere un programma che visualizzi la rappresentazione binaria di una variabile di tipo `byte` usando l'operatore `&`.

20. Scrivere un programma che, data una variabile di tipo `byte`, metta a 1 il quarto bit da destra senza modificare gli altri. Usare il codice dell'esercizio 16 per verificare il risultato.

21. Come il precedente ma il quarto bit deve essere messo a 0.

22. Come il precedente ma il quarto bit deve essere messo a 0 se vale 1 e viceversa.

Capitolo 5

Strutture di controllo iterative

Obiettivi didattici

- Strutture di controllo iterative
- `while`, `do-while`, `for`
- Cicli annidati, incrementi e decrementi, operatore virgola
- Approfondire ed estendere l'uso di espressioni, operatori e blocchi d'istruzioni
- Calcolo del fattoriale
- Interruzioni `break`, `continue`

5.1 Istruzione `while`

Quando si desidera ripetere una o più istruzioni in base a certe condizioni, o per un numero definito di volte, si può creare un *ciclo* (*loop*). Il modo più semplice per realizzarlo è mediante l'istruzione `while` che, dal punto di vista sintattico, è del tutto analoga all'istruzione `if` senza la clausola `else`. L'unica differenza consiste nel fatto che l'istruzione, semplice o composta, che la segue viene eseguita fintanto che l'espressione booleana tra parentesi tonde restituisce un valore `true`. La sintassi dell'istruzione `while` è

```
while (valore-booleano) istruzione;
```

Supponiamo di voler trovare il numero intero x più piccolo che sia divisibile per 7 e che sia maggiore di 30

```
int x = 0;
while (x < 30)
   x = x + 7;
```

Nell'esempio viene valutato il valore booleano tra parentesi tonde; se esso risulta `true` viene eseguito l'incremento di x e il controllo del programma ritorna alla valutazione della condizione, altrimenti l'istruzione con l'incremento viene ignorata e il controllo passa all'istruzione successiva. In questo esempio, nonostante sia estremamente semplice, si possono distinguere diverse parti che caratterizzano un ciclo. La prima parte è l'*inizializzazione* della variabile che condiziona il ciclo, che nel nostro caso corrisponde all'assegnazione di 0 alla variabile x. Possono esserci più variabili che condizionano il ciclo, oppure anche nessuna, come vedremo in seguito. L'inizializzazione della o delle variabili che condizionano il ciclo non è obbligatoria, ma è usata frequentemente.

In un ciclo `while` deve esserci sempre invece una *condizione d'uscita*, che nel nostro caso corrisponde all'espressione logica `x<30`. La condizione d'uscita può essere complessa quanto si vuole, oppure può essere la costante `false`, nel qual caso il ciclo non viene mai eseguito, oppure la costante `true`, nel qual caso si ha un ciclo infinito. Si ha infine il *corpo del ciclo*, che nel nostro caso corrisponde all'istruzione `x=x+7`. Come detto, il corpo può essere anche un blocco, oppure un'istruzione nulla. L'istruzione nulla è formata da un semplice punto e virgola (`;`) e ovviamente non effettua nessuna elaborazione. Tipicamente nel corpo del ciclo è inserita un'istruzione che varia almeno una delle variabili che condizionano il ciclo, altrimenti ritorniamo nel caso del ciclo infinito. Vediamo però un caso particolare

```
int x = -7;
while ((x = x + 7) < 30)
   ;
```

Questo esempio esegue lo stesso tipo di elaborazione del precedente. La valutazione della condizione di uscita provoca anche l'incremento di x, per cui il corpo del ciclo può essere un'istruzione nulla. La variabile x è stata inizializzata a -7 affinché il corpo del ciclo venga eseguito lo stesso numero di volte dell'esempio precedente. Se x fosse stata inizializzata a 0, al primo passaggio sarebbero confrontati 7 e 30, per cui il ciclo sarebbe stato eseguito una volta in meno, anche se poi il valore di x all'uscita sarebbe risultato il medesimo.

Facciamo ora un esempio più ampio e supponiamo di voler calcolare il fattoriale di un intero n. Il fattoriale di un numero n, indicato con `n!`, è definito nel modo seguente

```
n! = n * (n-1) * (n-2) * (n-3) ... * 2 * 1
```

dove 1! e 0! sono per definizione uguali a 1. Avremo, per esempio

```
4! = 4*3*2*1 = 24
6! = 6*5*4*3*2*1 = 720
```

Un programma per il calcolo del fattoriale usando il `while` può essere il seguente

```
class Fattoriale1  {
   public static void main (String argv[]) {
      int n = 5;
      int fat = 1;
      int i = n;
      while (i > 1) {
         fat = fat * i;
         i = i - 1;
      }
      System.out.print ("Il fattoriale di ");
      System.out.print (n);
      System.out.print ("  è  ");
      System.out.println (fat);
   }
}
```

Il programma calcola il fattoriale del numero contenuto nella variabile `n`; nel caso illustrato in cui essa vale 5, la visualizzazione sarà la seguente

```
Il fattoriale di 5 è 120
```

È importante sottolineare che se `n` è minore o uguale a 1, il corpo del `while` non viene mai eseguito, per cui `fat` vale 1 come da definizione.

5.2 Istruzione `do-while`

Nel paragrafo precedente, abbiamo visto che il corpo di un `while` può non essere mai eseguito quando la condizione risulta falsa fin dall'inizio. In alcune situazioni al contrario è comodo avere cicli che vengano eseguiti perlomeno una volta. In tali casi può essere utilizzato il costrutto `do-while` la cui sintassi è

```
do istruzione while (valore-booleano);
```

Supponiamo di voler visualizzare un numero positivo `n` con le cifre in ordine inverso, vale a dire con la cifra più significativa a destra e quella meno significativa a sinistra. Possiamo allora utilizzare il seguente codice

```
int n = 123456;
do {
   System.out.print (n % 10);
   n = n / 10;
   } while (n > 0);
System.out.println ("");
```

che produce il seguente risultato

```
654321
```

Nell'esempio è utile che il ciclo venga eseguito almeno una volta per gestire il caso particolare in cui n sia uguale a 0. Utilizzando un `while` il ciclo non verrebbe mai eseguito per cui non verrebbe visualizzato nulla; con il `do-while` invece viene visualizzato uno 0 come atteso.

L'algoritmo per il calcolo del fattoriale utilizzando questo costrutto potrebbe essere il seguente

```
class Fattoriale2  {
   public static void main (String argv[]) {
      int n = 5;
      int fat = 1;
      int i = 1;
      do {
         fat = fat * i;
         i = i + 1;
      } while (i <= n);
      System.out.print ("Il fattoriale di ");
      System.out.print (n);
      System.out.print ("  è  ");
      System.out.println (fat);
   }
}
```

Nell'esempio precedente il corpo del ciclo viene eseguito almeno una volta. La prima volta però non modifica in nessun caso il valore di `fat` in quanto `i` è uguale a 1, perciò, in effetti, viene fatta un'iterazione inutilmente. In ogni modo l'algoritmo funziona correttamente e fornisce gli stessi risultati del calcolo fattoriale precedente.

5.3 Istruzione `for`

L'istruzione `for` può essere considerata come una versione estesa dell'istruzione `while` dove la fase di inizializzazione e la fase di modifica della variabile o delle variabili che controllano il ciclo, sono espresse esplicitamente come parte dell'istruzione. La sintassi dell'istruzione `for` è

```
for (inizializzazione; valore-booleano; incremento) istruzione;
```

Come si vede, tra le parentesi tonde che seguono il `for`, sono presenti tre istruzioni, che però non possono essere blocchi. La prima, *istruzione di inizializzazione*, viene eseguita una sola volta prima di entrare nel ciclo; la seconda, *condizione d'uscita*, viene valutata prima di ogni iterazione, analogamente alla condizione di uscita del `while`; mentre la terza, *istruzione d'incremento*, viene eseguita al termine del corpo del ciclo, giusto prima di una nuova valutazione della condizione d'uscita. Supponiamo, per esempio, di voler scrivere un programma che visualizza i numeri dall'1 al 10. Il codice potrebbe essere il seguente:

```
for (x = 1; x <= 10; x = x + 1)
    System.out.println (x);
```

Le tre istruzioni all'interno delle parentesi del `for` possono essere anche nulle, perciò si può ottenere lo stesso comportamento di un `while` mettendo la prima e la terza istruzione nulla. Le seguenti istruzioni sono, per esempio, perfettamente equivalenti

```
for ( ; x <= 10; )        while (x <= 10)
```

Se la condizione di uscita, corrispondente alla seconda istruzione, è nulla, allora il ciclo è infinito. Si ha quindi l'equivalenza, per esempio, delle seguenti istruzioni

```
for ( ; ; )               while (true)
```

Nell'istruzione d'inizializzazione si può mettere anche la dichiarazione di una variabile: essa sarà visibile però solo nel corpo del ciclo. Nell'esempio seguente

```
for (int x = 1; x <= 10; x = x + 1)
    System.out.println (x);
System.out.println (x);              // Errore in compilazione!
```

l'ultima riga provoca un errore in fase di compilazione in quanto si tenta di visualizzare una variabile fuori dal ciclo in cui è stata dichiarata. Togliendo tale riga, il codice viene compilato senza problemi.

Spesso capita di avere dei cicli dove si deve inizializzare e modificare più di una variabile. Per permettere questo, le istruzioni d'inizializzazione e di incremento di un `for` possono essere formate da più istruzioni separate da una virgola (`,`). Vediamo come si può realizzare il calcolo del fattoriale utilizzando questo costrutto

```
class Fattoriale3  {
   public static void main (String argv[]) {
      int n = 5;
      int fat;
      int i;
      for (i = n, fat = 1; i > 1; i = i - 1)
         fat = fat * i;
      System.out.print ("Il fattoriale di ");
      System.out.print (n);
      System.out.print ("  è  ");
      System.out.println (fat);
   }
}
```

L'operatore virgola non può essere usato per comporre istruzioni al di fuori del `for`, eccetto che per le dichiarazioni, come abbiamo già visto. Nel seguente esempio

```
for (int i = 0, j = 0, n = 10;
     i < n;
     i = i + 1, j = j + 1) {...
```

l'istruzione d'inizializzazione è la dichiarazione con inizializzazione delle variabili `i`, `j` e `n`, che è corretta, come visto, anche al di fuori di un ciclo `for`.

Nell'esempio osserviamo che le variabili `i` , `j` e `n` non sono visibili al di fuori del ciclo `for`.

5.4 Cicli annidati

Quando un ciclo viene eseguito dall'interno di un altro ciclo, si parla di cicli *annidati*. In particolare si definisce *interno* il ciclo contenuto ed *esterno* il ciclo contenitore. Non c'è un limite al numero di annidamenti in Java, perciò è possibile che un ciclo sia interno rispetto a un altro ed esterno rispetto a un terzo. Per vedere un esempio pratico, supponiamo di voler visualizzare un quadrato formato da simboli + le cui dimensioni sono contenute in due variabili, `altezza` e `larghezza`. Il codice seguente

```
int altezza = 12, larghezza = 40;

for (int i = 0; i < altezza; i++) {
   for (int j = 0; j < larghezza; j++)
   System.out.print("+");
   System.out.println("");
}
```

produce

```
++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++
```

Si può notare come in questo esempio il ciclo controllato dalla variabile `i` sia il più esterno, mentre quello controllato dalla variabile `j` sia il più interno. L'istruzione `System.out.println("")` viene utilizzata per andare a riga nuova.

✔ **NOTA**

Ora che abbiamo sperimentato l'uso di strutture di controllo, blocchi d'istruzioni e variabili possiamo intuire la veridicità del teorema di Jacopini-Böhm, introdotto nel Capitolo 2 a proposito della programmazione strutturata, accennando alle idee che stanno alla base della dimostrazione. Ricordiamo che si vuol dimostrare che qualsiasi algoritmo è riscrivibile utilizzando sequenza, selezione e iterazione, in altre parole eliminando i salti indietro e in avanti. Possiamo eliminare un salto verso un'istruzione precedente riscrivendo di nuovo le istruzioni necessarie. Possiamo eliminare un salto a un'istruzione successiva continuando la sequenza del flusso normale senza però eseguire le istruzioni fino a che non si giunge al punto

di arrivo del salto. Si può ottenere ciò utilizzando una variabile di appoggio cui si assegnino valori diversi per il flusso normale e per quello saltante. Se ne controlla il valore: se la variabile indica la necessità del salto non si eseguono le istruzioni seguenti fino al punto di arrivo del salto stesso. Il programma strutturato potrebbe risultare più lungo ma certamente anche più leggibile, cosa che sappiamo essere di gran lunga il fattore più importante.

5.5 Incrementi e decrementi

Trattiamo in questo paragrafo gli operatori di autoincremento e autodecremento citati nel capitolo precedente, dato che l'argomento si ricollega molto bene alle istruzioni iterative, anche se il loro uso può essere generale. L'incremento unitario del valore di una variabile è una delle operazioni più frequenti:

```
x = x + 1;
```

In Java è possibile ottenere lo stesso effetto mediante l'operatore ++, costituito da due segni di addizione non separati da nessuno spazio. L'istruzione

```
++x;
```

incrementa di uno il valore di x, esattamente come faceva l'istruzione precedente. Lo stesso ragionamento vale per l'operazione di decremento, perciò

```
x = x - 1;
```

è equivalente a

```
--x;
```

L'operatore --, costituito da due segni di sottrazione, decrementa la variabile di una unità. Dunque anche l'istruzione `for`

```
for (x = 1; x <= 10; x = x + 1)
```

può essere trasformata in

```
for (x = 1; x <= 10; ++x)
```

Si noti come il codice Java diventi via via più compatto. Gli operatori ++ e -- possono precedere una variabile in una espressione.

```
int a, b, c;
a = 5;
b = 7;
c = ++a + b;
System.out.println (a);
System.out.println (b);
System.out.println (c);
```

Nell'espressione ++a+b, la variabile a viene incrementata di un'unità (++a) e sommata alla variabile b. Successivamente il risultato viene assegnato a c. Le tre println visualizzeranno rispettivamente 6, 7 e 13.

Gli operatori ++ e -- hanno priorità maggiore degli operatori binari aritmetici, relazionali e logici, come risulta osservando la Tabella 3.1.

Questi operatori possono tanto precedere che seguire una variabile:

```
++x;
x++;
--x;
x--;
```

La differenza di comportamento tra gli operatori preposti e posposti è avvertibile solo quando sono usati in espressioni ed è la seguente:

- se l'operatore ++ (--) precede la variabile, prima il valore della variabile viene incrementato (decrementato) poi viene valutata l'intera espressione.
- se l'operatore ++ (--) segue la variabile, prima viene valutata l'intera espressione, poi il valore della variabile viene incrementato (decrementato).

Per esempio

```
int a, b, c;
a = 5;
b = 7;
c = a++ + b;
System.out.println (a);
System.out.println (b);
System.out.println (c);
```

non produce la stessa visualizzazione della sequenza precedente. La variabile a viene sommata a b e il risultato viene assegnato a c; successivamente a viene incrementata di un'unità. Le istruzioni println visualizzeranno rispettivamente 6, 7 e 12. Si osservi che i seguenti cicli for sono identici

```
for (x = 1; x <= 3; ++x)        for (x = 1; x <= 3; x++)
```

poiché l'istruzione di incremento è da considerarsi a sé stante. Viceversa

```
for (x = 1; ++x <= 3; )        for (x = 1; x++ <= 3; )
```

sono diversi in quanto nel caso di sinistra x viene incrementata prima della valutazione della condizione d'uscita, per cui nel primo ciclo x prende il valore 2, nel secondo 3 e il terzo ciclo non verrà mai eseguito dato che x prende il valore 4. Nel caso di destra x assume il valore 4 solamente dopo il confronto operato nel terzo ciclo che quindi verrà portato a termine. L'esempio seguente

```
System.out.println ("++x");
for (int x = 1; ++x <= 3; )
   System.out.println (x);
System.out.println ("x++");
```

```
for (int x = 1; x++ <= 3; )
   System.out.println (x);
```

produce la seguente visualizzazione

```
++x
2
3
x++
2
3
4
```

Notare come la variabile x sia stata dichiarata all'interno sia nel primo che nel secondo `for`.

Nel caso che una variabile debba essere incrementata o decrementata di un valore diverso da uno, oltre che con il metodo classico

```
x = x + 9;
```

si può usufruire dell'operatore `+=`. Nell'esempio

```
x += 9;
```

la variabile x viene incrementata di 9. Questo tipo di *assegnamento compatto*, oltre a rendere più conciso e leggibile il codice, è anche più efficiente in quanto la variabile viene valutata una volta soltanto. Esiste un operatore d'assegnamento compatto per ogni operatore aritmetico binario. La lista completa degli operatori è riportata in Tabella 5.1.

Tabella 5.1 Scala di priorità di tutti gli operatori

massima precedenza														
						( )								
	!		- (unario)			~			++			--		
			*			/			%					
					+			-						
			<<			>>			>>>					
		<			>			<=			>=			
					==			!=						
						&								
						^								
						|								
						&&								
						||								
						?:								
=	+=	-=	*=	/=	%=	&=	|=	^=	<<=	>>=	>>>=			
minima precedenza														

Tutti gli operatori d'assegnamento compatti hanno la medesima precedenza dell'operatore di assegnamento (=). Vediamo alcuni esempi d'assegnamenti compatti con l'equivalente forma classica.

```
FORMA COMPATTA          FORMA CLASSICA
    a *= 5;                 a = a * 5;
    a -= b;                 a = a - b;
    a *= 4 + b;             a = a * (4 + b);
```

L'ultima riga evidenzia la bassa precedenza degli operatori di assegnamento compatti.

5.6 Istruzioni `break` e `continue`

L'istruzione `break`, come abbiamo visto, consente di interrompere l'esecuzione del `case`, provocando un salto del flusso di esecuzione alla prima istruzione successiva. Una seconda possibilità d'uso di quest'istruzione è quella di forzare la terminazione di un'iterazione `while`, `do-while` o `for`, provocando un salto alla prima istruzione successiva al ciclo. L'istruzione `break` viene utilizzata in quei casi in cui risulta conveniente controllare la condizione d'uscita in momenti differenti dall'inizio o dalla fine del ciclo. Supponiamo per esempio di voler visualizzare le somme ottenute accumulando i 5 valori immessi dall'utente.

```
import java.util.Scanner;
class EsempioBreak  {
   public static void main (String argv[]) {
      int numero = 0, somma = 0;
      Scanner in = new Scanner(System.in);
      for(int i = 1; ; ) {
         System.out.print ("Inser. intero: ");
         numero =  in.nextInt();
         somma += numero;
         System.out.print (somma);
         if(i++ <= 5)
            System.out.print (",    ");
         else
            break;
      }
   }
}
```

Inserendo i valori 3, 7, 1, 9 e 2 l'esecuzione ha il seguente output.

```
Inser. Intero: 3
3,   Inser. Intero: 7
10,   Inser. Intero: 1
11,   Inser. Intero: 9
20,   Inser. Intero: 2
22
```

In quest'esempio abbiamo utilizzato un ciclo infinito (condizione d'uscita nulla), in quanto all'interno è necessario sapere se c'è un altro valore da visualizzare per evitare di mettere una virgola di troppo in fondo alla visualizzazione, dopo 22 nell'esempio.

Lo stesso risultato dell'esempio precedente sarebbe stato ottenuto anche con il codice seguente

```
for(int i = 1; i <= 5; ) {
    System.out.print ("Inser. intero: ");
    numero =  in.nextInt();
    somma += numero;
    System.out.print (somma);
    if(++i <= 5)
        System.out.print (", ");
}
```

che però valuta due volte per ogni ciclo la stessa condizione di uscita.

L'istruzione `continue` funziona in modo simile all'istruzione `break`. Entrambe, infatti, interrompono un ciclo, evitando l'esecuzione di tutte le istruzioni fino al termine di esso ma, mentre `break` causa anche l'uscita dal ciclo, `continue` provoca solo il passaggio immediato alla successiva iterazione. Quest'ultima istruzione si usa per non eseguire una parte del corpo ciclo in determinate condizioni. Per esempio supponiamo di voler sommare i numeri immessi dall'utente. Il codice

```
import java.util.Scanner;
class Esempiocontinue  {
    public static void main (String argv[]) {
        int somma = 0, numero = 0;
        Scanner in = new Scanner(System.in);
        for (int i = 1; i <= 5; i++) {
            System.out.print ("Inser. intero: ");
            numero =  in.nextInt();
            if (numero <= 0)
                continue;
            somma += numero;
        }
        System.out.println (somma);
    }
}
```

produce il seguente output, inserendo i valori 4, -10, 5, -3 e 4:

```
Inser. Intero: 4
Inser. Intero: -10
Inser. Intero: 5
Inser. Intero: -3
Inser. Intero: 4
13
```

Va sottolineato che all'interno di uno `switch` inserito in un ciclo, l'istruzione `break` interrompe il flusso del `case` mentre `continue` provoca la successiva iterazione del ciclo.

Nei cicli annidati le istruzioni `break` e `continue` causano l'interruzione solo del ciclo più interno. In alcuni casi però può essere comodo interrompere anche dei cicli esterni oppure interrompere un ciclo dall'interno di un `case`. In Java non esiste un'istruzione che permetta di effettuare salti arbitrari (`goto`), per cui, per venire incontro ai programmatori, è stata ampliata la funzionalità delle istruzioni `break` e `continue` che permettono così di interrompere l'esecuzione di un'istruzione qualsiasi che li contiene, purché quest'ultima sia identificata da un nome. Per *nominare* un'istruzione, si posiziona il nome prima dell'inizio dell'istruzione stessa seguito dal carattere due punti (`:`). Il nome dovrà essere formato secondo le stesse regole che valgono per i nomi delle variabili e potrà essere utilizzato da `break` e `continue` per individuare l'istruzione che si intende interrompere.

Un'istruzione `break` seguita da un nome può essere utilizzata anche fuori dei cicli e dei `case`, mentre un'istruzione `continue` può essere usata solo nei cicli poiché altrimenti non avrebbe senso. Il nome di un'istruzione è visibile solo all'interno dell'istruzione stessa. Per esempio, supponiamo di voler sommare i valori immessi dall'utente fintanto che non si è inserito uno dei seguenti valori: 1, 7, 10, 50 o 100.

```
import java.util.Scanner;
class EsempioNomina  {
   public static void main (String argv[]) {
      int somma = 0, numero = 0;
      Scanner in = new Scanner(System.in);
interruzione:
      for (; ; ) {
         System.out.print ("Inser. intero: ");
         numero =  in.nextInt();
         switch (numero) {
            case 1:
            case 7:
            case 10:
            case 50:
            case 100:
               break interruzione;
            default:
               break;
         }
         somma += numero;
      }
      System.out.println (somma);
   }
}
```

Vediamo un esempio di esecuzione.

```
Inser. Intero: 2
Inser. Intero: 6
Inser. Intero: 99
Inser. Intero: 999
Inser. Intero: 222
Inser. Intero: 50
1328
```

Si può nominare anche un blocco. Modifichiamo il codice di esempio nel modo seguente

```
for (int i=1; i<=5; i++) interruzione: {
   System.out.print ("Inser. intero: ");
   numero = in.nextInt();
   switch (numero) {
      case 1:
      case 7:
      case 10:
      case 50:
      case 100:
         break interruzione;
      default:
         break;
   }
   somma += numero;
}
System.out.println (somma);
```

e osserviamo un esempio di esecuzione:

```
Inser. Intero: 100
Inser. Intero: 1
Inser. Intero: 2
Inser. Intero: 99
Inser. Intero: 3
104
```

L'interruzione del blocco non causa l'uscita dal `for` e quindi si comporta di fatto come un `continue`. Se sostituiamo il `break interruzione;` con un semplice `break;` otteniamo la somma dei valori immessi diversi da 1, 7, 10, 50, 100; al contrario del programma precedente l'elaborazione non si ferma quando trova uno di questi valori, ma prosegue fino al numero di immissioni prestabilite, nell'esempio 5.

Per vedere un utilizzo dell'istruzione `continue` con nome, supponiamo di voler visualizzare un triangolo formato da simboli + la cui altezza è contenuta in una variabile `altezza`. Il codice seguente

```
   int altezza = 12;
esterno:
```

```
for (int i = 0; i < altezza; i++) {
   for (int j = 0; j < altezza; j++)  {
      System.out.print("+");
      if (j >= i) {
         System.out.println("");
         continue esterno;
      }
   }
}
```

produce

```
+
++
+++
++++
+++++
++++++
+++++++
++++++++
+++++++++
++++++++++
+++++++++++
++++++++++++
```

Domande di verifica

1. Quali sono le sintassi delle istruzioni `while`, `do-while` e `for`?
2. Quando è più conveniente utilizzare il costrutto `do-while` rispetto al `while`?
3. È sempre possibile rimpiazzare un'istruzione `while` con un analogo `for`? Giustificare la risposta.
4. Scrivere un programma che ripeta un blocco d'istruzioni. Realizzarne una versione per ogni costrutto di controllo iterativo conosciuto. Fare in modo che il numero di ripetizioni non sia deciso a priori.
5. Quali sono e che significato hanno gli operatori di autoincremento e autodecremento?
6. Come si inseriscono `++` e `--` all'interno della gerarchia di priorità di tutti gli operatori esaminati?
7. Predisporre un esempio in cui è necessario utilizzare più strutture di controllo iterative annidate.
8. Perché i costrutti che si hanno a disposizione per ripetere l'esecuzione di una parte di programma rientrano nelle regole della programmazione strutturata?

9. Perché non è consigliabile fare un uso estensivo di `break` e `continue`?

10. Fare un esempio di ciclo infinito `while` in cui non sia così immediato accorgersi che l'iterazione non si fermerà mai.

Esercizi

1. Scrivere un programma che visualizzi il valore di tutte le prime n potenze di 2.

2. Scrivere un programma che visualizzi il quadrato dei primi 24 numeri naturali.

3. Predisporre un programma che stampi un rettangolo la cui cornice sia costituita da caratteri asterisco e la parte interna da caratteri `Q`. Il numero di linee e di colonne del rettangolo viene deciso a tempo di esecuzione dall'utente; per esempio, se il numero delle linee è uguale a 5 e il numero di colonne a 21, sul video deve apparire:

```
**********************
*QQQQQQQQQQQQQQQQQQQ*
*QQQQQQQQQQQQQQQQQQQ*
*QQQQQQQQQQQQQQQQQQQ*
**********************
```

4. Scrivere un programma che date n coppie di numeri interi, calcoli la somma di ogni coppia e la visualizzi.

5. Scrivere un programma che determini il maggiore e il minore fra gli n numeri immessi in ingresso.

6. Predisporre un programma che determini il maggiore, il minore e la media degli n valori immessi dall'utente.

7. Scrivere un programma che dato un numero m calcoli la somma

$$\sum_{k=1}^{m} \frac{1}{k}$$

dei primi m elementi della serie armonica.

8. Scrivere un programma che dati i numeri interi $n1$ e $n2$ calcoli il prodotto di tutti i numeri compresi fra $n1$ e $n2$. Considerare anche il caso in cui $n1 > n2$.

9. Scrivere un programma che dato un numero intero visualizzi tutti i suoi divisori.

10. Scrivere un programma che dato un numero intero stabilisca se questo è primo.

11. Scrivere un programma che dato un numero intero k calcoli la somma di tutti i multipli di 8 minori di k.

12. Scrivere un programma che visualizzi la tabella pitagorica.

13. Realizzare un programma che richieda all'utente *n* interi e visualizzi il numero di volte in cui sono stati eventualmente immessi i valori 10, 100 e 1000.

14. Predisporre i programmi completi in linguaggio Java degli esercizi riguardanti i cicli che nel Capitolo 2 erano richiesti in pseudolinguaggio.

Capitolo 6

Array, ricerche, ordinamenti

Obiettivi didattici

- Array mono e multidimensionali
- Indici di vettori e matrici
- Approfondire l'uso di condizioni logiche e cicli annidati
- Inizializzare gli array
- Ricerca di massimo, minimo, media e *prodotto di matrici*
- Implementare algoritmi di ricerca, ordinamento e fusione

6.1 Array monodimensionali

Un *array monodimensionale*, o *vettore* di *lunghezza l*, consiste in un insieme di *l* variabili dello stesso tipo, caratterizzate dallo stesso nome e distinguibili l'una dall'altra per mezzo di un *indice*, cioè un numero intero progressivo, che può assumere valori compresi tra 0 e *l*-1. Possiamo pensare agli array come a un tipo che può essere derivato da un qualsiasi tipo, sia esso primitivo o definito dal programmatore. Un array di variabili di tipo *T* è quindi un nuovo tipo, che si denota con *T[]*. Un array ha una dimensione fissa che non influenza il tipo, ciò significa che un array di 10 `int` è dello stesso tipo di un array di 20 `int`. Per fare un esempio, supponiamo di voler memorizzare i millimetri di pioggia giornalieri caduti nel mese di Gennaio; potremmo dichiarare un array di interi nel modo seguente

```
int mmGennaio[];
```

Come si vede, non abbiamo dichiarato la dimensione dell'array, perché in Java una variabile di tipo array è solo una *referenza* all'insieme di variabili e di per sé non contiene nulla. Per fare un paragone pratico, una referenza è come un assegno bancario, che in bianco non vale nulla e una volta riempito rappresenta in tutto e per tutto del denaro che però risiede in banca. Una variabile array appena dichiarata è come un assegno in bianco, non ha dimensioni e l'unica cosa che conosciamo su di essa è cosa può contenere.

In Java una referenza non ancora riempita, cioè non collegata ancora a nessun dato, contiene il valore convenzionale `null`. Questa parola riservata la ritroveremo in seguito, ed è una costante che indica appunto una referenza che non referenzia alcunché. Per utilizzare l'array è quindi necessario creare l'insieme di variabili e collegarlo alla variabile array; l'operatore speciale `new` alloca lo spazio necessario ed effettua il collegamento. Per utilizzare l'operatore d'allocazione dobbiamo specificare un tipo e un numero intero positivo racchiuso tra parentesi quadre che indica la lunghezza dell'array. L'esempio seguente alloca 31 interi

```
mmGennaio = new int[31];
```

Java controlla che i tipi siano rispettati, per cui non è possibile assegnare a `mmGennaio` l'allocazione di un array di tipo diverso da `int`, pena un errore in fase di compilazione.

L'operatore `new` effettua anche un'inizializzazione delle variabili, assegnando 0 alle variabili numeriche, compresi i `char`, e `false` alle variabili `boolean`. Per riferirsi a ogni singolo elemento dell'array, si utilizzano ancora le parentesi quadre. Per assegnare al quarto elemento dell'array il valore 7, si procede nel modo seguente

```
mmGennaio[3] = 7;
```

Va sottolineato il fatto che il quarto elemento ha indice 3 poiché il primo elemento, come detto, ha indice 0. Se si cerca di utilizzare un indice al di fuori dell'intervallo permesso, viene segnalato un errore in fase di esecuzione. Il seguente esempio

```
mmGennaio[31] = 0;
```

produce un errore in fase di esecuzione.

Una variabile array può essere *inizializzata* automaticamente in fase di dichiarazione in modo analogo a quanto già visto per i tipi primitivi. Per fare questo è sufficiente dichiarare l'insieme di valori da assegnare a ciascun elemento, racchiusi tra parentesi graffe e separati da virgole. Questa operazione alloca anche lo spazio per gli elementi e quindi evita l'uso dell'operatore `new`. Nell'esempio seguente

```
int serieDiFibonacci[] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55};
```

la variabile array `serieDiFibonacci` ha 10 elementi, individuabili da indici compresi tra 0 e 9, ognuno dei quali contiene un intero.

Una variabile array può essere assegnata ad un'altra dello stesso tipo, ma è importante tenere a mente che una variabile array è solo una referenza che consente di utilizzare delle variabili allocate dall'interprete; perciò dopo un tale assegnamento si hanno due array che permettono l'accesso a un unico insieme di variabili. Nell'esempio seguente

```
int mmGennaio[] = new int[31];
int serieDiFibonacci[] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55};
mmGennaio[3] = 7;
mmGennaio = serieDiFibonacci;
System.out.println (mmGennaio[3]);
mmGennaio[10] = 1;                    // Errore in esecuzione !!
```

l'assegnazione sulla quarta riga fa in modo che entrambe le variabili `mmGennaio` e `serieDiFibonacci` referenzino la stessa area di memoria. La quinta riga visualizza quindi il valore 3, quarto elemento della serie di Fibonacci, mentre la sesta riga provoca un errore in fase di esecuzione per accesso di un elemento fuori dai limiti consentiti, in quanto la variabile `mmGennaio` referenzia ora un array di soli 10 elementi.

Nell'esempio appena visto la dichiarazione dell'array `serieDiFibonacci` è servita unicamente per creare un array inizializzato da assegnare all'array `mmGennaio`. In Java è possibile inizializzare un array in fase di creazione specificandone il contenuto dopo l'operatore `new`. Usando questa sintassi l'esempio precedente diventa:

```
int mmGennaio[] = new int[31];
mmGennaio[3] = 7;
mmGennaio = new int[] {1, 1, 2, 3, 5, 8, 13, 21, 34, 55};
System.out.println (mmGennaio[3]);
mmGennaio[10] = 1;                    // Errore in esecuzione !!
```

La differenza tra i due tipi di inizializzazione è che la prima consente di inizializzare un array in fase di dichiarazione, mentre la seconda consente di creare un array anonimo che è possibile assegnare a una qualsiasi variabile adeguata.

In Java le variabili array possono subire delle trasformazioni durante l'elaborazione e la dichiarazione è piuttosto povera di informazioni. In compenso è possibile conoscere dinamicamente in fase di esecuzione la dimensione di un array poiché a ogni variabile di questo tipo allocata è virtualmente associato un intero contenente il numero degli elementi contenuti. Per accedere a tale intero si pospone al nome della variabile la parola `length` separata da un punto. Le istruzioni seguenti

```
int mmGennaio[] = new int[31];
System.out.println (mmGennaio.length);
int serieDiFibonacci[] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55};
mmGennaio = serieDiFibonacci;
System.out.println (mmGennaio.length);
```

visualizzano

```
31
10
```

La notazione col punto ha un significato preciso e un utilizzo più generale di quello illustrato qui per gli array, ma di questo parleremo dettagliatamente in seguito.

6.2 Scansione di un array

Osserviamo un esempio.

```
char parola[] = { 'p', 'r', 'o', 'v', 'a' };
```

L'array `parola` può essere rappresentato come in Figura 6.1.

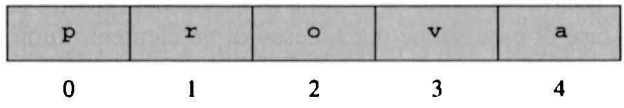


Figura 6.1 Struttura dell'array `parola`

Le istruzioni

```
parola[0] = 't';
a[4] = 'i';
```

assegnano al primo elemento del vettore `a` il valore `t` e al quinto `i`. Se `b` è una variabile carattere (cioè dello stesso tipo del vettore), è possibile assegnare il suo valore a un elemento di `parola` e viceversa:

```
b = 'n';
parola[3] = b;
```

Spesso l'array viene trattato all'interno di iterazioni; infatti risulta semplice far riferimento a suoi elementi incrementando ciclicamente il valore di una variabile intera e utilizzandola come indice. Consideriamo un'iterazione di esempio:

```
char parola[] = { 'p', 'r', 'o', 'v', 'a' };
int i;
for(i=0; i<=4; i++)
   System.out.print (parola[i]);
```

L'indice `i` dell'array `parola` è inizializzato a 0 e assume a ogni iterazione successiva i valori 1, 2, 3, 4. Il blocco del `for` visualizza in sequenza i valori p, r, o, v, a.

`prova`

Supponiamo di voler visualizzare l'array di `char` in ordine inverso. Possiamo allora usare il codice seguente

```
class RovesciaParole {
   public static void main (String argv[]) {
      char parola[] = { 'p', 'r', 'o', 'v', 'a' };
      int i1, i2;
```

```
        for (i1 = 0, i2 = parola.length - 1;
             i1 < i2;
             i1 = i1 + 1, i2 = i2 - 1) {
           char c = parola[i1];
           parola[i1] = parola[i2];
           parola[i2] = c;
        }
        System.out.println (parola);
    }
}
```

che produce il seguente risultato

```
avorp
```

✔ NOTA

La variabile `c` *in* `RovesciaParole` *viene dichiarata nel blocco e quindi non è visibile al di fuori di esso. La dichiarazione di una variabile all'interno del blocco rende più evidente che essa ha un utilizzo limitato ed evita che ci siano collisioni di nomi con altre variabili, anche se appesantisce l'elaborazione dal punto di vista delle prestazioni.*

Supponiamo di voler visualizzare i caratteri di un array `char` separati da virgole.

```
char a[] = { 'a', 'b', 'c' } ;
if (a.length > 0)
    for (int x = 0; ; ) {
        System.out.print (a[x]);
        if (++x < a.length)
            System.out.print (", ");
        else
            break;
    }
System.out.println ("");
```

Il programma produce la visualizzazione

```
a, b, c
```

Da notare che l'istruzione `if` prima del ciclo è necessaria per evitare errori nel caso in cui l'array a contenga 0 elementi. Abbiamo usato un ciclo infinito come nel Capitolo 5 per un caso simile e osserviamo di nuovo come lo stesso risultato sarebbe stato ottenuto con il codice seguente

```
char a[] = { 'a', 'b', 'c' };
for (int x = 0; x < a.length; ) {
    System.out.print (a[x]);
```

```
    if (++x < a.length)
       System.out.print (", ");
}
System.out.println ("");
```

in cui è valutata due volte per ogni ciclo la stessa condizione di uscita.

Per sommare tutti i numeri positivi contenuti in un array di `int` scriviamo il codice

```
int ai[] = { 10, -10, 5, -5, 7, -7 } ;
int somma = 0;
for (int x = 0; x < ai.length; x++) {
   if (ai[x] <= 0)
      continue;
      somma += ai[x];
   }
System.out.println (somma);
```

che produce il seguente risultato

```
22
```

Per esempio, supponiamo di voler visualizzare tutti i caratteri di un array di `char` fintanto che non si trova il primo segno d'interpunzione. Il codice seguente

```
      char a[] = {'a', 'b', 'c', '?' , 'd', 'e', 'f'};
interruzione:
      for (int x = 0; x < a.length; x++) {
         switch (a[x]) {
         case '.':
         case ',':
         case ';':
         case ':':
         case '!':
         case '?':
            break interruzione;
         default:
            break;
         }
         System.out.println (a[x]);
      }
      System.out.println ("fine");
```

produce il risultato

```
a
b
c
fine
```

6.3 Esempi con gli array

Osserviamo un esempio:

```
int voti[] = new int[6];
```

Le istruzioni

```
a[0] = 71;
a[1] = 4;
```

assegnano al primo elemento del vettore a il valore 71 e al secondo 4.

In generale un singolo elemento dell'array può essere utilizzato esattamente come una variabile semplice. Se g è una variabile intera, nell'espressione

```
g = g + voti[0] * voti[5];
```

il valore di g è addizionato al prodotto tra il primo e il sesto elemento di a e il risultato è assegnato a g.

Spesso l'array viene trattato all'interno di iterazioni; infatti risulta semplice far riferimento a suoi elementi incrementando ciclicamente il valore di una variabile intera e utilizzandola come indice. Consideriamo un'iterazione di esempio:

```
/* Inizializzazione dell'array */
for(i=0; i<=5; i++) {
   System.out.print ("Voto studente: ");
   voti[i] = sc.nextInt();
}
```

L'indice i dell'array a è inizializzato a 0 e assume a ogni iterazione successiva i valori 1, 2, 3, 4, 5. Il blocco del `for` richiede all'utente l'immissione di sei valori che vengono assegnati sequenzialmente agli elementi del vettore. Se quindi vengono inseriti in sequenza i valori 9, 18, 7, 15, 21 e 11, dopo l'esecuzione del ciclo il vettore si presenterà in memoria come in Figura 6.2.

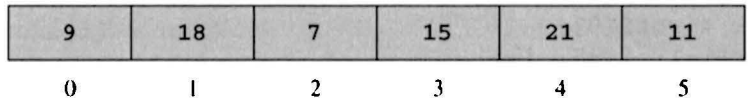


Figura 6.2 L'array `voti[6]` dopo la sua inizializzazione

Anche per ricercare all'interno dell'array valori che soddisfano certe condizioni si utilizzano abitualmente i cicli:

```
/* Ricerca del maggiore */
max = voti[0];
for (i = 1; i <= 5; i++)
```

```
if(voti[i] > max)
   max = voti[i];
```

L'esempio premette di determinare il maggiore degli elementi dell'array `voti`: la variabile `max` viene inizializzata al valore del primo elemento del vettore, quello con indice zero. Successivamente ogni ulteriore elemento viene confrontato con `max`: se risulta essere maggiore, il suo valore viene assegnato a `max`. Il ciclo deve comprendere dunque tutti gli elementi del vettore meno il primo, perciò l'indice `i` assume valori che vanno da 1 a 5.

Esaminiamo un programma che richiede all'utente l'immissione dei voti ottenuti da sei studenti, li memorizza nel vettore `voti` e ne determina il maggiore, il minore e la media.

```
/* Memorizza in un array di interi i voti ottenuti
 * da sei studenti e ne determina il maggiore,
 * il minore e la media                                    */
import java.util.Scanner;
class Array1 {
   public static void main (String argv[]) {

      int voti[] = new int[6];
      int i, max, min;
      float media;
      Scanner sc = new Scanner(System.in);

      System.out.print ("VOTI STUDENTI\n\n");
      /* Immissione voti */
      for(i=0; i<=5; i++) {
         System.out.print ("Voto studente n.");
         System.out.print (i+1);
         System.out.print (": ");
         voti[i] = sc.nextInt();
      }

      /* Ricerca del maggiore */
      max = voti[0];
      for(i=1; i<=5; i++)
         if(voti[i]>max)
            max = voti[i];

      /* Ricerca del minore */
      min = voti[0];
      for(i=1; i<=5; i++)
         if(voti[i]<min)
            min = voti[i];

      /* Calcolo della media */
      media = voti[0];
```

```
        for(i=1; i<=5; i++)
           media = media + voti[i];
        media = media/6;

        System.out.print ("Maggiore: ");
        System.out.println (max);
        System.out.print ("Minore: ");
        System.out.println (min);
        System.out.print ("Media: ");
        System.out.println (media);
    }
}
```

Si noti che la richiesta dei voti all'utente viene fatta evidenziando il numero d'ordine che corrisponde al valore dell'indice aumentato di una unità:

```
System.out.print ("Voto studente n.");
System.out.print (i+1);
System.out.print (": ");
```

Alla prima iterazione appare sullo schermo:

```
Voto studente n.1:
```

Considerazioni analoghe a quelle fatte per il calcolo del maggiore valgono per il minimo e la media.

Nella pratica, la memorizzazione in un vettore ha senso quando i valori debbono essere utilizzati più volte, come nel caso precedente. Nell'esempio, comunque, i calcoli potevano essere effettuati all'interno della stessa iterazione con un notevole risparmio di tempo di esecuzione:

```
/* Ricerca maggiore, minore e media */
max = voti[0];
min = voti[0];
media = 0;
for(i = 0; i <= 5; i++) {
    if(voti[i] > max)
        max = voti[i];
    if(voti[i] < min)
        min = voti[i];
    media = media+voti[i];
 }
```

Osserviamo un altro esempio che prende in considerazione più array monodimensionali. Per determinare la destrezza di *n* concorrenti sono state predisposte due prove, entrambe con una valutazione che varia da 1 a 10; il punteggio totale di ogni concorrente è dato dalla media aritmetica dei risultati delle due prove. Si richiede la visualizzazione di una tabella che contenga su ogni linea i risultati parziali e il punteggio totale di un concorrente.

```
/* Carica i punteggi di n concorrenti su due prove
 * Determina la classifica                          */
import java.util.Scanner;
class Classifica {
   public static void main (String argv[]) {

    int MAX_CONC = 1000; /* massimo numero di concorrenti */
    int MIN_PUN  = 1;  /* punteggio minimo per ogni prova */
    int MAX_PUN  = 10;/* punteggio massimo per ogni prova */
      float prova1[] = new float[MAX_CONC];
      float prova2[] = new float[MAX_CONC];
      float totale[] = new float[MAX_CONC];
      int i, n; float media;
      Scanner sc = new Scanner(System.in);

      do {
         System.out.print ("Numero di concorrenti: ");
         n = sc.nextInt();
      } while(n<1 || n>MAX_CONC);

      /* Per ogni concorrente, richiesta punteggio
       * nelle due prove                              */
      for(i=0; i<n; i++) {
         System.out.print ("\nConcorrente n.");
         System.out.print (i+1);
         System.out.print (": ");
         do {
            System.out.print ("\nPrima Prova: ");
            prova1[i] = sc.nextInt();
         } while(prova1[i]<MIN_PUN || prova1[i]>MAX_PUN);

         do {
            System.out.print ("Seconda Prova: ");
            prova2[i] = sc.nextInt();
         } while(prova2[i]<MIN_PUN || prova2[i]>MAX_PUN);
      }

      /* Calcolo media per concorrente */
      for(i=0; i<n; i++)
         totale[i] = (prova1[i]+prova2[i])/2;

      System.out.print ("\n      CLASSIFICA\n");
      for(i=0; i<n; i++) {
         System.out.print (prova1[i]);
         System.out.print (" ");
         System.out.print (prova2[i]);
         System.out.print (" ");
```

```
            System.out.print (totale[i]);
            System.out.println (" ");
        }
    }
}
```

Non conoscendo a priori il numero di concorrenti che parteciperanno alle gare facciamo l'ipotesi che comunque non siano più di 1000, valore che memorizziamo nella costante MAX_CONC. In conseguenza di ciò definiamo di lunghezza MAX_CONC gli array che conterranno i risultati: prova1, prova2 e totale. Richiediamo all'utente a tempo di esecuzione il numero effettivo dei concorrenti e verifichiamo che non sia minore di 1 e maggiore di MAX_CONC:

```
do {
    System.out.print ("Numero di concorrenti: ");
    n = sc.nextInt();
} while(n<1 || n>MAX_CONC);
```

In seguito richiediamo l'introduzione dei risultati della prima e della seconda prova di ogni concorrente, controllando che tale valutazione non sia minore di 1 e maggiore di 10, nel qual caso ripetiamo la richiesta. Abbiamo memorizzato in MIN_PUN e MAX_PUN i limiti inferiore e superiore del punteggio assegnabile, in maniera che, se questi venissero modificati, basterebbe intervenire sulle loro definizioni perché il programma continui a funzionare correttamente. Infine calcoliamo il punteggio totale e lo visualizziamo. Un esempio di esecuzione è mostrato in Figura 6.3.

```
Numero concorrenti: 3

Concorrente n.1
Prima prova: 8
Seconda prova: 7

Concorrente n.2
Prima prova: 5
Seconda prova: 9

Concorrente n.3
Prima prova: 8
Seconda prova: 8

      CLASSIFICA
8.000000  7.000000  7.500000
5.000000  9.000000  7.000000
8.000000  8.000000  8.000000
```

Figura 6.3 Esempio di esecuzione del programma

6.4 Array multidimensionali

Il concetto di array appena illustrato può essere esteso per rappresentare variabili distinguibili da più di un indice. In tal caso si parla di array di dimensione *n* o *matrici* di dimensione *n*. Gli array presentati nel paragrafo precedente sono quindi array di dimensione 1. Riprendendo l'esempio dei millimetri di pioggia giornalieri e avendo la necessità di memorizzare lo stato delle piogge di un anno invece che quello di un mese, può essere comodo avere una variabile array `mmAnno` di dimensione 2 in cui il primo indice indica il mese, mentre il secondo indica il giorno del mese. Per registrare che il 7 di ottobre sono caduti 5 millimetri di pioggia si potrebbe allora usare una notazione del tipo

```
mmAnno[9][6] = 5;
```

La dichiarazione di un'array multidimensionale è del tutto analoga a quella di uno monodimensionale, con la differenza che bisogna riportare tante parentesi aperte e chiuse quante sono le dimensioni. La variabile `mmAnno` si dichiara nel modo seguente

```
int mmAnno[][];
```

Per allocare la memoria necessaria, si usa ancora l'operatore `new` dichiarando il numero di elementi per ciascuna dimensione. Nel nostro esempio, l'allocazione diventa

```
mmAnno = new int[12][31];
```

In realtà, gli array multidimensionali sono implementati in Java come array di array. Questo significa che la variabile `mmAnno` è un array monodimensionale formato da 12 elementi che sono a loro volta array monodimensionali di 31 interi, come evidenziano le seguenti istruzioni

```
System.out.println (mmAnno.length);
System.out.println (mmAnno[0].length);
System.out.println (mmAnno[1].length);
System.out.println (mmAnno[2].length);
```

che visualizzano

```
12
31
31
31
```

Questo approccio permette di poter creare delle strutture dati che vanno oltre la definizione di array multidimensionale. Riprendendo l'esempio precedente, abbiamo allocato 31 giorni per ciascun mese, anche se sappiamo che in realtà alcuni giorni non saranno mai utilizzati. L'esempio seguente permette di evitare tale inconveniente, anche se nello specifico si tratta di uno spreco minimo, utilizzando una matrice incompleta.

```
int mmAnno[][] = new int[12][];
mmAnno[0]   = new int[31];
mmAnno[1]   = new int[29];
```

```
mmAnno[2]  = new int[31];
mmAnno[3]  = new int[30];
mmAnno[4]  = new int[31];
mmAnno[5]  = new int[30];
mmAnno[6]  = new int[31];
mmAnno[7]  = new int[31];
mmAnno[8]  = new int[30];
mmAnno[9]  = new int[31];
mmAnno[10] = new int[30];
mmAnno[11] = new int[31];
```

Quando si alloca un array di array in cui mancano una o più dimensioni, come sulla prima riga dell'esempio, l'operatore `new` assegna `null` a ogni elemento.

Anche gli array di array possono essere inizializzati come gli array semplici. L'esempio seguente permette di memorizzare in ogni elemento di una tabella di tre righe e quattro colonne il prodotto dei suoi indici.

```
class TavolaPitagorica {
   public static void main (String argv[]) {
      int tavolaPitagorica[][] = {
         { 0*0, 0*1, 0*2, 0*3 },
         { 1*0, 1*1, 1*2, 1*3 },
         { 2*0, 2*1, 2*2, 2*3 }
      };
      System.out.println (tavolaPitagorica.length);
      System.out.println (tavolaPitagorica[0].length);
      System.out.println (tavolaPitagorica[2][3]);
   }
}
```

Eseguendo questo codice si ottiene la visualizzazione

```
3
4
6
```

Osserviamo un altro esempio:

```
float mat[][] = new float[4][3];
```

L'array `mat` che abbiamo dichiarato contiene 4 righe e 3 colonne, per un totale di dodici elementi; per accedere a ciascuno di essi si utilizzano due indici: il primo specifica la riga, il secondo la colonna. Gli indici variano rispettivamente tra 0 e $r-1$ e tra 0 e $c-1$, dove r e c sono il numero di righe e il numero di colonne. Abbiamo cioè che la matrice è composta dai seguenti elementi:

```
mat[0][0]  mat[0][1]  mat[0][2]
mat[1][0]  mat[1][1]  mat[1][2]
mat[2][0]  mat[2][1]  mat[2][2]
mat[3][0]  mat[3][1]  mat[3][2]
```

Per esempio, `mat[1][2]` fa riferimento all'elemento presente nella seconda riga della terza colonna.

Per memorizzare i ricavi ottenuti dalla vendita di 10 prodotti in 5 punti vendita nei dodici mesi dell'anno, potremmo utilizzare la matrice tridimensionale `marketing` così dichiarata:

```
int marketing[][] = new int[10][5][12];
```

Realizziamo ora un programma che richiede all'utente i valori da inserire, li memorizza nella matrice bidimensionale `mat` e la visualizza.

Per effettuare il caricamento dei dati nella matrice utilizziamo due cicli, uno più esterno che mediante la variabile `i` fa la scansione delle righe da zero a 3 (= 4-1), e un altro che percorre, per mezzo della variabile `j`, le colonne da zero a 2 (= 3-1):

```
for(i=0; i<4; i++)
   for(j=0; j<3; j++) {
     System.out.print ("Inserisci un elemento: ");
     mat[i][j] = sc.nextFloat();
   }
```

In questo modo viene riempita tutta la prima riga, poi la seconda e così via; se l'utente passa i valori 2, 55, 12, 98, 34 ... essi verranno inseriti negli elementi `mat[0][0]`, `mat[0][1]`, `mat[0][2]`, `mat[1][0]`, `mat[1][1]` ...

Si può ottenere il caricamento per colonne invertendo semplicemente i due cicli:

```
for(j=0; j<3; j++)
  for(i=0; i<4; i++) {
     System.out.print ("Inserisci un elemento: ");
     mat[i][j] = sc.nextFloat();
  };
```

Il numero totale d'iterazioni è sempre uguale a 12 e gli indici `j` e `i` fanno la scansione delle colonne e delle righe della matrice senza fuoriuscire dai margini. Otteniamo la visualizzazione ancora una volta con due cicli `for` annidati uno nell'altro.

```
/* Caricamento di una matrice */
import java.util.Scanner;
class Matrice1 {
  public static void main (String argv[]) {

    int N = 4;          /* righe */
    int M = 3;          /* colonne */
    int i, j;
    float mat[][] = new float[N][M];
    Scanner sc = new Scanner(System.in);

    System.out.print ("CARICAMENTO DELLA MATRICE\n");
    for(i=0; i<N; i++)
      for(j=0; j<M; j++) {
        System.out.print ("Inserisci l'elemento (");
```

```
        System.out.print (i);
        System.out.print (", ");
        System.out.print (j);
        System.out.print ("): ");
        mat[i][j] = sc.nextFloat();
      }
    System.out.print ("\n\n VISUALIZZAZIONE DELLA MATRICE\n");
    for(i=0; i<N; i++) {
      for(j=0; j<M; j++) {
        System.out.print (mat[i][j]);
        System.out.print ("  ");
      }
      System.out.println ();
    }
  }
}
```

Nel programma precedente le dimensioni della matrice erano fissate a priori: modifichiamolo in modo da far decidere all'utente il numero delle righe e delle colonne.

```
/* Caricamento di una matrice
 * le cui dimensioni vengono decise dall'utente */
import java.util.Scanner;

class Matrice2 {
  public static void main (String argv[]) {

    int MAXRIGHE   = 100;
    int MAXCOLONNE = 100;
    float mat[][] = new float[MAXRIGHE][MAXCOLONNE];
    int n, m, i, j;
    Scanner sc = new Scanner(System.in);

    /* Richiesta delle dimensioni */
    do {
        System.out.print ("Numero di righe: ");
        n = sc.nextInt();
    } while((n>=MAXRIGHE) || (n<1));

    do {
        System.out.print ("Numero di colonne: ");
        m = sc.nextInt();
    } while((n>=MAXRIGHE) || (n<1));

    System.out.print ("CARICAMENTO DELLA MATRICE\n");
    for(i=0; i<n; i++)
      for(j=0; j<m; j++) {
        System.out.print ("Inserisci l'elemento (");
```

```
        System.out.print (i);
        System.out.print (", ");
        System.out.print (j);
        System.out.print ("): ");
        mat[i][j] = sc.nextFloat();
      }
    System.out.print ("\n\n VISUALIZZAZIONE DELLA MATRICE\n");
    for(i=0; i<n; i++) {
      for(j=0; j<m; j++) {
        System.out.print (mat[i][j]);
        System.out.print ("  ");
      }
      System.out.println ();
    }
  }
}
```

La matrice viene definita con un massimo numero di linee e di colonne:

```
float mat[][] = new float[MAXRIGHE][MAXCOLONNE];
```

dove MAXLINEE e MAXCOLONNE sono due costanti che abbiamo dichiarato precedentemente, il cui valore deve essere scelto in relazione alle massime dimensioni. Successivamente si richiede l'inserimento del valore di *n*, numero di linee che realmente verranno riempite:

```
do {
   System.out.print ("Numero di righe: ");
   n = sc.nextInt();
} while((n>=MAXRIGHE) || (n<1));
```

Come già evidenziato un array può essere *inizializzato* automaticamente in fase di dichiarazione, evitando l'uso dell'operatore new. Una matrice a due dimensioni è inizializzata in questo modo:

```
double mat[][] = {
                    {5, 8.3, 6},
                    {4, 9, 3.2},
                    {1, 0.1, 2},
                    {7, 5, 8.5}
                };
```

6.5 Prodotto di matrici

Passiamo adesso a un problema più complesso: date due matrici `mat1[N][P]`, `mat2[P][M]`, calcolare la matrice prodotto in cui ogni elemento è dato da:

$$\texttt{pmat[i][j]} = \sum_{k=0}^{P-1} \texttt{mat1[i][k]} * \texttt{mat2[k][j]}$$

per `i=0..N-1, j=0..M-1`

Il prodotto così definito si può ottenere soltanto se il numero di colonne della prima matrice (*P*) è uguale al numero di righe della seconda. La matrice `pmat` è dunque costituita da *N* righe e *M* colonne.

Consideriamo le matrici di Figura 7.5: l'elemento [2][4] della matrice prodotto è dato da

```
pmat[2][4] = mat1[2][0]*mat2[0][4] +
             mat1[2][1]*mat2[1][4] +
             mat1[2][2]*mat2[2][4]
```

ossia

```
pmat[2][4] = 5*3 + 2*4 + 0*5 = 23
```

Venendo dunque al programma richiesto, in cui in primo luogo si devono caricare i dati delle due matrici.

```
/* Calcolo del prodotto di due matrici */
import java.util.Scanner;
class ProdottoMatrici {
  public static void main (String argv[]) {

    int N = 4;        /* prima matrice */
    int P = 3;        /* seconda matrice */
    int M = 5;        /* matrice prodotto */
    float mat1[][] = new float[N][P];
    float mat2[][] = new float[P][M];
    float pmat[][] = new float[N][M];
    Scanner sc = new Scanner(System.in);

    int i, j, k;

    System.out.print ("CARICAMENTO DELLA PRIMA MATRICE\n");
    for(i=0; i<N; i++)
      for(j=0; j<P; j++) {
        System.out.print ("Inserisci l'elemento (");
        System.out.print (i);
        System.out.print (", ");
        System.out.print (j);
        System.out.print ("): ");
        mat1[i][j] = sc.nextFloat();
      }
    System.out.print ("CARICAMENTO DELLA SECONDA MATRICE\n");
    for(i=0; i<P; i++)
```

```
      for(j=0; j<M; j++) {
        System.out.print ("Inserisci l'elemento (");
        System.out.print (i);
        System.out.print (", ");
        System.out.print (j);
        System.out.print ("): ");
        mat2[i][j] = sc.nextFloat();
      }
    /* Calcolo del prodotto */
    for(i=0; i<N; i++)
      for(j=0; j<M; j++)
        for(k=0; k<P; k++)
          pmat[i][j] = pmat[i][j] + mat1[i][k] * mat2[k][j];
    System.out.print ("\n\n PRIMA MATRICE\n");
    for(i=0; i<N; i++) {
      for(j=0; j<P; j++) {
        System.out.print (mat1[i][j]);
        System.out.print ("  ");
      }
      System.out.println ();
    }
    System.out.print ("\n\n SECONDA MATRICE\n");
    for(i=0; i<P; i++) {
      for(j=0; j<M; j++) {
        System.out.print (mat2[i][j]);
        System.out.print ("  ");
      }
      System.out.println ();
    }
    System.out.print ("\n\n MATRICE PRODOTTO\n");
    for(i=0; i<N; i++) {
      for(j=0; j<M; j++) {
        System.out.print (pmat[i][j]);
        System.out.print ("  ");
      }
      System.out.println ();
    }
  }
}
```

Si ottiene il valore dell'elemento `i, j` della matrice prodotto con un ciclo che fa la scansione della riga `i` di `mat1` e della colonna `j` di `mat2` e accumula in `pmat[i][j]` la sommatoria dei prodotti dei corrispondenti elementi di `mat1` e `mat2`:

```
for(k=0; k<P; k++)
  pmat[i][j] = pmat[i][j] + mat1[i][k] * mat2[k][j];
```

La variabile k permette di scorrere contemporaneamente la linea i di mat1 e la colonna j di mat2; il suo valore varia da 0 a *P*. Il procedimento appena visto va ripetuto per ognuno degli elementi della matrice prodotto:

```
/* Calcolo del prodotto */
for(i=0; i<N; i++)
  for(j=0; j<M; j++)
    for(k=0; k<P; k++)
      pmat[i][j] = pmat[i][j] + mat1[i][k] * mat2[k][j];
```

I due cicli for fissano a ogni iterazione una certa riga di mat1 e di pmat e una certa colonna di mat2 e di pmat. Riportiamo in Figura 6.4 un esempio di esecuzione del programma.

```
CARICAMENTO DELLA PRIMA MATRICE
Inserisci l'elemento (0, 0): 1
Inserisci l'elemento (0, 1): 0
Inserisci l'elemento (0, 2): 0
Inserisci l'elemento (1, 0): 22
Inserisci l'elemento (1, 1): -6
Inserisci l'elemento (1, 2): 3
Inserisci l'elemento (2, 0): 5
Inserisci l'elemento (2, 1): 2
Inserisci l'elemento (2, 2): 0
Inserisci l'elemento (3, 0): 11
Inserisci l'elemento (3, 1): 4
Inserisci l'elemento (3, 2): 7
CARICAMENTO DELLA SECONDA MATRICE
Inserisci l'elemento (0, 0): 2
Inserisci l'elemento (0, 1): 0
Inserisci l'elemento (0, 2): 4
Inserisci l'elemento (0, 3): 0
Inserisci l'elemento (0, 4): 3
Inserisci l'elemento (1, 0): 0
Inserisci l'elemento (1, 1): 1
Inserisci l'elemento (1, 2): 5
Inserisci l'elemento (1, 3): 1
Inserisci l'elemento (1, 4): 4
Inserisci l'elemento (2, 0): 21
Inserisci l'elemento (2, 1): 1
Inserisci l'elemento (2, 2): 2
Inserisci l'elemento (2, 3): 2
Inserisci l'elemento (2, 4): 5
```

Figura 6.4 Esempio di esecuzione del programma del calcolo del prodotto di due matrici (*segue*)

```
PRIMA MATRICE
1.0      0.0      0.0
22.0    -6.0      3.0
5.0      2.0      0.0
11.0     4.0      7.0

SECONDA MATRICE
2.0      0.0      4.0      0.0      3.0
0.0      1.0      5.0      1.0      4.0
21.0     1.0      2.0      2.0      5.0

MATRICE PRODOTTO
2.0      0.0      4.0      0.0      3.0
107.0   -3.0     64.0      0.0     57.0
10.0     2.0     30.0      2.0     23.0
169.0   11.0     78.0     18.0     84.0
```

Figura 6.4 *(seguito)*

6.6 Ricerche e ordinamenti

È esperienza comune che nella gestione dei più svariati insiemi di dati (vettori o matrici, ma più in generale anche archivi cartacei, listini prezzi, voci di un'enciclopedia o addirittura semplici carte da gioco) sia spesso necessario: stabilire se un elemento è o no presente nell'insieme, ordinare l'insieme in un determinato modo (in genere in maniera crescente o decrescente), unire (fondere) due o più insiemi in un unico insieme evitando possibili duplicazioni. Queste tre attività, che in informatica vengono indicate rispettivamente con i termini di *ricerca*, *ordinamento* e *fusione* oppure con i loro equivalenti inglesi *search*, *sort* e *merge*, sono estremamente frequenti e svolgono un ruolo della massima importanza in tutti i possibili impieghi degli elaboratori. È stato per esempio stimato che l'esecuzione dei soli programmi di ordinamento rappresenti oltre il 30% del lavoro svolto dai computer. È quindi ovvio come sia della massima importanza disporre di programmi che svolgano questi compiti nel minor tempo possibile.

6.7 Ricerca completa

Un primo algoritmo per determinare se un valore è presente all'interno di un array, applicabile anche a sequenze non ordinate, è quello comunemente detto di *ricerca completa*, che opera una scansione sequenziale degli elementi del vettore confrontandoli con il valore ricercato. Nel momento in cui tale verifica dà esito positivo la scansione ha termine e viene restituita la posizione dell'elemento all'interno dell'array stesso, espressa tramite l'indice.

Per determinare che il valore non è presente, il procedimento deve controllare uno a uno tutti gli elementi fino all'ultimo, prima di poter sentenziare il fallimento della ricerca. L'array che conterrà la sequenza è `vet` formato da `MAX_ELE` elementi.

```
/* Ricerca completa.
 * Ricerca sequenziale di un valore nel vettore */
import java.util.Scanner;
class Ricerca {
   public static void main (String argv[]) {

      char vet[] = new char[1000];
      char c;
      int i, n;
      Scanner sc = new Scanner(System.in);

      /* Immissione lunghezza della sequenza */
      do {
          System.out.print ("Numero elementi: ");
          n = sc.nextInt();
          sc.nextLine();
      } while (n < 1 || n > vet.length);

      /* Immissione elementi della sequenza */
      for (i = 0; i < n; i++) {
         System.out.print ("Immettere carattere n.");
         System.out.print (i);
         System.out.print (": ");
         vet[i] = sc.nextLine().charAt(0);
      }

       System.out.print ("Elemento da ricercare: ");
       c = sc.nextLine().charAt(0);

       /* Ricerca sequenziale */
       for (i = 0; i < n && c != vet[i]; i++)
          ;
       if (i != n) {
         System.out.print ("Elemento ");
         System.out.print (c);
         System.out.print (" presente in posizione: ");
         System.out.println (i);
       }
       else
         System.out.println ("Elemento non presente");
   }
}
```

Il programma presenta le solite fasi di richiesta e relativa immissione del numero degli elementi della sequenza e dei valori che la compongono. Successivamente l'utente inserisce il carattere da ricercare, che viene memorizzato nella variabile `c`. La ricerca parte dal primo elemento dell'array (quello con indice zero) e prosegue fintantoché `i` è minore di `n` e contemporaneamente il confronto fra `c` e `vet[i]` dà esito negativo:

```
for (i = 0; i < n && c != vet[i]; i++)
    ;
```

Si osservi che, come abbiamo spiegato trattando le espressioni, la valutazione di `c!=vet[i]` non avviene nel caso che `i` non sia più minore di `n`. L'`if` successivo stabilirà se l'iterazione è terminata con `i` diverso da `n` (minore):

```
if(i != n)
```

Nel qual caso `c` è risultato essere uguale a `vet[i]`.

Esistono molte altre soluzioni al ciclo precedente. Per esempio si potrebbe adottare un `while`

```
while(c!=vet[i] && i<n-1)  ++i;
```

o anche:

```
while(c!=vet[i] && i++<n-1)
    ;
```

6.8 Ordinamenti

Un altro fondamentale problema dell'informatica spesso collegato con la ricerca è quello che consiste nell'ordinare un vettore disponendo i suoi elementi in ordine crescente o decrescente. Esistono numerosi algoritmi che consentono di ordinare un array; uno dei più famosi, ma anche meno efficienti, è quello comunemente detto *bubblesort (ordinamento a bolle)*. Il nome deriva dal fatto che gli elementi vengono portati nelle posizioni che competono loro in modo simile a delle bolle di un liquido che vengono a galla. Osserviamo una sua versione in Java che ordina un array in modo crescente. L'array è identificato da `vet` e ha *n* elementi:

```
for(j=0;j<n-1; j++)
  for(i=0; i<n-1; i++)
    if (vet[i]>vet[i+1]) {
       aux=vet[i];
       vet[i]=vet[i+1];
       vet[i+1]=aux;
    }
```

Nel ciclo più interno gli elementi adiacenti vengono confrontati: se `vet[i]` risulta maggiore di `vet[i+1]` si effettua lo scambio tra i loro valori. Per quest'ultima operazione si ha la necessità di una variabile di appoggio, che nell'esempio è `aux`. Il ciclo si ripete finché tutti gli elementi sono stati confrontati, quindi fino a quando `i` è minore di `n-1`, perché il confronto viene fatto tra `vet[i]` e `vet[i+1]`.

Questa serie di confronti non è in generale sufficiente a ordinare l'array. La sicurezza dell'ordinamento è data dalla sua ripetizione per $n-1$ volte; nell'esempio ciò si ottiene con un `for` più esterno controllato dalla variabile `j` che varia da 0 a $n-1$ (Figura 6.5).

In realtà il numero di volte per cui il ciclo interno va ripetuto dipende da quanto è disordinata la sequenza di valori iniziali. Per esempio, l'ordinamento di un array di partenza con valori 10, 12, 100, 50, 200, 315 ha bisogno di un solo scambio, che viene effettuato per `i=2` e `j=0`; dunque tutti i cicli successivi sono inutili. A questo proposito si provi a ricostruire i passaggi della Figura 6.5 con questi valori di partenza.

Si può dedurre che l'array è ordinato e l'esecuzione delle iterazioni cessa quando un intero ciclo interno non ha dato luogo ad alcuno scambio di valori tra `vet[i]` e `vet[i+1]`:

```
do {
  k = 0;
  for(i=0; i<n-1; i++)
    if(vet[i]>vet[i+1])
      {aux=vet[i]; vet[i]=vet[i+1]; vet[i+1]=aux; k=1;}
} while(k==1 && n>1);
```

Una prima ottimizzazione dell'algoritmo si ottiene interrompendo il ciclo esterno la prima volta che per un'intera iterazione del ciclo interno la clausola `if` non ha dato esito positivo.

Nel ciclo esterno la variabile `k` viene inizializzata a zero: se almeno un confronto del ciclo piccolo dà esito positivo, a `k` viene assegnato il valore 1. In pratica la variabile `k` è utilizzata come *flag* (bandiera): se il suo valore è 1 il ciclo deve essere ripetuto, altrimenti no.

```
vet[0] = 9
vet[1] = 18
vet[2] = 7
vet[3] = 15
vet[4] = 21
vet[5] = 11
```

	9 9 9 9 9 18 7 7 7 7 7 18 15 15 15 15 15 18 18 18 21 21 21 21 11 11 11 11 11 21	7 7 7 7 7 9 9 9 9 9 15 15 15 15 15 18 18 18 11 11 11 11 11 18 18 21 21 21 21 21	7 7 7 7 7 9 9 9 9 9 15 15 11 11 11 11 11 15 15 15 18 18 18 18 18 21 21 21 21 21	7 7 7 7 7 9 9 9 9 9 11 11 11 11 11 15 15 15 15 15 18 18 18 18 18 21 21 21 21 21	7 7 7 7 7 9 9 9 9 9 11 11 11 11 11 15 15 15 15 15 18 18 18 18 18 21 21 21 21 21
i	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4
j	0	1	2	3	4

Figura 6.5 Esempio di ordinamento con l'algoritmo di bubblesort

Nel caso dell'array di partenza di Figura 6.5, l'adozione dell'ultimo algoritmo fa risparmiare un ciclo esterno (cinque cicli interni) rispetto al precedente. La prima volta che l'esecuzione del ciclo esterno non dà esito a scambi corrisponde al valore di `j` uguale a 3, `k` rimane a valore zero e le iterazioni hanno termine. Si provi con valori iniziali meno disordinati per verificare l'ulteriore guadagno in tempo d'esecuzione.

Osservando ancora una volta la Figura 6.5 si nota che a ogni incremento di `j`, variabile che controlla il ciclo esterno, almeno gli ultimi `j+1` elementi sono ordinati. Il fatto è valido in generale, poiché il ciclo interno sposta di volta in volta l'elemento più pesante verso il basso. Dall'ultima osservazione possiamo ricavare un'ulteriore ottimizzazione dell'algoritmo:

```
do {
  k = 0;
  for(i=0; i<n-1; i++)
    if(vet[i]>vet[i+1])
      {aux=vet[i]; vet[i]=vet[i+1]; vet[i+1]=aux; k=1;}
  --n;
} while(k==1 && n>1);
```

In tale ottimizzazione, a ogni nuova ripetizione del ciclo esterno viene decrementato il valore limite del ciclo interno, in modo da diminuire di uno, di volta in volta, il numero di confronti effettuati. Ma è ancora possibile un'altra ottimizzazione:

```
p = n;
do {
  k = 0;
  for(i=0; i<n-1; i++)
    if(vet[i]>vet[i+1]) {
      aux=vet[i]; vet[i]=vet[i+1]; vet[i+1]=aux;
       k = 1; p = i+1;
    }
n = p;
} while(k==1 && n>1);
```

Il numero dei confronti effettuati dal ciclo interno si interrompe lì dove la volta precedente si è avuto l'ultimo scambio, come si osserva dal confronto tra le Figure 6.5 e 6.6.

6.9 Ricerca binaria

Quando l'array risulta ordinato la ricerca di un valore al suo interno può avvenire mediante criteri particolari, uno dei quali è la ricerca detta *binaria* o *dicotomica*.

```
/* Ricerca binaria
 * Programma completo di immissione, ordinamento e ricerca */
import java.util.Scanner;
class RicercaBinaria {
  public static void main (String argv[]) {
```

```
vet[0] = 9
vet[1] = 18
vet[2] = 7
vet[3] = 15
vet[4] = 21
vet[5] = 11
```

	9	9	9	9	9	7	7	7	7	7	7	7	7	7		
	18	7	7	7	7	9	9	9	9	9	9	9	9	9		
	7	18	15	15	15	15	15	15	15	15	15	11	11	11		
	15	15	18	18	18	18	18	18	11	11	11	15	15	15		
	21	21	21	21	11	11	11	11	18	18	18	18	18	18		
	11	11	11	11	21	21	21	21	21	21	21	21	21	21		
i	0	1	2	3	4	0	1	2	3	0	1	2	0	1		
j	0					1				2			3			

Figura 6.6 Esempio di ordinamento con l'algoritmo di bubblesort ottimizzato

```
int n = 6;              /* numero di elementi dell'array */
char vet[] = new char[n]; /* array contenente
                             i valori immessi */
char ele=' ';             /* elemento da ricercare */
char aux;      /* variabile di appoggio per lo scambio */
int basso, alto, pos; /* usati per la ricerca binaria */
int i, k, p;
Scanner sc = new Scanner(System.in);

/* Immissione elementi della sequenza */
for(i=0; i<n; i++) {
  System.out.print ("Immettere carattere n.");
  System.out.print (i);
  System.out.print (": ");
  vet[i] = sc.nextLine().charAt(0);
}

/* Ordinamento ottimizzato */
p = n;
do {
    k = 0;
      for(i=0; i<n-1; i++) {
```

```
            if(vet[i]>vet[i+1]) {
              aux = vet[i]; vet[i] = vet[i+1]; vet[i+1] = aux;
              k = 1; p = i+1;
            }
          }
          n = p;
    } while(k==1 && n>1);

    System.out.print ("\nElemento da ricercare: ");
    ele = sc.nextLine().charAt(0);

    /* Ricerca binaria */
    n = 6;
    alto = 0; basso = n-1; pos = -1;
    do {
        i = (alto+basso)/2;
        if(vet[i]==ele)  pos = i;
        else
          if(vet[i]<ele)
            alto = i+1;
          else
            basso = i-1;
    } while(alto<=basso && pos==-1);

    if(pos != -1) {
      System.out.print ("\nElemento ");
      System.out.print (ele);
      System.out.print (" presente in posizione: ");
      System.out.print (pos);
    }
    else
      System.out.print ("Elemento non presente");
  }
}
```

Dopo l'immissione dei valori del vettore, il loro ordinamento con bubblesort e l'accettazione dell'elemento da cercare, abbiamo i comandi della ricerca binaria vera e propria:

```
/* ricerca binaria */
alto = 0; basso = n-1; pos = -1;
do {
  i = (alto+basso)/2;
  if(vet[i]==ele) pos=i;
  else
    if(vet[i]<ele)
      alto = i+1;
```

```
    else
      basso = i-1;
} while(alto<=basso && pos==-1);
```

Si confronta il valore da ricercare, che è memorizzato nella variabile `ele`, con l'elemento intermedio dell'array. L'indice `i` di tale elemento lo si calcola addizionando l'indice inferiore dell'array (`0`), memorizzato nella variabile `alto`, a quello superiore (`n-1`), memorizzato nella variabile `basso`, e dividendo la somma per due. Essendo l'array ordinato si possono presentare tre casi:

1. `vet[i]` è uguale a `ele`, la ricerca è finita positivamente, si memorizza l'indice dell'elemento in `pos` e il ciclo di ricerca ha termine;
2. `vet[i]` è minore di `ele`, la ricerca continua tra i valori maggiori di `vet[i]` che sono memorizzati negli elementi con indice compreso tra `i+1` e `basso`, per cui si assegna ad `alto` il valore `i+1`. Se non si sono già esaminati tutti gli elementi del vettore (`alto` è minore o uguale a `basso`) la ricerca continua assegnando ancora una volta a `i` il valore `(basso+alto)/2`;
3. `vet[i]` è maggiore di `ele`, la ricerca continua tra i valori minori di `vet[i]` che sono memorizzati negli elementi con indice compreso tra `alto` e `i-1`, per cui si assegna a `basso` il valore `i-1`. Se non si sono già esaminati tutti gli elementi del vettore (`alto` è minore di `basso`) la ricerca continua assegnando ancora una volta a `i` il valore `(basso+alto)/2`.

Nella Figura 6.7 si osserva il mutare dei valori di `alto`, `basso` e `i` fino al reperimento del valore desiderato ("o"). Il numero di cicli e corrispondenti confronti effettuati è risultato uguale a tre, mentre se avessimo utilizzato la ricerca sequenziale avremmo avuto nove iterazioni. La ricerca sequenziale esegue nel caso più fortunato – quello in cui l'elemento cercato è proprio il primo – un unico confronto; nel caso più sfortunato – quello in cui l'elemento cercato è invece l'ultimo – esegue n confronti. Si ha quindi che la ricerca sequenziale effettua in media $(n + 1)/2$ confronti.

La ricerca binaria offre delle prestazioni indubbiamente migliori: al massimo esegue un numero di confronti pari al logaritmo in base 2 di n. Questo implica che nel caso in cui n sia uguale a 1000 per la ricerca sequenziale si hanno in media 500 confronti, per quella binaria al massimo 10. Poiché, come per l'ordinamento, il tempo impiegato per eseguire il programma è direttamente proporzionale al numero dei confronti effettuati, è chiaro come la ricerca binaria abbia tempi di risposta mediamente molto migliori della ricerca sequenziale.

Osserviamo, tuttavia, che mentre si può effettuare la ricerca sequenziale su qualsiasi vettore, per la ricerca binaria è necessario disporre di un vettore ordinato, così che non sempre risulta possibile applicare tale algoritmo.

6.10 Fusione

Un altro algoritmo interessante è quello che partendo da due array monodimensionali ordinati ne ricava un terzo, anch'esso ordinato. I due array possono essere di lunghezza qualsiasi e in generale non uguale. Il programma richiede all'utente l'immissione

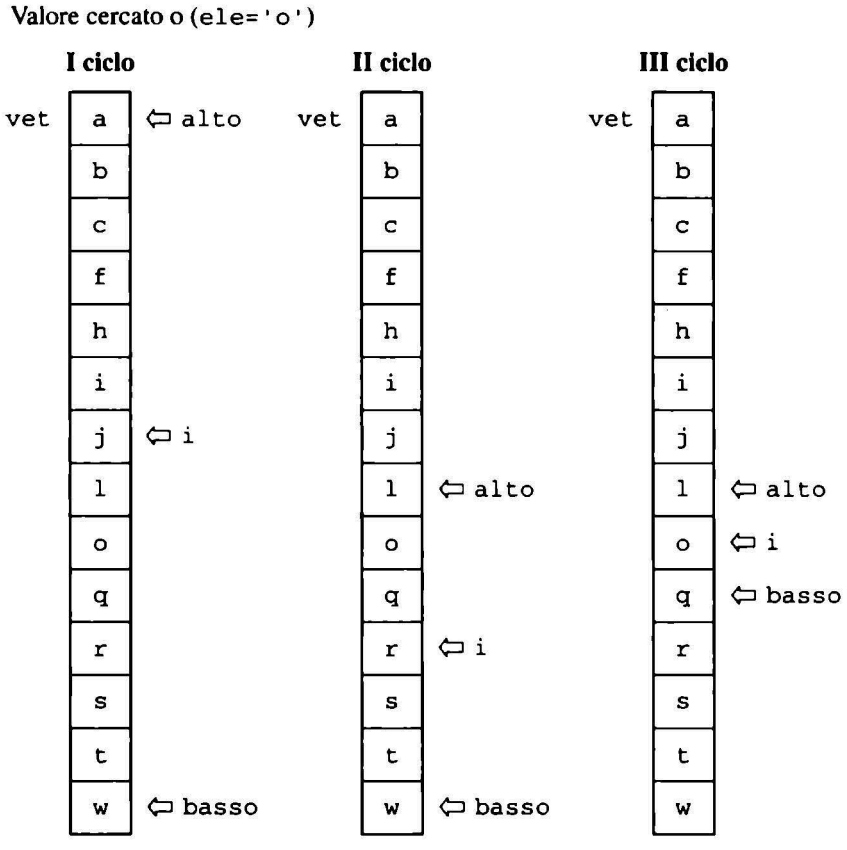


Figura 6.7 Esempio di ricerca binaria di `ele='o'`

della lunghezza di ognuna delle due sequenze e gli elementi che le compongono. Successivamente ordina le sequenze ed effettua la fusione (*merge*) di una nell'altra, memorizzando il risultato in un array a parte.

```
/* Fusione di due sequenze ordinate
 * Programma completo di immissione e merge */
import java.util.Scanner;
class Fusione {
  public static void main (String argv[]) {
    int MAX_ELE = 1000;
    char vet1[] = new char[MAX_ELE];   /* prima sequenza */
    char vet2[] = new char[MAX_ELE];   /* seconda sequenza */
    char vet3[] = new char[MAX_ELE*2];   /* merge */
    int n;                    /* lunghezza prima sequenza */
```

```
int m;          /* lunghezza seconda sequenza */
char aux;       /* variabile di appoggio per lo scambio */
int i, j, k, p, n1, m1;
Scanner sc = new Scanner(System.in);

do {
    System.out.print ("Lunghezza prima sequenza: ");
    n = sc.nextInt();
    sc.nextLine();
} while(n<1 || n>MAX_ELE);

/* Caricamento prima sequenza */
for(i=0; i<=n-1; i++) {
  System.out.print ("Immettere carattere n.");
  System.out.print (i+1);
  System.out.print (": ");
  vet1[i] = sc.nextLine().charAt(0);
}

do {
    System.out.print ("Lunghezza seconda sequenza: ");
    m = sc.nextInt();
    sc.nextLine();
} while(m<1 || m>MAX_ELE);

/* Caricamento seconda sequenza */
for(i=0; i<=m-1; i++) {
  System.out.print ("Immettere carattere n.");
  System.out.print (i+1);
  System.out.print (": ");
  vet2[i] = sc.nextLine().charAt(0);
}

/* Ordinamento prima sequenza */
p = n; n1=n;
do {
    k = 0;
    for(i=0; i<n1-1; i++) {
      if(vet1[i]>vet1[i+1]) {
        aux = vet1[i]; vet1[i] = vet1[i+1]; vet1[i+1] = aux;
        k = 1; p = i+1;
      }
    }
    n1 = p;
} while(k==1 && n>1);
```

```
    /* Ordinamento seconda sequenza */
    p = m; m1 = m;
    do {
         k = 0;
         for(i=0; i<m1-1; i++) {
           if(vet2[i]>vet2[i+1]) {
             aux = vet2[i]; vet2[i] = vet2[i+1]; vet2[i+1] = aux;
             k = 1; p = i+1;
           }
         }
         m1 = p;
    } while(k==1 && m1>1);

    /* Fusione delle due sequenze (merge) */
    i = 0; j = 0; k = 0;
    do {
        if(vet1[i]<=vet2[j])
          vet3[k++] = vet1[i++];
        else
          vet3[k++] = vet2[j++];
    }
    while(i<n && j<m);

    if(i<n)
      for(; i<n; vet3[k++] = vet1[i++])
        ;
    else
      for(; j<m; vet3[k++] = vet2[j++])
        ;

    /* Visualizzazione della fusione */
    for(i=0; i<k; i++)
      System.out.println (vet3[i]);
  }
}
```

In Figura 6.8 osserviamo il merge tra gli array ordinati `vet1` e `vet2` ordinati. L'operazione viene effettuata in due parti. La prima è data da:

```
i = 0; j = 0; k = 0;
do {
  if(vet1[i]<=vet2[j])
    vet3[k++] = vet1[i++];
  else
    vet3[k++] = vet2[j++];
} while(i<n && j<m);
```

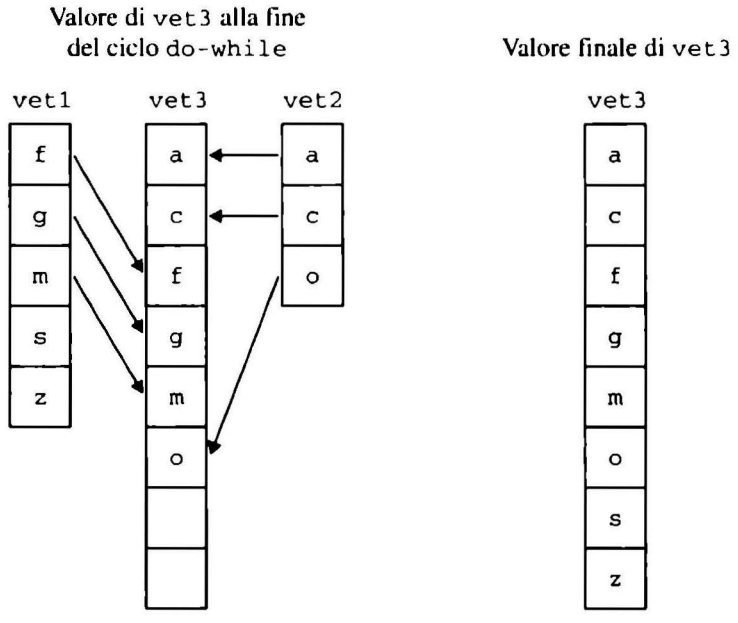


Figura 6.8 Risultato parziale e finale della fusione tra due vettori

Si controlla se l'*i*-esimo elemento di `vet1` è minore o uguale al *j*-esimo elemento di `vet2`, nel qual caso si aggiunge `vet1[i]` a `vet3` e si incrementa `i`. Nel caso contrario si aggiunge a `vet3` l'array `vet2[j]` e si incrementa `j`. In ogni caso si incrementa `k`, la variabile che indicizza `vet3`, perché si è aggiunto un elemento a `vet3`. Dal ciclo si esce quando `i` ha valore `n-1` o `j` ha valore `m-1`.

Si devono ancora aggiungere a `vet3` gli elementi di `vet1` (`j=m-1`) o di `vet2` (`i=n-1`) che non sono stati considerati. Nell'esempio precedente in `vet3` non ci sarebbero `s` e `z`. La seconda parte del merge ha proprio questo compito:

```
if(i<n)
  for(; i<n; vet3[k++] = vet1[i++])
    ;
else
  for(; j<m; vet3[k++] = vet2[j++])
    ;
```

Domande di verifica

1. Cosa hanno in comune tutti gli elementi di uno stesso array?
2. Qual è il significato di dimensioni e indici in riferimento agli array?

3. Qual è il valore dell'indice del primo elemento di un qualsiasi array monodimensionale?
4. Come si dichiara un vettore di dimensione 12? E una matrice 7 × 9?
5. Definire un array atto a contenere:
   a) un listino prezzi;
   b) i nomi dei giocatori di una squadra di calcio;
   c) lo stesso del caso b), ma occupando il minimo di memoria possibile.
6. Fare degli esempi di insiemi di valori che i vettori potrebbero adeguatamente contenere. Dichiarare gli array corrispondenti.
7. Per scorrere una matrice quadrata colonna per colonna, a iniziare dall'ultima, come deve essere il `for` che controlla il ciclo più esterno?
8. Qual è la formula per il calcolo del prodotto di due matrici?
9. Cos'è un algoritmo di ricerca?
10. Come opera la ricerca completa? È sempre possibile utilizzarla?
11. Cos'è un algoritmo di ordinamento? In che termini si misura la sua "bontà"?
12. Come opera l'ordinamento bubblesort nella prima versione presentata? Quali considerazioni permettono di ottimizzarlo?
13. Quali sono le differenze tra la prima versione di bubblesort e la sua versione ottimizzata? È sempre possibile quantificare il risparmio in termini di numero di cicli eseguiti indipendentemente dai valori presenti?
14. In quali situazioni è particolarmente vantaggioso utilizzare il bubblesort?
15. Come opera la ricerca binaria? È sempre possibile applicarla?
16. Quali vantaggi offre la ricerca binaria rispetto alla ricerca sequenziale?
17. Quanti confronti vengono in media effettuati per una ricerca binaria? E per una sequenziale?
18. Quanti confronti vengono al più effettuati in una ricerca binaria su un vettore di 2000 elementi? E in una completa?
19. Cos'è un algoritmo di fusione?
20. In quale sequenza due array ordinati vengono scanditi dall'algoritmo di fusione esaminato nel capitolo?
21. Spiegare con dettaglio come si effettua lo scambio dei valori di due variabili.

Esercizi

1. Scrivere un programma che, inizializzati in due vettori `a` e `b` della stessa lunghezza *n* valori interi, calcoli la somma incrociata degli elementi: `a[1]+b[n]`, `a[2]+b[n-1]`, ... la memorizzi nel vettore `c` e visualizzi quindi `a`, `b` e `c`.

2. Modificare il programma, esaminato nel presente capitolo, che determina il maggiore, il minore e la media degli elementi di un array, in modo che vengano diminuiti in media il numero di confronti effettuati nel ciclo durante l'esecuzione.

3. Scrivere un programma che inizializzi e quindi visualizzi un vettore con i valori alternati 0, 1, 0, 1, 0, 1, 0, 1, ... Ripetere l'esercizio con i valori 0, -3, 6, -9, 12, -15, 18, -21, ...

4. Scrivere un programma che, letti gli elementi di un vettore `vettore` e un numero `numero`, determini l'elemento di `vettore` più prossimo a `numero`.

5. Scrivere un programma che, inizializzato un vettore di `char` con una stringa di lettere dell'alfabeto e punteggiatura, visualizzi il numero complessivo delle vocali e delle consonanti del vettore.

6. Scrivere un programma di inizializzazione che, richiesto un elemento, controlli – prima di inserirlo nel vettore – se è già presente, nel qual caso chieda che l'elemento sia digitato di nuovo.

7. Scrivere un programma che inizializzi e quindi visualizzi una matrice di `int` in cui ciascun elemento è dato dalla somma dei propri indici.

8. [*Matrici simmetriche*] Una matrice quadrata $n \times n$ di un tipo qualsiasi si dice simmetrica se gli elementi simmetrici rispetto alla diagonale principale (dal vertice alto sinistro al vertice basso destro) sono due a due uguali. Scrivere un programma che, letta una matrice quadrata di interi, controlli se è simmetrica.

9. Scrivere un programma che, inizializzata una matrice $n \times n$, visualizzi la matrice che si ottiene da quella data scambiando le righe con le colonne.

10. Scrivere un programma che, letta una matrice di interi o reali, individui la colonna con somma degli elementi massima.

11. Scrivere un programma che richieda all'utente i voti delle otto prove sostenute durante l'anno da diciotto studenti di una classe e calcoli la media di ogni studente, la media di ogni prova e la media globale. Il programma dovrà infine visualizzare l'intera matrice e la media globale. [*Suggerimento*: si utilizzi una matrice di 19 righe e 9 colonne dove nelle prime otto vengono memorizzati in ciascuna riga i voti di uno studente e nella nona la rispettiva media; nella diciannovesima riga viene invece memorizzata la media per prova.]

12. Memorizzare in un array tridimensionale i numeri estratti al gioco del Lotto su tutte le ruote per dieci estrazioni consecutive. Verificare su quali ruote e in quali estrazioni si ripete un certo numero passato in ingresso dall'utente.

13. Predisporre un array adatto a conservare i valori della pressione del sangue di una persona misurata quattro volte al giorno per una settimana. Realizzare un programma per l'immissione delle misurazioni, il calcolo della media per giorno e complessiva, la visualizzazione dei valori e delle medie per giorno. [Ogni singola misurazione dovrebbe prevedere la raccolta di tre valori: la pressione minima, la pressione massima e la frequenza dei battiti cardiaci.]

14. Scrivere un programma di ordinamento in senso decrescente.

15. Scrivere un programma che carichi una matrice bidimensionale di caratteri e successivamente ricerchi al suo interno un valore passato in ingresso dall'utente. Il programma deve quindi restituire il numero di riga e di colonna relativo all'elemento cercato se questo è presente nella matrice, il messaggio `Elemento non presente` altrimenti.

16. Modificare il programma per la ricerca binaria esaminato nel capitolo in modo che visualizzi i singoli passi effettuati. Sperimentare il comportamento del programma con la ricerca dell'elemento `45` nel seguente vettore:

vet	21	33	40	41	45	50	60	66	72	81	88	89	91	93	99

17. Scrivere un programma che, richiesti i valori di un vettore ordinato in modo crescente, li inverta ottenendo un vettore decrescente. Si chiede di risolvere il problema utilizzando un solo ciclo.

18. Verificare il comportamento del programma di fusione applicato ai seguenti vettori:

vet1	3	31	41	43	44	45	80

vet2	5	8	21	23	46	51	60	66

19. Se il vettore è ordinato la ricerca completa può essere migliorata in modo da diminuire in media il numero di confronti da effettuare: come? Modificare in questo senso il programma esaminato nel presente capitolo.

20. Scrivere un programma che, richiesti all'utente i primi $n - 1$ elementi già ordinati di un vettore di dimensione n e un ulteriore elemento finale, inserisca quest'ultimo nella posizione corretta facendo "scivolare" verso il basso tutti gli elementi più grandi.

21. [*Insertion-sort*] Utilizzare l'algoritmo del precedente esercizio per scrivere un programma che ordini il vettore contemporaneamente all'inserimento dei dati da parte dell'utente.

22. Scrivere un programma che, richiesti all'utente i valori di una matrice, ne ordini tutte le colonne in senso crescente.

23. Scrivere un programma che, richiesti all'utente i valori di una matrice, ne ordini le righe in modo che un ipotetico vettore i cui elementi corrispondono alla somma delle righe della matrice risulti ordinato in senso crescente.

Capitolo 7

I metodi

Obiettivi didattici

- Sottoprogrammi, funzioni, metodi
- Dichiarare, definire e lavorare con propri metodi
- Visibilità e mascheramento dei nomi
- Valore di ritorno
- Tipo `void`
- Passaggio dei parametri
- Un metodo di input
- Ricorsione
- Gestione di una sequenza per mezzo di funzioni di immissione, ordinamento, ricerca ecc.

7.1 Il concetto di sottoprogramma

Già agli albori della programmazione ci si rese conto come spesso accada di dover codificare ripetitivamente identici gruppi di istruzioni. Supponiamo per esempio di voler accettare i dati anagrafici di un individuo per inserirli nella base dati esclusivamente con lettere maiuscole. Ecco allora che per ogni dato, nome, cognome, indirizzo, città, ecc., sarebbe stato necessario verificare che i caratteri accettati fossero lettere maiuscole, eventualmente effettuare delle sostituzioni tra minuscole e maiuscole oppure segnalare un errore. Poter scrivere una sola volta il codice da riutilizzare per tutti i dati accettati, non solo rendeva meno noioso realizzare i programmi, ma

riduceva la possibilità di errori insiti nella duplicazione e, cosa molto importante per i tempi, diminuiva la dimensione dei programmi e quindi della memoria necessaria per eseguirli.

Nacquero così i *sottoprogrammi* (subroutine) che consistevano in brani di codice con uno o più punti d'ingresso e uno o più punti di uscita. Con un'istruzione apposita, detta *chiamata del sottoprogramma* (subroutine call), si poteva provocare un salto a un punto d'ingresso del sottoprogramma stesso che era eseguito fino al primo punto di uscita incontrato, quindi il flusso del programma tornava all'istruzione successiva alla chiamata.

I sottoprogrammi si rivelarono utili anche per un altro motivo; con essi, infatti, era possibile suddividere un programma in unità più piccole, più facili da scrivere e manutenere, indipendentemente dal numero di volte che venivano eseguite. Per favorire questo tipo di approccio, il concetto di sottoprogramma si trasformò in quello di *funzione*.

Una funzione ha un unico punto d'ingresso e uno o più punti di uscita, inoltre rispetto a un generico sottoprogramma ha le seguenti caratteristiche:

- la possibilità di poter accettare in ingresso uno o più parametri passati dal chiamante;
- la possibilità di poter definire delle variabili visibili solo al suo interno;
- la possibilità di restituire un dato come risultato dell'elaborazione.

In Java, e nei linguaggi a oggetti, il concetto è stato ulteriormente esteso e le funzioni si sono evolute in metodi. In questo capitolo vedremo in che modo i metodi di Java si comportano come funzioni.

7.2 I metodi come funzioni

La forma generale della definizione di un metodo è la seguente

```
tipo-restituito nome-metodo ([tipo1 par1 [,tipo2 par2 ...
    [,tipoN parN]]]) {
    ...
    ...
}
```

La definizione stabilisce il nome del metodo `nome-metodo`, i valori in ingresso completi di tipo `tipo1 par1 ...`, detti *parametri formali*, il blocco di istruzioni che ne costituiscono il contenuto e il tipo del valore restituito `tipo-restituito`. Un metodo può non avere parametri formali nel qual caso le parentesi tonde non racchiudono nulla, come evidenziato nella sintassi dalle parentesi quadre. Per i nomi dei metodi valgono le consuete regole in uso per gli identificatori.

In Java possono esistere all'interno della stessa classe due metodi con lo stesso nome, a patto che abbiano parametri formali di tipo differente. Per questo motivo si definisce *firma* del metodo (method signature) l'insieme nome più parametri formali: è questa firma che identifica univocamente un metodo all'interno di una classe.

Tra le parentesi graffe di un metodo può essere inserita qualsiasi istruzione, compresa un'invocazione di metodo. Con l'istruzione `return` seguita dal valore restituito, il cui tipo deve corrispondere (o essere promovibile) a *`tipo-restituito`*, si indica un punto di uscita del metodo. Un metodo può anche non restituire nessun valore e in tal caso `return` non è seguita dal valore restituito e *`tipo-restituito`* deve essere la parola riservata `void`. Si consideri l'esempio seguente.

```
class MetodoCubo {
   public static void main (String argv[]) {
      float f = 3;
      double d = 4;
      double r;

      r = cubo (f);                              // linea 7
      System.out.print (f);
      System.out.print (" elevato al cubo e' ");
      System.out.println (r);                    // linea 10
      r = cubo (d);                              // linea 11
      System.out.print (d);
      System.out.print (" elevato al cubo e' ");
      System.out.println (r);                    // linea 14
   }
   static double cubo (float c) {                // linea 16
      System.out.println ("cubo con float");
      return c * c * c;
   }
   static double cubo (double c) {               // linea 20
      System.out.println ("cubo con double");
      return c * c * c;
   }
}
```

A linea 16 viene dichiarato il metodo `double cubo(float c)`. La parola chiave `static` indica che un metodo si comporta come una funzione. Si noti che a linea 20 viene definito un metodo che, pur essendo molto simile al precedente, non genera ambiguità poiché differisce nel parametro d'ingresso.

A linea 7 c'è invece l'invocazione del metodo. L'invocazione di un metodo può trovarsi in un qualsiasi punto dove è ammesso il valore che restituisce. L'istruzione a linea 10 potrebbe quindi essere sostituita con

```
System.out.println (cubo (f));
```

Risulta chiaro dunque come a linea 7 e a linea 11 vengano chiamati due diversi metodi e si intuiscono le ricadute che questa scelta ha sul linguaggio, anche se il nostro caso è puramente esemplificativo.

7.3 Visibilità

Una dichiarazione introduce un nome in un determinato ambito di definizione, detto *scope*. In altre parole ciò significa che un nome può essere usato o, come si usa dire, è *visibile* soltanto in una specifica parte del programma. Per un nome dichiarato all'interno di un blocco, detto nome *locale*, la visibilità si estende dal punto di dichiarazione alla fine del blocco in cui è contenuto. Questo è valido tanto per i blocchi che per esempio seguono un `if` o un ciclo, quanto per un blocco di un metodo. Anche i parametri formali di un metodo hanno un campo di visibilità che si estende dall'inizio alla fine del blocco istruzioni del metodo e sono quindi da considerarsi a tutti gli effetti variabili locali alla funzione. Nell'esempio

```
static void g (int y, char z) {
   int k;
   int i;
   ...
}
```

le variabili `y` e `z` sono locali al metodo `g` e hanno una visibilità che si estende dalla parentesi graffa aperta `{` alla parentesi graffa chiusa `}`. Quindi la definizione di `y` e `z` precede all'interno del blocco la definizione delle altre variabili locali `k` e `i`, che hanno anch'esse una visibilità che va dal punto di definizione alla fine del blocco. Per questo motivo, un metodo come il seguente

```
static void f (int x) {
   int x;
   ...
}
```

è errato, poiché si tenta di definire la variabile locale `x` dove ne esiste un'altra visibile con lo stesso nome.

Una dichiarazione di una variabile in un blocco non può avere lo stesso nome di una variabile locale dichiarata in un blocco più esterno, in quanto quest'ultima è ancora visibile nel blocco più interno. Nell'esempio

```
static void f () {
   int x = 1;
   {
      int x = 2;                    // Errore!!
      System.out.println (x);
   }
   System.out.println (x);
}
```

si ottiene un errore in compilazione poiché si tenta di ridefinire `x` in un blocco in cui è visibile già un'altra variabile con lo stesso nome. Vale appena la pena sottolineare che metodi diversi possono avere variabili locali con lo stesso nome, in quanto da un metodo non sono visibili le variabili locali degli altri metodi. Nell'esempio seguente

```
static void g () {
   {
      int x = 2;
   }
   System.out.println (x);     // Errore!!
}
```

l'errore è causato dal fatto che si tenta di visualizzare una variabile che non è più visibile.

Notare che in questi esempi si è potuto evitare l'utilizzo dell'istruzione `return` poiché i metodi sono stati dichiarati di tipo `void`, cioè che non restituiscono nessun valore. In tali casi viene assunto un `return` implicito al termine del blocco. L'istruzione `return` può lo stesso essere utilizzata, senza parametri, per interrompere l'esecuzione del metodo in un qualsiasi punto. Nell'esempio seguente

```
static void g (x) {
   if (x == 2)
      return;
   System.out.println (x);
}
```

il metodo `g` causa la visualizzazione di `x` tranne nel caso che valga 2.

Se un metodo è dichiarato di un tipo diverso da `void`, il `return` con il valore da restituire è obbligatorio, e anzi il compilatore si preoccupa di assicurarsi che esista un'istruzione di questo tipo per ciascun possibile flusso del programma. Nell'esempio

```
static double cubo (float c) {
   if (c != 3)
      return c * c * c;
}                          // Errore!!
```

otteniamo un errore in compilazione, in quanto non in tutti i possibili flussi viene eseguito un `return`. Modificando l'esempio precedente nel modo seguente, il codice viene compilato correttamente.

```
static double cubo (float c) {
   if (c != 3)
      return c * c * c;
   else
      return 0;
}
```

Notare che viene restituito un `int` anziché un `double`. In questi casi il compilatore effettua una promozione automatica e viene restituito il tipo corretto. Nell'esempio seguente viceversa

```
static int cubo (float c) {
   return c * c * c;          // Errore!!
}
```

la conversione da `double` a `int` non può essere effettuata automaticamente, in quanto si rischia una perdita di precisione, per cui viene segnalato un errore. Con un cast esplicito

```
static int cubo (float c) {
   return (int)(c * c * c);
}
```

la compilazione viene portata a termine senza problemi.

7.4 Leggere da tastiera

Negli esempi abbiamo spesso utilizzato il metodo `println` che ci ha permesso di verificare che le elaborazioni dei programmi fossero corrette e ci siamo avvalsi dei metodi della classe `Scanner` per leggere i dati digitati dall'operatore. Tutti gli esempi riportati fino a ora, e altri nel seguito, sono stati fatti utilizzando come dispositivo d'input/output un terminale a caratteri. Java e le sue classi sono state progettate per lavorare principalmente in modalità grafica, ma questo tipo di programmazione è assai più complessa e richiede l'utilizzo di concetti che devono ancora essere illustrati.

Vediamo comunque come sia possibile leggere i dati digitati sulla tastiera senza far ricorso alla classe `Scanner`, in modo da capire i meccanismi di basso livello che il linguaggio mette a disposizione. Quello che illustreremo è in realtà un metodo molto limitato, in quanto può prendere solo valori interi o caratteri singoli, ma comunque è sufficiente per comprendere i meccanismi di base. Vediamo dunque la seguente classe

```
class Input
{
   public static void main (String argv[]) {
      int inp = 0;
      do {
         inp = leggi(true);
         System.out.print ("hai inserito ");
         System.out.println (inp);
      } while (inp != 0);
      do {
         inp = leggi(false);
         System.out.print ("hai inserito ");
         System.out.println ((char)inp);
      } while (inp != 0);
   }

   static int leggi(boolean num) {
      byte b[] = new byte[9];
      int Return = 0;
      try {
```

```
            System.in.read(b);
        } catch (Exception e) { }
        if (num)
            for (int i = 0; i < b.length; i++)
                if (b[i] >= '0' && b[i] <= '9')
                    Return = Return * 10 + b[i] - '0';
                else
                    break;
        else
            if (b[0] >= ' ')
                Return = b[0];
        return Return;
    }
}
```

In questa classe abbiamo definito il metodo `leggiIntero`, che legge una riga dal terminale e restituisce o il numero intero digitato, se il parametro num vale `true`, o il primo carattere digitato, se il parametro vale `false`. Per la lettura viene utilizzato il metodo fornito con il linguaggio `System.in.read(b)` che riempie l'array di byte passato come argomento con i caratteri digitati dall'operatore. Il costrutto `try-catch` è stato inserito perché altrimenti il codice non viene compilato, ma verrà illustrato in seguito. A questo punto, se il parametro num vale `true` si esegue un ciclo sugli elementi dell'array `b` fintanto che sono compresi tra 0 e 9. Ogni volta che si trova un valore accettabile, si eseguono le seguenti operazioni:

- si moltiplica per 10 il valore della variabile `Return` per tenere conto della posizione: alla prima iterazione `Return` vale 0 per cui tale prodotto restituisce 0;
- si sottrae dal codice del carattere letto il codice del carattere '0' (48) per ottenere un intero compreso tra 0 e 9 corrispondente al numero digitato;
- si sommano questi due valori e si memorizza il risultato nella stessa variabile `Return`.

Non appena si trova il primo carattere che non corrisponde a una cifra, si esce dal ciclo e si restituisce il valore della variabile `Return`. Da sottolineare di nuovo il fatto che, poiché Java distingue tra maiuscole e minuscole, possiamo dichiarare una variabile di nome `Return` senza che questo causi un'ambiguità con l'istruzione `return`.

Se il parametro num vale `false`, viene verificato se il primo carattere digitato ha un codice uguale o superiore a spazio, poiché i codici inferiori tipicamente corrispondono a caratteri di controllo non visualizzabili. Se tale carattere è visualizzabile, esso viene restituito, altrimenti viene restituito 0.

Il codice contenuto nel metodo `main` visualizza tutti i numeri inseriti fino a che il metodo `leggiIntero` non restituisce 0, dopodiché visualizza tutti i primi caratteri delle stringhe inserite finché non si inserisce una riga nulla o un carattere non visualizzabile. Nella seconda `println` è stato utilizzato un cast per ottenere la visualizzazione del carattere e non del suo codice.

7.5 Invocazione di un metodo

Un metodo viene invocato in Java facendo riferimento al nome e passandogli una lista di parametri conforme in tipo, numero e ordine alla lista dei parametri formali elencata nella definizione del metodo. Nell'esempio seguente

```
class Input
{
   public static void main (String argv[]) {
      float b, h;
      double a;
      char p;

      System.out.print ("Inserire poligono
                         (T)riangolo/(R)ettangolo : ");
      p = (char)leggi (false);
      System.out.print ("Inserire base : ");
      b = leggi (true);
      System.out.print ("Inserire altezza : ");
      h = leggi (true);
      a = area (b, h, p);
      System.out.print ("Il poligono ha l'area = ");
      System.out.println (a);
   }

   static double area (float base, float altezza,
                                   char poligono) {
      switch (poligono) {
      case 'T':
         return base * altezza / 2.0;
      case 'R':
         return base * altezza;
      default:
         return 0;
      }
   }

   static int leggi(boolean num) {
      byte b[] = new byte[9];
      int Return = 0;
      try {
         System.in.read(b);
      } catch (Exception e) { }
      if (num)
         for (int i = 0; i < b.length; i++)
            if (b[i] >= '0' && b[i] <= '9')
```

```
                Return = Return * 10 + b[i] - '0';
            else
                break;
        else
            if (b[0] >= ' ')
                Return = b[0];
        return Return;
    }
}
```

il contenuto delle variabili di tipo `float` `b`, `h` e di tipo `char` `p` viene passato al metodo `area` per mezzo dell'istruzione

```
a = area (b, h, p);
```

Le variabili `b`, `h` e `p` sono dette *parametri attuali*, poiché contengono i valori di ingresso di quella specifica invocazione di metodo che permettono di calcolare l'area del poligono. Si osservi come tipo, numero e ordine dei parametri debbano essere coerenti con quelli stabiliti dalla definizione dei parametri formali del metodo. Al posto di una variabile si può comunque passare una costante dello stesso tipo, per esempio

```
a = area (b, h, 'T');
```

dove `'T'` viene immesso nel parametro formale `poligono`. Non sono invece corrette le seguenti invocazioni

```
int x = 'T';

a = area (b, 'T', h);       // Errore! ordine errato
a = area (b, h);            // Errore! Numero parametri errato
a = area (b, h, 'T', x);    // Errore! Numero parametri errato
a = area (b, h, x); // Errore! Tipo errato (x necessita un cast)
```

Il passaggio di parametri è errato o per ordine o per numero o per discordanza di tipo.

7.6 Passaggio dei parametri

Abbiamo distinto due tipi di parametri: i parametri *formali* e i parametri *attuali*. I parametri formali sono quelli dichiarati per tipo, numero e ordine nella definizione del metodo, mentre i parametri attuali sono quelli che vengono passati al metodo all'atto della chiamata. In Java il passaggio dei parametri avviene sempre e soltanto *per valore*; ciò significa che, all'atto dell'invocazione di un metodo, ogni parametro formale è inizializzato con il valore del corrispondente parametro attuale. In pratica è come se all'atto dell'invocazione del metodo a ogni parametro formale fosse assegnato il corrispondente parametro attuale per mezzo dell'operatore =. Fra parametri formali e attuali non è quindi necessario che esista una corrispondenza di tipo puntuale, ma

deve esistere una corrispondenza tale da permettere la corretta esecuzione degli assegnamenti. Rifacendosi agli esempi precedenti, il codice

```
a = area (3, 4, 'T');
```

non crea nessun problema, anche se i primi due parametri sono di tipo `int` anziché `float`, mentre se scriviamo

```
a = area (3.0, 4.0, 'T');
```

otteniamo un errore in compilazione, in quanto i primi due parametri sono di tipo `double` e non è consentito assegnare un `double` a un `float` senza un cast.

Il fatto che i parametri attuali vengano copiati nei parametri formali implica che qualsiasi modifica venga fatta su questi ultimi, non va a incidere in nessun modo sui parametri attuali. Nell'esempio

```
class Parametri1
{
   public static void main (String argv[]) {
      double d = 30;
      f (d);
      System.out.println (d);
   }
   static void f (double d) {
      d = 0;
   }
}
```

la variabile `d` non viene modificata dal metodo `f` per cui viene visualizzato 30.

È opportuno sottolineare che quando un parametro è di tipo array, viene passata per valore la referenza. Abbiamo già visto che assegnando una variabile di tipo array a un'altra, entrambe le variabili si riferiscono alla stessa area di memoria che contiene gli elementi dell'array. Lo stesso accade nel passaggio di parametri a un metodo, per cui quest'ultimo può effettivamente modificare il contenuto dell'array. Modificando l'esempio precedente

```
class Parametri2
{
   public static void main (String argv[]) {
      double d[] =  new double[1];
      d[0] = 30;
      f (d);
      System.out.println (d[0]);
   }
   static void f (double d[]) {
      d[0] = 0;
   }
}
```

viene visualizzato 0. Vedremo in seguito altri tipi di variabili referenza e anche per loro varrà quanto detto.

Un parametro attuale può essere anche un'espressione o il risultato dell'invocazione di un metodo. Quando si invoca un metodo con due o più parametri, può accadere che il loro valore dipenda dall'ordine di valutazione degli argomenti. Nell'invocazione

```
int i = 0;
m (i++, i++);
```

se l'interprete valuta gli argomenti da sinistra verso destra, il metodo m viene invocato con i parametri attuali 0 e 1, mentre se l'interprete valuta gli argomenti da destra verso sinistra, il metodo viene invocato con argomenti 1 e 0. In C per esempio l'ordine di valutazione degli argomenti non è specificato a livello di linguaggio e varia quindi da implementazione a implementazione. In Java viceversa l'ordine di valutazione degli argomenti è specificato ed è da sinistra verso destra, per cui le istruzioni precedenti equivalgono a

```
int i = 0;
m (i, i + 1);
i += 2;
```

Questa seconda notazione in ogni caso permette una comprensione più immediata delle intenzioni del programmatore, per cui è da preferirsi anche se contiene un'istruzione in più.

7.7 Metodi ricorsivi

All'interno di un metodo si possono chiamare altri metodi. Nel caso particolare in cui un metodo invochi se stesso, si parla di *ricorsione*. La tecnica della ricorsione viene usata in sostituzione della creazione di cicli (*iterazione*), tipicamente per scorrere degli alberi binari, in quanto permette di scrivere algoritmi più semplici e comprensibili. Questa maggiore semplicità si paga in termini di efficienza, ma molto spesso ne vale la pena.

Realizzare programmi per la gestione degli alberi binari ci porterebbe troppo lontano e comunque esula dagli scopi del libro, per cui vedremo altri esempi iniziando dal più classico riportato in letteratura: il calcolo del fattoriale. Utilizziamo questo esempio perché è molto semplice, anche se purtroppo non evidenzia appieno i vantaggi dell'utilizzo della ricorsione.

Abbiamo già visto in un capitolo precedente il significato di fattoriale e alcuni modi di calcolarlo utilizzando i cicli. La seguente versione utilizza invece la ricorsione.

```
class Fattoriale4
{
   public static void main (String argv[]) {
      int n = 5;

      System.out.print ("Il fattoriale di ");
      System.out.print (n);
      System.out.print ("  è  ");
```

```
        System.out.println (fattoriale (n));
    }
    static int fattoriale (int x) {
        if (x <= 1)
            return 1;
        else
            return x * fattoriale(x - 1);
    }
}
```

Se si confronta questo codice con quello illustrato nel capitolo precedente, si nota che la soluzione ricorsiva corrisponde in modo più diretto con la definizione di fattoriale. Esaminiamo da vicino gli ambienti generati dalle successive invocazioni del metodo ricorsivo. Supponiamo di chiamare il metodo `fattoriale` con argomento 5: poiché x è maggiore di 1, viene immediatamente invocato di nuovo il metodo con argomento 4 (`x-1`), e così via fin tanto che `fattoriale` non viene invocato con argomento 1. A questo punto abbiamo 5 invocazioni del metodo aperte, ciascuna delle quali ha una propria variabile locale x contenente l'argomento, che è un numero compreso tra 1 e 5, come illustra il seguente schema

```
fattoriale (5)
    fattoriale (4)
        fattoriale (3)
            fattoriale (2)
                fattoriale (1)
```

Quando avviene l'ultima invocazione, l'espressione `x<=1` restituisce `true` per cui l'ultimo ambiente si chiude restituendo al precedente 1; si chiudono poi in successione tutti gli ambienti aperti restituendo al precedente il valore calcolato.

La zona di memoria riservata alle chiamate viene gestita con la logica di una *pila* (stack): a ogni invocazione di `fattoriale`, il sistema alloca uno spazio di memoria libero in testa alla pila riservato al suo parametro formale x. Man mano che i metodi restituiscono un valore, la pila viene liberata, sempre a partire dalla testa.

Il fattoriale è una funzione che cresce molto rapidamente al crescere dell'argomento, più di una funzione esponenziale; poiché il nostro metodo accetta parametri e restituisce valori di tipo `int`, il codice mostrato funziona correttamente per valori di x piuttosto bassi. Per lavorare con numeri più alti, è necessario che il metodo accetti parametri e restituisca valori di tipo `long` o `double`.

7.8 Permutazioni, disposizioni, combinazioni

Osserviamo alcune procedure di *calcolo combinatorio* allo scopo di progettare metodi ricorsivi.

Si definiscono *permutazioni semplici* di *n* oggetti distinti i gruppi che si possono formare in modo che ciascuno contenga tutti gli *n* oggetti dati e che differisca dagli altri soltanto per l'ordine in cui vi compaiono gli oggetti stessi. Dati due oggetti `e1`,

e2, si possono avere solamente due permutazioni; se gli oggetti sono tre, le permutazioni semplici diventano sei; se gli oggetti sono quattro (e1 e2 e3 e4), si hanno le seguenti ventiquattro possibilità:

```
e1 e2 e3 e4     e1 e2 e4 e3     e2 e1 e3 e4     e1 e2 e4 e3
e3 e1 e2 e4     e3 e1 e4 e2     e1 e3 e2 e4     e1 e3 e4 e2
e2 e3 e1 e4     e2 e3 e4 e1     e3 e2 e1 e4     e3 e2 e4 e1
e1 e4 e2 e3     e1 e4 e3 e2     e2 e4 e1 e3     e2 e4 e3 e1
e3 e4 e1 e2     e3 e4 e2 e1     e4 e1 e2 e3     e4 e1 e3 e2
e4 e2 e1 e3     e4 e2 e3 e1     e4 e3 e1 e2     e4 e3 e2 e1
```

In generale, il numero di permutazioni semplici P di n oggetti è dato $Pn = n!$, da cui risultano appunto, nel nostro caso, 4! = 24 possibilità distinte. Il metodo corrispondente lo abbiamo appena scritto. Se desideriamo conoscere il numero di permutazioni di 13 oggetti è sufficiente mandare in esecuzione il programma appena visto per il calcolo del fattoriale, con l'accortezza di utilizzare un tipo dati adeguato, per scoprire che sono: 6.227.020.800.

Dati n oggetti distinti e detto k un numero intero positivo minore o uguale a n, si chiamano invece *disposizioni semplici*, presi k a k di questi n oggetti, i gruppi distinti che si possono formare in modo che ogni gruppo contenga soltanto k oggetti e che differisca dagli altri o per qualche oggetto o per l'ordine in cui gli oggetti stessi sono disposti. Le disposizioni di quattro oggetti ($n = 4$) presi uno a uno ($k = 1$) sono dunque i gruppi che contengono un solo oggetto:

```
e1      e2      e3      e4
```

cioè in totale quattro. Le disposizioni di quattro oggetti presi due a due ($k = 2$) sono invece 12:

```
e1 e2     e2 e1     e3 e1     e4 e1
e1 e3     e2 e3     e3 e2     e4 e2
e1 e4     e2 e4     e3 e4     e4 e3
```

Il calcolo delle disposizioni usa la formula generale:

$$D_{n,k} = n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot (n - k + 2) \cdot (n - k + 1)$$

Nel caso di $n = 4$ e $k = 1$ verifichiamo

$$D_{4,1} = 4$$

e nel caso di n = 4 e k = 2:

$$D_{4,2} = 4 \cdot 3 = 12$$

Possiamo dunque scrivere un metodo ricorsivo che calcoli le disposizioni semplici:

```
static int dispo (int k, int n, int m) {
   if(n==m-k)
      return(1);
   else
```

```
      return(n*dispo(k, n-1, m));
}
```

Al momento della prima invocazione di `dispo`

```
dispo(k, n, n);
```

dobbiamo passare al metodo, oltre ai valori di `k` e di `n`, anche un ulteriore valore `n` (che diventa il parametro formale `m`) perché esso possa conoscere il numero totale degli oggetti e, quindi, effettuare il controllo `n=m-k`, dato che a ogni ulteriore chiamata il parametro formale `n` viene decrementato di una unità rispetto al precedente.

```
import java.util.Scanner;
class Disposizioni {
  public static void main (String argv[]) {
    int n, k;
    Scanner sc = new Scanner(System.in);
    System.out.print ("Dammi n: ");
    n = sc.nextInt();
    System.out.print ("Dammi k: ");
    k = sc.nextInt();
    System.out.print ("Disposizioni semplici di ");
    System.out.print (k);
    System.out.print (" su ");
    System.out.print (n);
    System.out.print (" oggetti: ");
    System.out.print (dispo(k, n, n));
  }

   /* Calcolo delle disposizioni semplici di n oggetti
    * presi k a k                                          */
   static int dispo (int k, int n, int m) {
     if(n==m-k)
       return(1);
     else
       return(n*dispo(k, n-1, m));
   }
}
```

Un'altra versione, più sintetica ed elegante, del metodo `dispo` è la seguente:

```
static int dispo(int k, int n) {
   if(k==1) return(n);
   else return(n*dispo(k-1, n-1));
}
```

Osserviamo che il calcolo delle disposizioni è simile a quello del fattoriale. In particolare, le disposizioni di n elementi presi n a n sono proprio pari a $n!$:

$D_{n,n} = n!$

Nel caso fosse $n = 4$ e $k = 4$ avremmo:

$D_{4,4} = 24$

Possiamo allora scrivere un nuovo metodo `dispo2` che sfrutti il metodo `fattoriale` precedentemente definito:

```
/* Calcolo delle disposizioni semplici utilizzando
 * il metodo  per il calcolo delle permutazioni     */
static int dispo2(int k, int n)
{
  return(fattoriale(n)/fattoriale(n-k));
}
```

Infatti $D_{n,k}$ = `fattoriale(n)/fattoriale(n-k)`, come si può facilmente dedurre dal confronto delle due formule.

Si chiamano *combinazioni semplici* di n oggetti distinti, presi k a k ($k \leq n$) i gruppi di k oggetti che si possono formare con gli n oggetti dati, in modo che i gruppi stessi differiscano tra loro almeno per un oggetto. Per esempio, i quattro elementi `e1`, `e2`, `e3` ed `e4`, presi due a due, danno origine alle seguenti sei combinazioni:

```
e1 e2    e1 e3    e1 e4    e2 e3    e2 e4    e3 e4
```

La formula generale che consente di calcolare il numero delle combinazioni è

$C_{n,k} = D_{n,k} / k!$

Dunque il numero di combinazioni di n oggetti presi k a k è uguale al numero di disposizioni di n oggetti presi k a k, diviso k fattoriale.

Il metodo `comb`, che calcola il numero di combinazioni semplici possibili, può richiamare `dispo` per calcolare le disposizioni $D_{n,k}$ e `fattoriale` per calcolare il fattoriale, passando `k` come numero di elementi:

```
static int comb (int k, int n) {
   return(dispo(k, n, n)/fattoriale(k));
}
```

Ecco il programma completo per le combinazioni semplici.

```
import java.util.Scanner;
class Combinazioni {
  public static void main (String argv[]) {
    int n, k;
    Scanner sc = new Scanner(System.in);
    System.out.print ("Dammi n: ");
    n = sc.nextInt();
    System.out.print ("Dammi k: ");
    k = sc.nextInt();
    System.out.print ("Le combinazioni semplici di ");
    System.out.print (n);
```

```
    System.out.print (" su ");
    System.out.print (k);
    System.out.print (" sono: ");
    System.out.print (comb(k, n));
  }

  /* Calcolo delle combinazioni semplici di n oggetti
   * presi k a k                                          */
  static int comb (int k, int n) {
      return(dispo(k, n, n)/fattoriale(k));
  }

  static int dispo (int k, int n, int m) {
     if(n==m-k)
        return(1);
     else
        return(n*dispo(k, n-1, m));
  }
  static int fattoriale (int n) {
     if (n <= 1)
        return 1;
     else
        return n * fattoriale(n - 1);
  }
}
```

Una prima alternativa è quella di utilizzare in `comb` soltanto il metodo `dispo`, dato che $D_{k,k}$ è uguale a $k!$:

```
/* Calcolo delle combinazioni semplici utilizzando soltanto
 * il metodo per il calcolo delle disposizioni          */
static int comb (int k, int n) {
     return(dispo(k, n, n)/dispo(k, k, k));
}
```

Una seconda possibilità si ottiene sfruttando il metodo `dispo2` che, come abbiamo visto in precedenza, utilizzava a sua volta `fattoriale` per calcolare le disposizioni:

```
/* Calcolo delle combinazioni semplici
 * utilizzando dispo2 e fattoriale    */
static int comb(int k, int n) {
   return(dispo2(k, n)/fattoriale(k));
}

/* Calcolo delle disposizioni semplici
 * utilizzando fattoriale             */
static int dispo2(int k, int n) {
    return(fattoriale(n)/fattoriale(n-k));
}
```

Così facendo abbiamo decomposto le formule risolutive di combinazioni e disposizioni rimandando il problema al calcolo del fattoriale.

7.9 Fibonacci, torre di Hanoi

Nella *successione di Fibonacci* ogni termine è ottenuto addizionando i due che lo precedono; il termine generico è pertanto

$F(n) = F(n - 1) + F(n - 2)$

dove n è un numero intero maggiore o uguale a 2. Inoltre $F(0) = 0$ e $F(1) = 1$. Dunque si ha:

$F(2) = F(2 - 1) + F(2 - 2) = F(1) + F(0) = 1 + 0 = 1$
$F(3) = F(3 - 1) + F(3 - 2) = F(2) + F(1) = 1 + 1 = 2$
$F(4) = F(4 - 1) + F(4 - 2) = F(3) + F(2) = 2 + 1 = 3$
$F(5) = F(5 - 1) + F(5 - 2) = F(4) + F(3) = 3 + 2 = 5$
$F(6) = F(6 - 1) + F(6 - 2) = F(5) + F(4) = 5 + 3 = 8$
$F(7) = F(7 - 1) + F(7 - 2) = F(6) + F(5) = 8 + 5 = 13$
...

È evidente il carattere ricorsivo di tale definizione: ogni chiamata di `fibo` genera due ulteriori chiamate ricorsive, una passando il parametro attuale `n-1`, l'altra `n-2`; dunque al crescere del valore di n la memoria tende rapidamente a saturarsi.

```
import java.util.Scanner;
class Fibonacci {
  public static void main (String argv[]) {
    Scanner sc = new Scanner(System.in);
    int n, i;
    System.out.println ("Successione di Fibonacci f(0)=1
                         f(1)=1 f(n)=f(n-1)+f(n-2)");
    System.out.print ("Dammi n: ");
    n = sc.nextInt();
    for (i=0;i<=n;i++) {
      System.out.print (fibo(i));
      System.out.print (" ");
    }
}

   /* Calcolo dei numeri di fibonacci */
   static int fibo (int n) {
      if(n==0)       return(1);
      else if(n==1) return(1);
         else      return(fibo(n-1)+fibo(n-2));
   }
}
```

Abbiamo introdotto le tecniche della programmazione ricorsiva, applicandole a problemi quali il calcolo combinatorio. In tutti questi casi si tratta di problemi di cui non è però difficile trovare anche delle soluzioni iterative equivalenti: forse non altrettanto eleganti, ma certo non meno efficienti di quelle ricorsive. Occorre infatti ricordare che l'uso di procedure ricorsive non è, nella scrittura di un algoritmo, indispensabile, ed esistono specifici algoritmi per convertire un qualsiasi programma ricorsivo nel suo equivalente iterativo. Nonostante ciò vi sono numerosi esempi di problemi che, benché ammettano soluzioni ricorsive relativamente semplici, hanno versioni strettamente iterative laboriose e di difficile lettura, così che se ne preferisce, come nel calcolo del fattoriale, la versione ricorsiva.

Approcciamo ora un antico passatempo orientale che evidenzia così non solo la grande potenza e semplicità delle tecniche ricorsive, ma anche la loro estrema flessibilità e adattabilità alle più diverse situazioni.

Alla fine dell'Ottocento riscosse un notevole successo, inizialmente in Francia e poi in tutta Europa, un passatempo orientale apparentemente insolubile noto come la *torre di Hanoi*. L'abile campagna pubblicitaria usata per commercializzare il gioco narrava che i sacerdoti del tempio buddista di Hanoi dedicavano gran parte del loro tempo alla soluzione del seguente rompicapo:

> *Nel tempio vi sono tre pioli e 64 dischi d'oro di grandezze tutte diverse, con un foro nel centro. Inizialmente i 64 dischi, impilati uno sull'altro in ordine di dimensione crescente dal più grande al più piccolo, formano una torre attorno al primo piolo* (Figura 7.1). *Lo scopo del gioco è di spostare la torre dal primo al secondo piolo muovendo un solo disco alla volta e senza che un disco più grande venga mai posto sopra ad uno più piccolo. Il terzo piolo deve naturalmente essere utilizzato come piolo d'appoggio per i trasferimenti.*

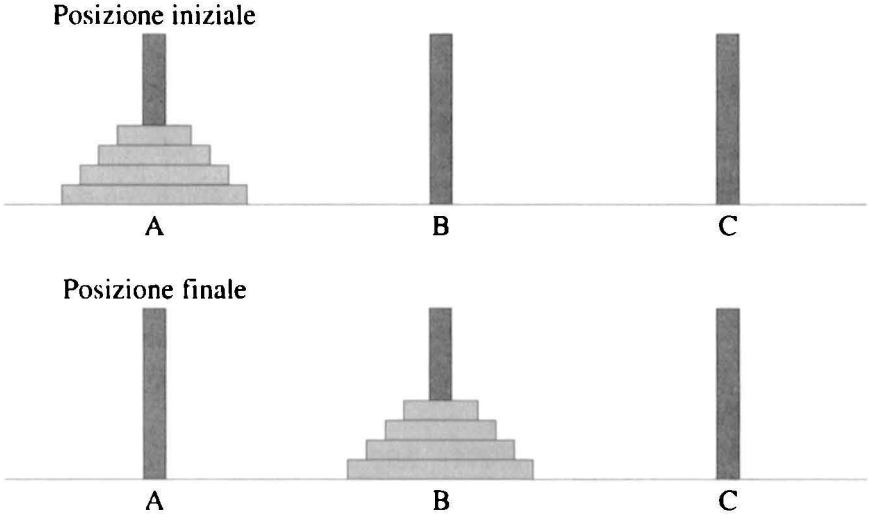


Figura 7.1 La torre di Hanoi in versione semplificata (4 dischi)

Proviamo dunque a scrivere un programma che descriva passo per passo i movimenti che devono essere effettuati per spostare la torre dal primo al secondo piolo.

Poiché il disco più grande non può essere appoggiato su nessun altro disco, non è difficile capire che per spostarlo dal primo al secondo piolo è necessario prima spostare tutti gli altri dischi sul terzo piolo, dove questi formeranno a loro volta una torre di 63 piani. Generalizzando il problema a un qualsiasi numero *n* di dischi e indicando per comodità i tre pioli con le lettere A, B e C, l'algoritmo risolutivo potrà essere impostato secondo il seguente schema:

- spostare la torre composta dai primi *n* - 1 dischi dal piolo A al piolo C;
- spostare il disco *n*-esimo dal piolo A al piolo B;
- spostare la torre di *n* - 1 dischi dal piolo C al piolo B.

Come è facile vedere, la soluzione del problema per *n* dischi è stata ricondotta alla soluzione dello stesso problema per *n* - 1 dischi. Questa osservazione ci consente quindi di ricavare una soluzione ricorsiva al problema della torre di Hanoi; una soluzione estremamente semplice e naturale, tanto che il programma finale è molto più compatto e di facile comprensione degli equivalenti algoritmi iterativi.

```
/* Programma ricorsivo per risolvere la torre di Hanoi */
import java.util.Scanner;
class Hanoi {
  public static void main (String argv[]) {
    Scanner sc = new Scanner(System.in);
    int mossa=0, DISCHI=4;
    System.out.println ("Mosse da eseguire per spostare");
    System.out.print (DISCHI);
    System.out.println (" dischi");
    mossa = hanoi(mossa, DISCHI, 'A', 'B', 'C');
  }

   /* Metodo ricorsivo "hanoi" per spostare una torre
    * di n dischi da pioloP a pioloA usando aus come
    * piolo di ausilio                                  */
   static int hanoi(int mossa, int n, char pioloP,
                                 char pioloA, char aus) {
      if(n==1) mossa = muovi(mossa, 1, pioloP, pioloA);
      else {
         mossa = hanoi(mossa, n - 1, pioloP, aus, pioloA);
         mossa = muovi(mossa, n, pioloP, pioloA);
         mossa = hanoi(mossa, n - 1, aus, pioloA, pioloP);
      }
      return(mossa);
   }
```

```
    /* Metodo "muovi" per spostare il disco nd
     * dal piolo di partenza pP al piolo di arrivo pA */
    static int muovi(int mossa, int nd, char pP, char pA) {
        char invio;
        mossa = mossa + 1;
        System.out.println("");
        System.out.print(mossa);
        System.out.print(": muovere disco ");
        System.out.print(nd);
        System.out.print(" da ");
        System.out.print(pP);
        System.out.print(" a ");
        System.out.print(pA);
        return(mossa);
    }
}
```

Il programma è facilmente comprensibile e non necessita di particolari spiegazioni. Dopo aver inizializzato a zero il numero delle mosse già effettuate e visualizzato l'intestazione dell'output, il programma richiama la procedura ricorsiva `hanoi` che nel caso di un solo disco (`n==1`) si limita a spostarlo dalla torre A alla torre B con il metodo `muovi`. Se la torre iniziale è invece composta da più dischi, `hanoi` richiama se stesso per spostare i primi `n-1` dischi della torre sul piolo ausiliare, sposta quindi l'ultimo disco e infine richiama nuovamente se stessa per spostare sul piolo di destinazione la torre che ora si trova sul piolo ausiliare. In Figura 7.2 vediamo l'output applicato a una torre di 4 dischi.

```
Mosse da eseguire per spostare 4 dischi
---------------------------------------
 1: Muovere disco 1 da A a C
 2: Muovere disco 2 da A a B
 3: Muovere disco 1 da C a B
 4: Muovere disco 3 da A a C
 5: Muovere disco 1 da B a A
 6: Muovere disco 2 da B a C
 7: Muovere disco 1 da A a C
 8: Muovere disco 4 da A a B
 9: Muovere disco 1 da C a B
10: Muovere disco 2 da C a A
11: Muovere disco 1 da B a A
12: Muovere disco 3 da C a B
13: Muovere disco 1 da A a C
14: Muovere disco 2 da A a B
15: Muovere disco 1 da C a B
```

Figura 7.2 Il risultato dell'esecuzione del programma con `DISCHI = 4`

È infine interessante dare, anche in considerazione di quanto detto a proposito degli algoritmi ricorsivi, una valutazione del tempo necessario per eseguire il programma. A questo proposito indichiamo con $f(n)$ il numero degli spostamenti che devono essere effettuati per portare dal piolo A al piolo B una torre composta da n dischi. Come abbiamo visto, per questo è necessario prima spostare $n - 1$ dischi dal piolo A al piolo C, quindi spostare l'ultimo disco sul piolo B e infine spostare nuovamente $n - 1$ dischi, stavolta dal piolo C al piolo B. Abbiamo così l'equazione ricorsiva

$$f(n) = f(n - 1) + 1 + f(n - 1) = 2 \cdot f(n - 1) + 1$$

Da cui con semplici calcoli, applicando ripetutamente la precedente uguaglianza e ricordando che $f(1) = 1$, otteniamo

$$f(n) = 2^{n-1} + 2^{n-2} + \ldots + 2^2 + 2 + 1$$

ovvero la funzione esponenziale

$$f(n) = 2^n - 1$$

In particolare, calcolando il valore della funzione per $n = 4$ abbiamo 15 spostamenti (come possiamo infatti verificare anche dall'output di Figura 7.2), mentre per $n = 64$, il numero dei dischi che formavano la torre del gioco originario, abbiamo

$$f(64) = 2^{64} - 1$$

che equivale a circa $0.2 \cdot 10^{20}$ spostamenti. Anche ipotizzando che un monaco – certamente ben allenato – riesca a spostare un disco al secondo senza mai stancarsi, il tempo necessario per muovere l'intera torre di Hanoi dal primo al secondo piolo sarebbe approssimativamente pari a $0.2 \cdot 10^{20}$ secondi. Poiché un anno corrisponde a circa $0.3 \cdot 10^8$ secondi, il tempo impiegato dai monaci per spostare la torre sarebbe dunque nell'ordine di alcune centinaia di miliardi di anni!

Come per tutti gli algoritmi caratterizzati da una funzione esponenziale non si deve però credere che le prestazioni migliorino utilizzando una macchina: anche con i più veloci calcolatori oggi a disposizione l'esecuzione di `hanoi` con `DISCHI = 1000` richiede infatti tempi ben superiori alla durata prevista per l'universo e l'algoritmo, per quanto concettualmente semplice e sintatticamente corretto, è quindi applicabile solo per valori relativamente piccoli.

7.10 Gestione di una sequenza

In questo paragrafo consideriamo il problema di far gestire all'utente una sequenza di interi mediante il seguente menu:

```
      GESTIONE SEQUENZA
1. Immissione
2. Ordinamento
3. Ricerca completa
4. Ricerca binaria
5. Visualizzazione
0. Fine
Scegliere una opzione:
```

Le opzioni possono essere scelte un numero di volte qualsiasi, finché non si seleziona la numero zero, che fa terminare il programma. Ovviamente, innanzitutto si deve scegliere la prima opzione per immettere la sequenza, successivamente questa possibilità può essere sfruttata per lavorare su altre sequenze.

Il programma visualizza il menu, recepisce la scelta dell'utente ed effettua l'azione corrispondente. Viene naturale far corrispondere a ogni opzione un metodo che svolga il compito stabilito.

```
static int immissione(int MAX_ELE, int vet[]) {..}
static void ordinamento(int n, int vet[]) {
static int ricerca ( int n, int ele, int vet[] ) {..}
static int ricBin( int n, int ele, int vet[] ) {..}
static void visualizzazione( int n, int vet[] ) {..}
```

Dato che tutti gli algoritmi relativi sono stati visti nel Capitolo 6, adesso ci soffermiamo soltanto sull'uso dei metodi e sul passaggio dei parametri. Avremo un array che conterrà la sequenza

```
int vet[] = new int[MAX_ELE]; /* array che ospita la sequenza */
```

e un ciclo `while` che presenta il menu e scatena il metodo corrispondente, fino a che l'utente non sceglie l'opzione 0.

```
while(scelta != 0) {
      System.out.println ("\n\n\n   GESTIONE SEQUENZA");
      System.out.println ("1. Immissione");
      System.out.println ("2. Ordinamento");
      ...
      System.out.print ("\nScegliere una opzione: ");
      scelta = sc.nextInt();
      switch (scelta) {
        case 1: n = immissione( MAX_ELE, vet );
                break;
        case 2: ordinamento( n, vet );
                break;
        ...
```

Nel caso sia selezionata la prima opzione viene immesso il valore 1 nella variabile intera `scelta` e, per mezzo del costrutto `switch-case`, è mandato in esecuzione `immissione`:

```
n = immissione( MAX_ELE, vet );
```

Al metodo `immissione` sono passati in ingresso il numero massimo di elementi che l'array può contenere e il riferimento all'array stesso e deve ritornare il numero di valori effettivamente immessi in modo che renderlo noto al chiamante; tale valore viene memorizzato nella variabile `n`. Si noti come coerentemente il valore di ritorno di `immissione` sia stato definito di tipo `int`.

Al metodo `ordinamento` deve essere passato il numero di elementi effettivi della sequenza e il riferimento all'array che la contiene:

```
case 2: ordinamento( n, vet );
```

Anch'esso agisce su `vet[]` ordinando i suoi elementi, ma non restituisce alcun valore: infatti il suo valore di ritorno è descritto come `void`.

Nel caso di scelta 3 al metodo `ricerca` deve essere passato, oltre alla lunghezza della sequenza e il riferimento all'array che la contiene, anche il valore dell'elemento da ricercare, precedentemente richiesto all'utente:

```
posizione = ricerca( n, ele, vet );
```

Il metodo ritorna un valore intero, che corrisponde alla posizione dove è stato reperito l'elemento. Considerazioni analoghe valgono per il metodo di ricerca binaria `ricBin`.

```
/* Gestione di una sequenza con l'uso di funzioni per
 * immissione, ordinamento, ricerca completa, ricerca
 * binaria e visualizzazione */
import java.util.Scanner;
class GestioneSequenza {
  public static void main (String argv[]) {
    int MAX_ELE = 1000;  /* numero di elementi dell'array */
    int vet[] = new int[MAX_ELE];    /* array che ospita
                                        la sequenza      */
    char invio;
    int scelta=-1, n=0, ele, posizione;
    Scanner sc = new Scanner(System.in);
    while(scelta != 0) {
      System.out.println ("\n\n\n   GESTIONE SEQUENZA");
      System.out.println ("1. Immissione");
      System.out.println ("2. Ordinamento");
      System.out.println ("3. Ricerca completa");
      System.out.println ("4. Ricerca binaria");
      System.out.println ("5. Visualizzazione");
      System.out.println ("0. Fine");
      System.out.print ("\nScegliere una opzione: ");
      scelta = sc.nextInt();
      switch (scelta) {
        case 1: n = immissione( MAX_ELE, vet );
                break;
        case 2: ordinamento( n, vet );
                break;
        case 3: System.out.print ("\n\nElemento da
                                    ricercare: ");
                  ele = sc.nextInt();
                posizione = ricerca( n, ele, vet );
```

```
                if(ele == vet[posizione]) {
                  System.out.print ("Elemento presente
                                        in posizione: ");
                  System.out.println (posizione);
                }
                else
                  System.out.println ("Elemento non
                                            presente");
                break;
        case 4: System.out.print ("\n\nElemento da
                                        ricercare: ");
                ele = sc.nextInt();
                posizione = ricBin( n, ele, vet );
                if(posizione != -1) {
                  System.out.print ("Elemento presente
                                        in posizione: ");
                  System.out.println (posizione);
                }
                else
                  System.out.println ("Elemento non
                                            presente");
                break;
        case 5: visualizzazione( n, vet );
                break;
      }
    }
}

/* Immissione elementi della sequenza */
static int immissione(int MAX_ELE, int vet[]) {
  int i, n;
  Scanner sc = new Scanner(System.in);
  do {
     System.out.print ("\n\nNumero elementi: ");
     n = sc.nextInt();
  }
  while (n < 1 || n > MAX_ELE);

  for(i = 0; i < n; i++) {
      System.out.print ("Immettere un intero: ");
      vet[i] = sc.nextInt();
  }
  return( n );
}
```

```
  /* Ordinamento ottimizzato */
  static void ordinamento(int n, int vet[]) {
    int i, p=n, k, n1, aux;
    do {
       k = 0;
       for(i = 0; i < n-1; i++)
        if(vet[i] > vet[i+1]) {
          aux = vet[i]; vet[i] = vet[i+1];
          vet[i+1] = aux;
          k = 1; p = i + 1;
       }
       n = p;
    }
    while (k == 1 && n>1);
  }

  /* Ricerca sequenziale */
  static int ricerca ( int n, int ele, int vet[] ) {
    int i; i = 0;
    while (ele != vet[i] && i < n-1) ++i;
    return( i );
  }

  /* ricerca binaria */
  static int ricBin( int n, int ele, int vet[] ) {
    int i, alto = 0, basso = n-1, pos = -1;
    do {
       i = (alto+basso)/2;
       if(vet[i] == ele) pos = i;
       else if(vet[i] < ele) alto = i + 1;
             else basso = i - 1;
    }
    while(alto <= basso && pos == -1);
    return( pos );
  }

  /* visualizza sequenza */
  static void visualizzazione( int n, int vet[] ) {
    int i;      char invio;
    System.out.println ("");
    for(i = 0; i < n; i++)
         System.out.println (vet[i]);
  }
}
```

Domande di verifica

1. Che cosa si intende per *metodo*? A cosa servono i metodi?
2. Che differenza c'è tra definizione e invocazione di un metodo?
3. Dove, come e quando si può invocare un metodo?
4. Cosa sono i parametri formali? E i parametri attuali? In che modo avviene il passaggio dei parametri?
5. Come è gestito il valore di ritorno di un metodo?
6. Predisporre un esempio in cui, richiamando un metodo che a sua volta chiama un altro metodo, si osservi la vita dei parametri e il passaggio dei valori.
7. Cosa si intende per *visibilità di un nome*?
8. Illustrare significato e uso della parola chiave `void`.
9. Presentare, in termini generali, un problema reale e la sua soluzione attraverso moduli correlati, seguendo la logica top-down. Successivamente far corrispondere ai moduli i metodi e per ognuno di essi definire i parametri d'ingresso e l'eventuale valore di ritorno.
10. Quando un metodo si dice ricorsivo?
11. Quali strutture di programmazione possono essere sostituite dai metodi ricorsivi?
12. Le successive chiamate ricorsive di un metodo condividono le stesse variabili locali?
13. Scrivere le formule che calcolano le permutazioni, le disposizioni e le combinazioni semplici.
14. Presentare degli esempi di funzioni matematiche definite in modo ricorsivo.
15. Una soluzione ricorsiva è in generale più efficiente in termini di velocità di esecuzione di una corrispondente soluzione iterativa? Perché?
16. A proposito di ricorsione e iterazione, che osservazioni si possono fare relativamente a eleganza e chiarezza della soluzione? Sono considerazioni univoche o devono essere rapportate al problema e, quindi, ai possibili algoritmi risolutivi?

Esercizi

1. Scrivere un metodo che dati in ingresso `b`, reale ed `e`, intero positivo, calcoli `b` elevato alla `e`.
2. Modificare il metodo del precedente esercizio così che calcoli anche potenze negative.

3. Esaminare i listati del libro e per ogni metodo si scriva la lista delle variabili locali.

4. Scrivere un metodo che calcoli, al variare della x, il valore dell'espressione

$$3x^3 - \sqrt{\frac{x^2+3}{2}}$$

Il metodo per il calcolo della radice quadrata è `Math.sqrt(double a)`.

5. Scrivere un programma che determini, a meno di un errore di 0.0001, uno zero della funzione matematica $f(x) = 2x^3 - 4x + 1$ nell'intervallo [0,1] utilizzando il metodo dicotomico valido quando $f(x)$ è una funzione continua che negli estremi dell'intervallo [a,b] assume valori di segno discorde, ovvero uno negativo e uno positivo e quindi tale che $f(a) \cdot f(b) < 0$. Sotto queste condizioni, sia $m = (a+b)/2$ il punto medio dell'intervallo, se $f(m) = 0$ abbiamo ottenuto lo zero cercato, altrimenti si considera l'intervallo $[a, m]$ se $f(a)$ e $f(m)$ hanno segno discorde o l'intervallo $[m, b]$ se $f(a)$ e $f(m)$ hanno segno concorde. Si itera quindi lo stesso procedimento nell'intervallo appena determinato e si prosegue fino a trovare uno zero di f.

6. Modificare l'algoritmo risolutivo del precedente esercizio per determinare lo zero di altre funzioni, per esempio delle funzioni

$\sqrt{x}$

$f(x) = 1 - x^3$ in $[0, 1]$

$g(x) = x - |1 - x|$ in $[0, 1]$

$h(x) = x - \cos(x)$ in $[0, \pi/2]$

7. Scrivere un metodo che visualizzi

```
PICOSOFT
Sistema per la gestione integrata

          OPZIONI DISPONIBILI

          1. Magazzino
          2. Clienti
          3. Fornitori
          4. Personale
          0. Fine

            Scegliere una opzione: _
```

e quindi restituisca al programma chiamante la scelta effettuata dall'utente.

8. Scrivere un metodo che visualizzi la scritta

```
Premere un invio per continuare
```

e interrompa quindi l'esecuzione del programma chiamante finché non viene premuto un tasto.

9. Scrivere un metodo ricorsivo che calcoli e visualizzi il fattoriale di tutti i numeri minori o uguali a n.

10. Scrivere un metodo ricorsivo che accetti in ingresso n valori e ne restituisca la somma al programma chiamante.

11. Scrivere un metodo ricorsivo che calcoli una potenza con esponente intero maggiore o uguale a zero.

12. Estendere la soluzione dell'esercizio precedente in modo che l'esponente possa essere un numero intero qualsiasi (anche negativo).

13. Scrivere un metodo ricorsivo che calcoli il massimo comun divisore di due numeri interi positivi ricordando che

$$\mathrm{MCD}(a,b) = \begin{cases} \mathrm{MCD}(a-b,b) & \text{se } a > b \\ a & \text{se } a = b \\ \mathrm{MCD}(b,a) & \text{se } a < b \end{cases}$$

14. Scrivere un metodo ricorsivo che calcoli il massimo comun divisore di due numeri interi positivi utilizzando l'algoritmo euclideo per cui:

$$\mathrm{MCD}(a,b) = \begin{cases} a & \text{se } b = 0 \\ \mathrm{MCD}(b,a) & \text{se } b > a \\ \mathrm{MCD}(b, a \bmod b) & \text{altrimenti} \end{cases}$$

15. Scrivere una versione ricorsiva della ricerca binaria su un vettore ordinato.

16. Scrivere un programma che calcoli i numeri ottenuti in base alla seguente definizione:

$a_1 = 3$

$a_2 = 7$

$a_n = 2 \cdot a_{n-1} - 3 \cdot a_{n-2}$ per $n \geq 3$

17. Modificare la funzione di immissione della sequenza esaminata nell'ultimo paragrafo del presente capitolo in modo che dopo aver effettuato l'inizializzazione dell'array esso venga ordinato.

18. Scrivere un programma che accetti in ingresso i valori di due array numerici, li ordini e ne ottenga la fusione in un terzo. L'array risultato della fusione contiene tutti i valori ordinati del primo e del secondo array. Utilizzare singoli metodi per l'inserimento dei valori, l'ordinamento e la fusione.

Capitolo 8

Verso la programmazione a oggetti

Obiettivi didattici

- Programmi comprensibili
- Problemi, moduli, funzioni
- Aggregazioni per modellare dati complessi
- Limiti dell'approccio tradizionale a moduli
- Maniglie per consentire l'accesso ai dati esclusivamente attraverso le funzioni appositamente predisposte

Questo capitolo è rivolto in modo particolare a coloro i quali conoscono già qualche linguaggio di programmazione non orientato agli oggetti e desiderano approfondire le ragioni per cui la comunità dei programmatori ha pian piano sposato il paradigma degli oggetti. Se gli argomenti qui trattati dovessero sembrare non abbastanza stimolanti, è possibile andare direttamente al capitolo successivo nel quale saranno esposti i concetti fondamentali della programmazione a oggetti senza far riferimento all'evoluzione storica.

Quando apparve, la programmazione a oggetti suscitò molto clamore e accese discussioni contrastanti sulla sua validità e utilità. In molti casi fu presentata e percepita come un sistema di programmazione completamente rivoluzionario, per il quale era necessario rinunciare a tutte le conoscenze di programmazione acquisite e ricominciare da zero. In realtà le cose non stanno proprio così e la programmazione a oggetti rappresenta il punto di arrivo di un percorso, probabilmente non ancora terminato, di miglioramento dei linguaggi di programmazione iniziato con lo sviluppo del primo macro-assembler. Ripercorreremo idealmente parte di questa evoluzione per illustrare da dove provenga l'esigenza della programmazione a oggetti, in modo da focalizzare le problematiche che i programmatori hanno incontrato negli anni e le risposte che fornisce l'approccio *Object Oriented*.

8.1 Il linguaggio ideale

Nel corso degli anni i progettisti di linguaggi di alto livello si sono posti diversi obiettivi da raggiungere, tra questi ne esiste uno che sta alla base di tutto e senza il quale gli altri perdono di significato. L'obiettivo primario di qualsiasi linguaggio di programmazione di alto livello è mettere in grado il programmatore di sviluppare programmi *comprensibili* agli esseri umani. Meno un programma è comprensibile, più difficile diventa non solo la manutenzione, ma anche lo sviluppo e l'individuazione degli errori. Fin dagli albori dell'era informatica, si è sognato un computer in grado di comprendere il linguaggio umano, un computer al quale fosse sufficiente una spiegazione come a un qualsiasi collega affinché potesse fare i suoi conti e dare la risposta attesa. Nel tentativo di raggiungere questo scopo, i progettisti del COBOL permisero che nel proprio linguaggio fossero incluse istruzioni quanto più simili alle frasi del linguaggio corrente: per esempio, in Java per controllare se la variabile v contiene un valore maggiore o uguale a 0 si scrive:

```
if (v >= 0)
```

In COBOL si può invece usare una notazione come la seguente:

```
if v is greater or equal to zero
```

Il risultato ottenuto però, alla luce dell'esperienza, non è stato quello di avere programmi più comprensibili, ma solo programmi più verbosi.

Riflettendoci ci si rende conto che anche spiegare un problema a un collega non è poi una cosa così semplice, specialmente se questi è inesperto. A conforto di ciò basta pensare che quando viene assunto un neolaureato in un'azienda, possono passare mesi prima che diventi produttivo, nonostante i venti o più anni complessivi di studi alle spalle. È evidente quindi la difficoltà nell'esporre problemi e metodi di risoluzione che esiste anche fra esseri umani e non solo tra essere umano e macchina. Per questo motivo prima di sviluppare un linguaggio adatto per descrivere un problema a una macchina, è necessario individuare una metodologia che sia adatta a descrivere i problemi in generale. Un risultato definitivo in tal senso non si è ancora ottenuto, ma alcuni principi generali sono stati individuati.

8.2 Divide et impera

Come è noto la mente umana ha una capacità limitata di manipolare molti dati contemporaneamente; è sufficiente una semplice moltiplicazione a memoria di due numeri di quattro cifre per mettere in crisi la maggior parte delle persone. L'unico modo che abbiamo per trattare un problema complesso è di suddividerlo in problemi più piccoli e risolverli separatamente; è quanto facciamo normalmente per trovare il risultato della moltiplicazione precedente: si suddivide in una serie di moltiplicazioni di una cifra per una cifra e si esegue qualche somma. Il principio è applicabile e applicato dagli esseri umani alla risoluzione di qualsiasi problema ed è quindi il metodo adatto per scrivere programmi comprensibili. La programmazione strutturata si basa su questo principio e i linguaggi strutturati forniscono alcuni strumenti per metterlo in pratica.

L'idea di base consiste nel suddividere in cascata un problema in problemi di volta in volta più semplici. Una volta giunti a un livello sufficientemente elementare si implementano i vari moduli applicativi che risolvono i singoli sottoproblemi. Detto in questi termini sembra un processo piuttosto lineare, ma c'è una complicazione: i moduli debbono necessariamente interagire tra loro per cui, quando si ha a che fare con molti moduli, si genera una rete di interazioni che è parte integrante del problema e che introduce una complessità difficile da gestire. È necessario quindi che i moduli siano quanto più possibile indipendenti l'uno dall'altro e che le inevitabili interazioni siano governate da protocolli di comunicazione chiari e semplici da comprendere. Riprendendo l'esempio precedente, la moltiplicazione fra due numeri viene suddivisa in moltiplicazioni più semplici che però interagiscono l'una con l'altra tramite somme.

Lo strumento più potente disponibile nei linguaggi strutturati atto allo scopo è la *funzione*. Una funzione consiste di un insieme di istruzioni, caratterizzate da un nome, che possono venire attivate da un punto qualsiasi del programma per quante volte si vuole. Nella definizione più restrittiva, una funzione viene attivata sempre con uno o più parametri e, in base solamente a questi, fornisce un risultato univoco. Per calcolare il risultato, una funzione può utilizzare dei dati interni che sono completamente invisibili al resto del programma. Questo tipo di comportamento, che riproduce quello delle funzioni matematiche, è stato poi esteso per motivi pratici, per cui le funzioni usate nei linguaggi di programmazione possono essere attivate senza parametri, possono non fornire alcun risultato oppure il risultato fornito in momenti diversi può essere differente anche con parametri uguali. In questi casi l'analogia col modello matematico è andata perduta.

In C un esempio di chiamata di funzione è

```
a = cos(b);
```

Il significato della riga precedente dovrebbe essere abbastanza chiaro: assegno alla variabile `a` il valore del coseno della variabile `b`. In questo semplice esempio le interazioni sono veramente ridotte al minimo, si fornisce un parametro in ingresso e si ottiene un risultato; cosa accade nella funzione che calcola il coseno non ci interessa e in ogni caso non aggiungerebbe informazioni per la comprensione del programma. Purtroppo non tutti i problemi sono risolubili in questo modo poiché spesso è necessario trattare dati complessi, cioè insiemi di più dati semplici che hanno delle dipendenze uno dall'altro.

Supponiamo che in un programma sia necessario manipolare dei tempi, dei dati cioè che contengono ore, minuti e secondi più un simbolo utilizzato come separatore nella visualizzazione. I linguaggi strutturati permettono di aggregare più dati insieme; lo strumento che fornisce il C si chiama *struttura*, `struct`, e consente di rappresentare il nostro dato tempo nel modo seguente.

```
struct Tempo {
int ore;
     int minuti;
     int secondi;
     char separatore;
   } mioTempo;
```

La definizione precedente descrive un tipo di dato composto da tre interi e un carattere; la parola chiave `int` indica, infatti, che il nome che segue corrisponde a una variabile di tipo intero mentre la parola chiave `char` specifica una variabile di un byte. È importante notare subito che `struct Tempo` indica una forma, un modello di dati e semanticamente ha lo stesso scopo della parola `int`; `mioTempo` è invece il nome della variabile e a esso corrisponde un'area di memoria allocata. Per riferirsi a un elemento interno si usa l'operatore punto preceduto dal nome della variabile e seguito da quello dell'elemento. Le seguenti istruzioni assegnano valori agli elementi di `mioTempo`.

```
mioTempo.ore        = 12;
mioTempo.minuti     = 30;
mioTempo.secondi    = 25;
mioTempo.separatore = ':';
```

Nel trattare con dati strutturati si deve porre molta attenzione se non si vuol correre il rischio di ottenere degli stati inconsistenti; un buon sistema è quello di creare un insieme definito di funzioni che trattino il nostro dato aggregato in modo analogo alla funzione coseno vista precedentemente.

```
struct Tempo aggiungiOre (struct Tempo mioTempo, int numOre)
struct Tempo togliOre (struct Tempo mioTempo, int numOre)
...
```

In particolare una funzione dell'insieme potrebbe essere

```
struct Tempo aggiungiOre (struct Tempo mioTempo, int numOre)
{
   mioTempo.ore = mioTempo.ore + numOre;
   while (mioTempo.ore > 23)
         mioTempo.ore = mioTempo.ore - 24;
   return mioTempo;
}
```

Il comportamento della funzione `aggiungiOre` è il seguente: si aggiunge il parametro di ingresso `numOre` all'elemento della struttura `mioTempo.ore`, dopodiché si esegue un ciclo fintanto che `mioTempo.ore` contiene un valore superiore a 23, sottraendo a tale elemento il valore 24 a ogni iterazione. Se `mioTempo.ore` contiene un valore inferiore o uguale a 23 il ciclo stesso non viene eseguito. L'istruzione `return` restituisce il valore modificato di `mioTempo` all'istruzione chiamante. Il dato restituito da `return` deve essere dello stesso tipo dichiarato prima del nome della funzione stessa, in questo caso `struct Tempo`.

✔ NOTA

Questa funzione si comporta correttamente solo nel caso in cui `numOre` *sia positivo. In questa fase non vale la pena complicare gli algoritmi, gli esempi sono rivolti solo a comprendere il senso dell'esposizione.*

L'approccio è sicuramente valido, ma dimostra almeno tre punti deboli.

1. La struttura `Tempo` viene trattata dalle funzioni, ma è visibile e modificabile anche fuori da esse: ciò implica che pur avendo delle funzioni corrette per trattare la struttura, non abbiamo nessuna garanzia dal linguaggio che i dati in essa contenuti siano sempre consistenti. Da tale visibilità deriva inoltre un altro problema. Nel caso in cui sia necessario modificare la struttura `Tempo` per motivi di performance o perché non corrisponde più alle nuove esigenze, non è sufficiente modificare le funzioni che la trattano, ma va controllato l'intero programma per adattare tutti i punti in cui viene trattata direttamente. L'ideale sarebbe poter definire a livello di linguaggio che i dati di una struttura siano manipolabili solo da determinate funzioni, così avremmo la certezza che una volta che queste funzioni sono corrette, lo stato è sicuramente consistente; nel caso in cui diventi necessario modificare il formato della struttura, sarà sufficiente adattare solo le funzioni dedicate al suo trattamento e lasciare inalterato il resto del programma.
2. Abbiamo definito un insieme di funzioni che sono accomunate da uno stesso scopo, cioè il trattamento di una stessa struttura. Questo fatto però è noto solo a chi ha partecipato allo sviluppo, nella migliore delle ipotesi è riportato nella documentazione e non è desumibile direttamente dal programma se non dopo un'analisi accurata. Avere questa informazione desumibile immediatamente dal programma ne renderebbe sicuramente più semplice la comprensione.
3. Se nello stesso programma dovessimo trattare anche il tipo di struttura `Tempo2` contenente i centesimi di secondo oltre le ore, i minuti e i secondi, dovremmo definire un nuovo insieme distinto di funzioni, anche nel caso in cui si richieda lo stesso risultato. Dovremmo, per esempio, definire una funzione `aggiungiOre(struct Tempo mioTempo, int ore)` e una `aggiungiOre2(struct Tempo2 mioTempo2, int ore)` anche se le due funzioni eseguono i medesimi calcoli.

8.3 Programmazione a *maniglie*

La visibilità dei dati di cui abbiamo parlato precedentemente diventa assolutamente inaccettabile nel caso in cui si utilizzino delle funzioni atte a trattare i dati complessi come quelli che fanno parte del sistema operativo (risorse), per esempio quando si deve gestire un file. Le cause di questa inaccettabilità sono principalmente tre – ci riferiamo ai sistemi operativi come esempio particolarmente significativo oltre che importante nell'evoluzione storica dei linguaggi ma il ragionamento è valido in generale.

1. I sistemi operativi sono soggetti a continue evoluzioni per cui le strutture interne possono cambiare anche fra un rilascio e il successivo. Con i dati interni visibili, sarebbe necessario perlomeno ricompilare tutti i programmi a ogni cambiamento anche minimo delle strutture interne. Oltre a questo è opportuno che operazioni analoghe (come la gestione di un file) siano effettuate nello stesso modo anche su sistemi operativi differenti, mettendo così in grado i programmatori di realizzare programmi funzionanti su più piattaforme.

2. Una modifica delle strutture dati interne può mettere a repentaglio non solo il buon funzionamento del nostro programma ma anche del sistema operativo stesso, col rischio di fermare il lavoro di più persone e di perdita di dati.
3. La visibilità delle strutture interne rende inutile qualsiasi tentativo di gestire la riservatezza dei dati e ciò non è concepibile specialmente in un sistema operativo multiutente.

I problemi esposti sono così gravi e antichi che i progettisti dei sistemi operativi hanno da tempo trovato una soluzione: il trattamento delle risorse tramite *maniglie* (*handle*). Questo sistema è così valido che viene ancora abbondantemente usato anche nei sistemi operativi più moderni e rappresenta una tappa evolutiva tanto importante da indurci, invece che a una descrizione generica del funzionamento, a vederne un esempio pratico quale l'utilizzo di tale meccanismo per il trattamento del tempo. A tale scopo, facciamo riferimento ancora al C che è il linguaggio più usato per questo tipo di programmazione.

Un programma C è normalmente formato da più sorgenti che possono venire compilati separatamente e infine legati insieme (fase di *link*) in modo da ottenere il programma funzionante. In C è possibile definire delle variabili globali che possono essere manipolate da tutti i moduli sorgenti oppure delle variabili, dette globali statiche, che possono essere manipolate solo all'interno di un singolo modulo sorgente. Ogni variabile globale, statica o meno, può essere inizializzata; è possibile cioè assegnarle un valore che conterrà dall'inizio dell'elaborazione fino a che il programma non lo modificherà.

In C, così come in Java, è possibile utilizzare gli array che consistono in un insieme di variabili dello stesso tipo distinguibili l'una dall'altra per mezzo di un indice che può assumere un valore compreso tra 0 e $n - 1$, dove n è il numero degli elementi che lo compongono. Nella dichiarazione di un vettore dobbiamo specificarne prima il tipo, poi il nome e successivamente, tra parentesi quadre, il valore di n. Infatti in C, a differenza di Java, la dichiarazione corrisponde alla definizione, per cui se vogliamo ottenere un array di tre numeri interi è sufficiente scrivere:

```
int mioArray[3];
```

I tre elementi costituenti l'array sono individuabili con le seguenti notazioni:

```
mioArray[0]
mioArray[1]
mioArray[2]
```

Anche i vettori possono essere inizializzati. I singoli elementi dell'array possono essere oltre che di tipo semplice, nell'esempio precedente `int`, anche di tipo composto come le strutture. La dichiarazione

```
struct Tempo ArrayTempo[DIM_ARRAY_TEMPO];
```

specifica che il vettore `ArrayTempo` ha `DIM_ARRAY_TEMPO` elementi costituiti ciascuno da una `struct` di tipo `Tempo` che abbiamo già visto precedentemente.

Con questi strumenti a disposizione ci proponiamo di sviluppare un modulo sorgente tale che permetta al resto del programma di poter trattare con i tempi in modo semplice, senza però avere la possibilità di inserire neppure per sbaglio dei tempi incongruenti. In questo modulo dichiariamo un array globale statico `ArrayTempo` di strutture `Tempo` esattamente come sopra. Inizializziamo il dato `ore` di ciascun elemento dell'array a 99. Come detto gli elementi di tale array non saranno visibili in nessun altro modulo sorgente. Dichiariamo poi la funzione:

```
int newTempo ()
{
   int i;

   for (i = 0; i < DIM_ARRAY_TEMPO; i++)
      if (ArrayTempo[i].ore == 99) {
         ArrayTempo[i].ore = 0;
         ArrayTempo[i].minuti = 0;
         ArrayTempo[i].secondi = 0;
         return i;
      }
   return -1;
}
```

Lo scopo di `newTempo` è creare una corrispondenza tra un dato strutturato di tipo `Tempo` e un numero. Tale numero è detto appunto maniglia (handle) in quanto consente di *tenere* un insieme di dati non visibile in altra maniera. In questo caso la nostra handle è l'indice di un array, ma in generale può rappresentare cose diverse che riguardano però solo il lato implementativo della gestione dei dati interni e non il resto del programma.

A questo punto quindi, ogni volta che dobbiamo trattare con dei tempi, non dichiariamo più una struttura come visto in precedenza, bensì richiediamo una handle tramite la funzione `newTempo`. Affinché una handle possa essere utile a qualche cosa necessitano però altre funzioni come le seguenti:

```
int assegnaTempo (int handle, int ora, int minuti, int secondi)
{
   if (ora >= 0 && ora < 24 &&
       minuto >= 0 && minuto < 60 &&
       secondo >= 0 && secondo < 60) {
      ArrayTempo[handle].ore = ora;
      ArrayTempo[handle].minuti = minuto;
      ArrayTempo[handle].secondi = secondo;
      return 0;
   } else
      return -1;
}
```

```
int leggiOra (int handle)
{
   return ArrayTempo[handle].ore;
}

int leggiMinuti (int handle)
{
   return ArrayTempo[handle].minuti;
}

int leggiSecondi (int handle)
{
   return ArrayTempo[handle].secondi;
}
```

La funzione che abbiamo già visto per aggiungere le ore diverrebbe quindi:

```
void aggiungiOre (int handle, int numOre)
{
  ArrayTempo[handle].ore = ArrayTempo[handle].ore + numOre;
  while (ArrayTempo[handle].ore > 23)
       ArrayTempo[handle].ore = ArrayTempo[handle].ore - 24;
}
```

✔ **NOTA**

Anche in C la parola chiave `void` *indica che la funzione non restituisce nessun valore all'istruzione chiamante.*

Come si vede, se le funzioni descritte vengono implementate correttamente, diventa impossibile ritrovarsi con dei tempi incongruenti. Inoltre, se un giorno decidessimo che è più conveniente memorizzare un tempo come il numero di secondi a partire dalla mezzanotte invece che utilizzare la struttura `Tempo`, sarebbe sufficiente modificare solo un modulo sorgente mentre il resto del programma rimarrebbe invariato. Questo modo di programmare sicuramente interessante induce alcune considerazioni:

1. Il sistema delle handle nasconde completamente i dati che vengono trattati, ma non risolve gli altri problemi illustrati precedentemente. In certi casi il fatto di nascondere completamente la struttura dati comporta un sovraccarico di elaborazione, senza nessun vantaggio pratico dal punto di vista della congruenza della struttura dati stessa. Nel nostro caso, per esempio, sarebbe necessario definire due funzioni per assegnare e leggere il dato `separatore` anche se il contenuto di questo dato non può in nessun caso inficiare la congruenza degli altri dati. Il poter trattare `separatore` in modo diretto renderebbe quindi più efficiente e forse anche più leggibile il programma.

2. Tutta la gestione del *vettore invisibile* è affidata al programmatore. Ogni volta che sia necessario trattare una nuova struttura, il programmatore deve implementare di nuovo tutta la gestione del vettore. In questi esempi poi abbiamo utilizzato il sistema del vettore per chiarezza di esposizione, ma in un ambiente reale tale scelta non è molto conveniente in quanto richiede una dimensione fissa che in certi casi potrebbe rivelarsi eccessivamente sovrastimata mentre in altri potrebbe risultare insufficiente. Questo problema potrebbe essere risolto mediante l'uso di strutture più adatte, come, per esempio, delle liste concatenate che però renderebbero ancora più gravoso il compito del programmatore. Disporre di un linguaggio che gestisca in modo automatico questo tipo di problemi rappresenterebbe sicuramente un grosso vantaggio.
3. Le maniglie sono numeri interi, non sono legate in alcun modo con il dato cui corrispondono. Ciò significa che possono essere alterate dal programma perdendo in tal modo l'aggancio con i dati cui inizialmente erano legate. Nel caso in cui si utilizzi il sistema delle maniglie per trattare per esempio le date oltre che i tempi, non esiste nessun controllo possibile da parte del compilatore perché non vengano usate le handle dei tempi nelle funzioni che trattano le date e viceversa.
4. Le funzioni illustrate hanno perso l'analogia con le funzioni matematiche così evidente nel caso del calcolo del coseno. Fornendo, infatti, un medesimo valore in ingresso a una funzione matematica, essa fornisce sempre lo stesso risultato mentre fra quelle illustrate ne esiste perfino una che non restituisce alcunché. Nel modo in cui noi le abbiamo utilizzate esse, più che funzioni, sono *metodi* di trattamento di un dato.

Il sistema delle maniglie dunque è sicuramente valido ma necessita di una sovrastruttura che è a carico del programmatore e un utilizzo delle funzioni che è stravolto rispetto alla definizione originale. Deve allora esistere un paradigma che permetta di utilizzare questo approccio in modo più semplice e dove le funzioni diventano metodi. Questo nuovo paradigma è quello della programmazione orientata agli oggetti.

Domande di verifica

1. Quale caratteristiche dovrebbero avere le istruzioni di un linguaggio di programmazione per facilitare il lavoro del progettista software?
2. Quali sono i vantaggi della programmazione modulare?
3. Per quale ragione sono stati introdotti gli aggregati di dati come le strutture del linguaggio C?
4. Perché è così importante il concetto di *visibilità*?
5. Quali sono i limiti dell'approccio funzionale?
6. Quali problemi risolvono le "maniglie" e quali lasciano ancora aperti?

Capitolo 9

Programmazione a oggetti in Java

Obiettivi didattici

- Classi
- Esemplari o istanze di classi: oggetti
- Dati e metodi pubblici/privati
- Identificatore di un oggetto
- Incapsulamento
- Ereditarietà
- Polimorfismo
- Overriding
- Binding dinamico
- Operatori sulle istanze
- Costruttori e finalizzazione
- `this` e `super`
- Variabili e metodi di classe
- Classi astratte
- UML
- Pratica di programmazione a oggetti

Nel capitolo precedente abbiamo visto alcuni problemi che si possono presentare sviluppando programmi con linguaggi strutturati come il C e come essi sono stati affrontati. Ciò che si voleva sottolineare con questi esempi è che i linguaggi strutturati

costringono spesso i programmatori a sforzi che potrebbero essere evitati con l'impiego di strumenti più adatti. Adesso vogliamo dimostrare come un linguaggio a oggetti come Java, diretta evoluzione dei linguaggi strutturati, sia in possesso degli strumenti adatti per risolvere i problemi illustrati.

9.1 Un mondo di oggetti

La caratteristica che rende così utili e interessanti i computer è la loro capacità di simulare realtà virtuali più o meno aderenti alla realtà che noi sperimentiamo tutti i giorni; questo è evidente usando un videogioco che conduce il giocatore in mondi immaginari, ma rimane vero anche usando un programma di videoscrittura, che simula un foglio di carta e una macchina per scrivere, e perfino usando un normale programma di contabilità, che simula un mondo popolato da clienti, fornitori e banche.

Queste simulazioni sono utili, e a volte divertenti, perché estendono le simulazioni che siamo in grado di fare con il nostro cervello: siamo in grado di immaginare un mondo fantastico ma il computer ci permette di vederlo, siamo in grado di pensare a una lettera da scrivere ma non di stamparla, siamo in grado di tenere a mente la nostra situazione contabile a patto che non sia troppo complessa.

Dunque chi scrive programmi per computer deve tipicamente analizzare una determinata realtà virtuale così come è percepita dal nostro cervello, estrapolarne le regole e trasferirle in un computer cercando di rendere evidenti quanto più possibile i legami tra il programma e la realtà virtuale simulata. Il modo più semplice e naturale che abbiamo per descrivere una qualsiasi realtà virtuale è quello di descriverne le parti e le relazioni tra esse.

Il principio base della programmazione a oggetti è proprio questo, permettere di rappresentare ogni parte di una realtà virtuale tramite un costrutto che lo rappresenta logicamente: ci si riferisce a questi costrutti con il termine *oggetti* e non è un caso che il primo linguaggio in cui presero forma questi concetti si chiami "Simula".

In un programma generico esisteranno diversi tipi di oggetto e, per ogni tipo, diversi oggetti. Ciò che è importante è la *descrizione del tipo di oggetto*: la *classe*, che è il nucleo fondamentale di ogni applicazione Java. Il termine classe in questo contesto va interpretato come "gruppo di cose o individui caratterizzati da medesime qualità e/o comportamenti" (mentre in generale ha anche altri significati). I bambini appartenenti a una stessa classe scolastica hanno in comune solo il comportamento di seguire il medesimo corso di studi, mentre gli oggetti appartenenti alla classe degli elettrodomestici hanno in comune il fatto di funzionare a elettricità, che è una qualità, e di essere utili nelle faccende domestiche, che è un comportamento. Descrivere una classe significa descrivere in modo astratto qualità e comportamenti di un gruppo di oggetti o individui senza fare riferimento all'oggetto o all'individuo singolo. Una classe a volte può essere suddivisa in sottoclassi: per esempio la classe degli elettrodomestici può essere suddivisa in lavatrici, lavapiatti, frullatori, aspirapolvere ecc.; la classe degli aspirapolvere a sua volta può essere suddivisa in scope elettriche e aspirapolvere con ruote e così via. L'aspirapolvere che abbiamo in casa è un oggetto, vale a dire un *esemplare della classe* aspirapolvere o, come si usa dire nella terminologia della pro-

grammazione a oggetti, *un'istanza della classe* aspirapolvere. Il concetto di classe è molto antico e il fatto che i linguaggi di programmazione l'abbiano adottato mostra come questi si siano adattati al nostro modo di pensare.

9.2 Classi

Supponiamo di dover trattare dei tempi, dei dati cioè che contengono ore, minuti e secondi più un simbolo utilizzato come separatore nella visualizzazione. In Java il modo più naturale per fare questo è dichiarare una classe che incorpori tutte le caratteristiche che ci interessano. Un esempio potrebbe essere il seguente:

```
class Tempo {
   private int ore;
   private int minuti;
   private int secondi;
   public char separatore;

   public int assegnaTempo (int ora, int minuto,
                                        int secondo) {
      if (ora >= 0 && ora < 24 &&
          minuto >= 0 && minuto < 60 &&
          secondo >= 0 && secondo < 60) {
         ore = ora;
         minuti = minuto;
         secondi = secondo;
         return 0;
      } else
         return -1;
   }

   public int leggiOra () {
      return ore;
   }

   public int leggiMinuti () {
      return minuti;
   }

   public int leggiSecondi () {
      return secondi;
   }

   public void aggiungiOre (int numOre) {
      ore = ore + numOre;
      while (ore > 23)
         ore = ore - 24;
   }
```

```
    public void visualizza (boolean ritornoACapo) {
        System.out.print (ore);
        System.out.print (separatore);
        System.out.print (minuti);
        System.out.print (separatore);
        if (ritornoACapo)
            System.out.println (secondi);
        else
            System.out.print (secondi);
    }
}
```

In questa classe sono dichiarate alcune variabili (`ore`, `minuti`, `secondi` e `separatore`) e alcuni metodi che vengono invocati per modificarle. Notiamo infatti che prima di ogni dichiarazione di dati o metodi è presente la parola chiave `public` o la parola chiave `private`, che consentono di stabilire se i dati o i metodi che seguono possono essere visibili anche a metodi non dichiarati nella classe (`public`) oppure se sono visibili solo all'interno dei metodi dichiarati nella classe (`private`). Nella classe `Tempo` tutte le variabili, eccettuato `separatore`, sono inaccessibili dal di fuori della classe e possono essere modificate solo tramite i metodi della classe `Tempo`. Ci si riferisce a variabili e metodi di una classe col termine *membri della classe*.

Una classe rappresenta una descrizione di qualità, consistente nei dati dichiarati, e comportamenti, consistente nei metodi. Per avere dei dati reali su cui poter lavorare è necessario creare un *oggetto* corrispondente alle caratteristiche dichiarate nella classe, o come si dice, creare un'istanza della classe o anche istanziare una classe. L'oggetto quindi è un insieme di dati reali, allocati nella memoria del calcolatore, corrispondenti a quelli dichiarati nella classe e manipolabili per mezzo dei suoi metodi. La creazione di un oggetto avviene tramite l'operatore `new`.

Per istanziare la nostra classe sono necessarie due istruzioni, la prima

```
Tempo mioTempo;
```

dichiara una variabile *identificatore di oggetto* (object-id) di nome `mioTempo` che può essere utilizzata solo per trattare oggetti della classe `Tempo`; al momento della dichiarazione, non esiste ancora nessuna istanza della classe `Tempo`: la nuova istanza della classe viene creata con l'istruzione

```
mioTempo = new Tempo();
```

Il metodo `new` crea un oggetto secondo le caratteristiche descritte nella classe `Tempo`, affida questo oggetto a un gestore di oggetti e restituisce l'object-id; questo viene memorizzato in `mioTempo` tramite l'operatore di assegnazione =. Come viene gestito l'oggetto da Java è un argomento che non ci deve interessare; l'unica cosa da sapere è che quando un oggetto non è più referenziato tramite alcun object-id, esso viene automaticamente eliminato dal sistema. Nel seguente esempio

```
Tempo mioTempo1, mioTempo2;
mioTempo1 = new Tempo();
mioTempo2 = mioTempo1;
```

abbiamo due object-id, ma un solo oggetto allocato in memoria cui entrambi gli object-id si riferiscono, per cui operare su uno o sull'altro alternativamente produce i medesimi effetti che operare sempre sullo stesso.

Riassumendo, abbiamo introdotto i seguenti tre concetti:

- *classe*: descrizione di un gruppo di oggetti tramite dati e metodi di accesso ad essi;
- *istanza della classe o oggetto*: un area di memoria del calcolatore, invisibile direttamente, che è conforme alle caratteristiche della classe da cui è istanziata;
- *object-id*: variabile che consente di identificare un oggetto nella memoria del calcolatore.

Abbiamo già visto le referenze a oggetti quando abbiamo parlato degli array e in effetti in Java essi sono oggetti speciali, come vedremo meglio in seguito.

Una volta che abbiamo un'istanza, è possibile accedere ai suoi membri pubblici tramite l'operatore punto (.) che connette un object-id a un membro della classe. Nel nostro esempio:

```
mioTempo.separatore = ':';
```

Questo assegnamento è possibile solo perché `separatore` è stato dichiarato `public`, in caso contrario il compilatore avrebbe segnalato errore. Per l'invocazione di un metodo si usa una notazione analoga; per esempio

```
mioTempo.assegnaTempo (15, 22, 44);
```

Anche in questo caso il compilatore verifica che il metodo invocato sia visibile; avendolo noi dichiarato `public` questa verifica viene superata. Quando l'interprete deve eseguire questa istruzione cerca nella classe di appartenenza di `mioTempo` il metodo `assegnaTempo`, dopodiché ne attiva l'esecuzione passandogli i parametri indicati più un parametro implicito che è l'object-id dell'oggetto. Questo parametro implicito è sempre passato ed è sempre visibile nel metodo tramite il nome convenzionale `this`. Poiché i metodi hanno come scopo quello di trattare i dati di un oggetto, l'uso di `this` è sottinteso per tutte le variabili dichiarate nella classe e, infatti, nella stesura dei metodi della nostra classe, esso non compare mai. È comunque possibile scrivere il metodo `aggiungiOre` nel modo seguente:

```
public void aggiungiOre (int numOre) {
   this.ore = this.ore + numOre;
   while (this.ore > 23)
      this.ore = this.ore - 24;
}
```

I metodi di una classe hanno, ovviamente, la completa visibilità sui dati della classe indipendentemente dal fatto che siano dichiarati `public` o `private`. Come si vede le variabili dichiarate nella classe esistono in realtà solo in un'istanza; per questo motivo vengono chiamate *variabili d'istanza*, mentre i metodi che possono trattarle vengono detti *metodi d'istanza*.

In Java ogni classe che sia visibile da qualsiasi classe, o come si dice ogni classe pubblica, deve essere dichiarata in un file sorgente il cui nome sia identico a quello della classe, facendo attenzione anche a maiuscole e minuscole, con il suffisso `.java`. Nel nostro caso dovremo quindi scrivere il codice della classe in un file sorgente il cui nome sia `Tempo.java`. Nello stesso file sorgente è possibile dichiarare altre classi che però non saranno visibili al di fuori del file stesso. Il compilatore genera un modulo con lo stesso nome del file sorgente, ma con il suffisso `.class`.

Per vedere come funzionano gli esempi riportati, possiamo creare una classe in modo analogo a come abbiamo già fatto nei primi capitoli del libro e dichiarare, nel metodo `main` o come variabile della classe, un object-id di `Tempo`. Il compilatore Java, `javac`, verifica se esistono delle dipendenze fra le classi e, in caso affermativo, effettua automaticamente le compilazioni necessarie quando non esiste il modulo relativo `.class` oppure se tale modulo risulta essere stato compilato precedentemente all'ultima modifica del modulo sorgente. Per fare un esempio pratico potremmo scrivere il seguente modulo sorgente di nome `ProvaTempo1.java`:

```
class ProvaTempo1{
   public static void main (String argv[]) {
      Tempo mioTempo;

      mioTempo = new Tempo();
      mioTempo.separatore = ':';
      mioTempo.assegnaTempo(22,30,5);
      mioTempo.visualizza(true);
      mioTempo.aggiungiOre (10);
      mioTempo.visualizza(true);
   }
}
```

Per compilare la classe è sufficiente l'istruzione

```
javac ProvaTempo1.java
```

Si può quindi eseguire il programma con l'istruzione

```
java ProvaTempo1
```

ottenendo il seguente risultato:

```
22:30:5
8:30:5
```

✔ NOTA

Il meccanismo di compilazione automatica di Java funziona correttamente la maggior parte delle volte, quando si modificano però delle classi in modo da cambiare la struttura di moduli già compilati, tale meccanismo non risulta più adeguato e si possono ottenere degli errori inattesi in fase di esecuzione. In questi casi la cosa più semplice ed efficiente che consigliamo è di cancellare tutti i moduli `.class` *e ricompilare la classe principale.*

A questo punto dovrebbe essere chiaro che con i linguaggi a oggetti si ottiene in modo conciso e leggibile ciò che con un linguaggio strutturato è complesso da leggere e richiede una quantità di codice aggiuntivo. Ciò è dovuto al fatto che il modello delle funzioni non è adatto a rendere invisibili i dati; si può forzare il modello usando al limite gli strumenti che mette a disposizione, ma il risultato in ogni caso risente della forzatura. Abbiamo appena varcato la soglia della programmazione a oggetti e abbiamo visto i primi vantaggi che offre, ma questo è solo il primo passo in un mondo che forse in parte è ancora inesplorato.

9.3 Incapsulamento

La possibilità di rendere invisibili i dati e di poterli trattare solo tramite metodi è detta *incapsulamento*: infatti, è come se i dati fossero racchiusi in una capsula con dei bottoni e manopole all'esterno che ne permettono la manipolazione. È l'approccio che viene usato da sempre per nascondere la complessità degli oggetti che usiamo tutti i giorni. Pensiamo, per esempio, a un'automobile: essa può essere guidata tramite poche leve e pedali da una persona che non conosce il funzionamento dei suoi componenti. Ciò permette, tra l'altro, di poter guidare indifferentemente auto di marche diverse e con motori diversi. Leve e pedali sono rappresentati in Java dai metodi.

È vero però che un'auto ha un compito ben preciso e che le sue caratteristiche standard sono state definite nel corso di anni. Un programma software invece nasce solitamente da specifiche generiche e nel corso del suo sviluppo e della sua vita attiva necessita di evoluzioni e adattamenti che lo rendano conforme a nuove esigenze. Abbiamo visto che l'incapsulamento semplifica la modifica dei programmi, ma non offre la flessibilità richiesta dall'evoluzione moderna. Per migliorare la flessibilità degli oggetti è stato introdotto allora il concetto di *ereditarietà*.

9.4 Ereditarietà

Il concetto generale di classe prevede la possibilità che questa sia suddivisibile in sottoclassi, di volta in volta più specifiche. Si può dire quindi che una sottoclasse *eredita* le caratteristiche della classe di cui fa parte e, *in più*, ne possiede altre che la rendono più adatta a un determinato scopo. Nell'esempio degli elettrodomestici fatto in precedenza, l'aspirapolvere ha tutte le caratteristiche di un elettrodomestico e, in più, ha la caratteristica di poter aspirare la polvere.

Per vedere in pratica come questo concetto sia tradotto in Java, riprendiamo la nostra classe `Tempo` e supponiamo di aver bisogno in certi casi, ma solo in certi casi, di tempi che contengono anche i centesimi di secondo. Possiamo allora creare una nuova classe, chiamiamola `Tempo2`, che possiede i dati e i metodi di `Tempo` con in più una variabile intera per contenere i centesimi di secondo e i metodi adatti a trattarla. La classe ottenuta in questo modo è detta *classe derivata* o *sottoclasse*, mentre la classe di partenza è detta *classe genitrice* o *superclasse*; in Java, la classe `Tempo2`, derivata da `Tempo`, potrebbe essere dichiarata come segue.

```
class Tempo2 extends Tempo {
   private int centesimi;
   public void assegnaCentesimi (int cent) {
      centesimi = cent;
   }
   public int leggiCentesimi () {
      return centesimi;
   }
}
```

La discendenza di una classe da un'altra viene stabilita per mezzo della parola chiave `extends`. Un'istanza della classe `Tempo2` incapsula tutti i dati della classe `Tempo` e, in più, incapsula `centesimi`. Su un'istanza di essa può essere invocato un qualsiasi metodo della classe genitrice. Tale meccanismo permette di risparmiare molto codice che altrimenti dovrebbe essere duplicato; inoltre, fattore di maggior rilevanza, è applicabile anche per realizzare vere analisi top-down, come predicato da anni dagli studiosi della programmazione, e consente di poter implementare il risultato dell'analisi direttamente nel linguaggio di programmazione. Nell'analisi di un problema infatti, si può cominciare descrivendo una generica `ClasseCheRisolveIlProblema` e successivamente derivarla e integrarla con classi che, di volta in volta, risolvano problemi sempre più specifici, fino ad arrivare all'implementazione minuta. Ecco quindi che il paradigma degli oggetti diventa un valido strumento per la descrizione dei problemi in generale.

9.5 Polimorfismo

Supponiamo di avere la seguente riga di codice in un programma Java:

```
a = b + c;
```

Le elaborazioni eseguite da un programma a fronte di un'istruzione come questa dipendono in modo decisivo dai tipi delle variabili coinvolte. Una somma fra variabili di tipo `double`, per esempio, richiede un'elaborazione diversa rispetto a una somma tra variabili di tipo `int`.

Fortunatamente, il compilatore solleva il programmatore dal dover scegliere di volta in volta il tipo d'elaborazione necessaria, permettendogli di scrivere solo la semantica dell'operazione. Questa caratteristica è presente praticamente in tutti i linguaggi di programmazione per i tipi previsti dal linguaggio, ma i linguaggi a oggetti vanno oltre, permettendo di poter usare questo tipo di semplificazione anche per gli oggetti definiti dal programmatore. Polimorfismo significa "molte forme" e, in parole povere, è la capacità di un linguaggio a oggetti di invocare metodi diversi con lo stesso nome a seconda degli oggetti coinvolti. La più semplice forma di polimorfismo può essere considerata il fatto che due classi differenti possono avere metodi con il medesimo nome. Le altre caratteristiche polimorfiche di Java sono le seguenti:

- in una classe e/o in una classe e in una sua superclasse, possono coesistere più metodi con lo stesso nome ma con argomenti differenti per tipo e/o per numero;
- oltre a questo, una classe può definire un metodo identico per nome, numero e tipo di argomenti ad uno già dichiarato in una sua superclasse.

Vediamo un primo esempio di polimorfismo nella nostra classe `Tempo`. In essa sarebbe utile disporre di un metodo che permetta di assegnare a un'istanza il contenuto di un'altra istanza dello stesso tipo; un tale metodo ha la stessa semantica di `assegnaTempo` per cui è comodo per il programmatore poter riutilizzare questo nome.

```
public int assegnaTempo (Tempo t) {
    ore = t.ore;
    minuti = t.minuti;
    secondi = t.secondi;
    return 0;
}
```

Questo tipo di polimorfismo è detto *sovraccarico dei metodi*, *methods overload*, a indicare che uno stesso metodo è sovraccaricato di più implementazioni. Il meccanismo è applicabile anche se i metodi vengono ridefiniti nelle classi derivate.

Nel nostro esempio, per assegnare a un'istanza di `Tempo2` un tempo completo di centesimi di secondo, è necessario invocare i metodi `assegnaTempo` e `assegnaCentesimi`. Sarebbe molto più comodo definire un metodo unico che esegua entrambe le invocazioni; un tale metodo ha di nuovo la stessa semantica di `assegnaTempo`, per cui poter mantenere questo nome migliora la leggibilità del programma. Il polimorfismo lo permette, per cui possiamo inserire nella classe `Tempo2` il seguente metodo:

```
public void assegnaTempo (int ora, int minuto, int secondo,
int cent) {
    assegnaTempo (ora, minuto, secondo);
    centesimi = cent;
}
```

Va notato che i metodi vengono distinti in base al nome e agli argomenti, ma non in base a cosa viene restituito; se si definiscono due metodi differenti solo per il tipo del valore restituito si ottiene un errore in fase di compilazione. Spesso ci si riferisce all'insieme nome+argomenti come alla *firma del metodo*, *method signature*.

9.6 Overriding

Nella classe `Tempo` abbiamo definito il metodo `visualizza`, il cui scopo è quello di rendere visibile sullo schermo il contenuto di un'istanza. Se invochiamo questo metodo da un'istanza di `Tempo2` ovviamente i centesimi non vengono visualizzati. Potremmo definire un metodo con un nome diverso, ma questo costringerebbe il programmatore a ricordare nomi diversi per funzionalità analoghe. Java permette di

sovrascrivere, *override*, in una classe derivata i metodi della classe genitrice. Ecco che quindi potremo definire in `Tempo2` il seguente metodo:

```
public void visualizza (boolean ritornoACapo) {
    System.out.print (ore);
    System.out.print (separatore);
    System.out.print (minuti);
    System.out.print (separatore);
    System.out.print (secondi);
    System.out.print (separatore);
    if (ritornoACapo)
       System.out.println (centesimi);
    else
       System.out.print (centesimi);
}
```

Provando a compilare la classe `Tempo2` così modificata, il compilatore segnala che si è cercato di accedere a dati non visibili. Ciò accade perché abbiamo dichiarato `ore`, `minuti` e `secondi` come variabili private e questo tipo di dichiarazione le rende inaccessibili anche alle classi derivate. Per poter accedere a una variabile d'istanza da un metodo di una classe derivata, è sufficiente utilizzare la parola riservata `protected` anziché `private`, dichiarando in tal modo che la variabile è visibile dalle sottoclassi. Così facendo, risolviamo il nostro problema, ma rendiamo il legame tra classe e sottoclasse più stretto. Se un giorno dovessimo modificare quindi la struttura interna della classe genitrice, ciò si rifletterebbe anche sulle classi derivate. In questi casi fortunatamente il compilatore aiuta moltissimo nell'individuare i punti da modificare.

Va notato che se in una sottoclasse si ridefinisce un metodo presente nella classe genitrice che si differenzia da quest'ultimo solo per il tipo del valore restituito, si ottiene un errore in fase di compilazione. In tal caso, infatti, non si può parlare di overriding in quanto i metodi sono diversi, e un overloading di questo tipo non è permesso, come detto nel paragrafo precedente.

✔ **NOTA**

Se il lettore esperimenta gli esempi riportati, si renderà conto che le variabili definite con accesso `protected` *sono in pratica visibili anche al di fuori delle sottoclassi. Per illustrare compiutamente i tipi di accesso disponibili in Java è necessario introdurre il concetto di* `package` *che verrà illustrato nel prossimo capitolo; in esso verranno compiutamente presentati i vari tipi di accesso e il loro significato.*

9.7 Binding dinamico

Una delle caratteristiche più utili dell'ereditarietà è quella che permette di trattare le istanze delle classi derivate nello stesso modo delle istanze della classe genitrice. Per comprendere bene questo concetto, supponiamo di dover gestire una porta automatica

di un edificio; questa porta deve aprirsi automaticamente quando un impiegato inserisce un badge magnetico, ma solo nell'orario di lavoro. Per ottenerlo definiamo una classe Accesso come segue:

```
class Accesso {
   private int oraInizio;
   private int oraFine;

   public void assegnaOraInizio (int ora) {
     oraInizio = ora;
   }

   public void assegnaOraFine (int ora) {
      oraFine = ora;
   }

   public boolean verificaOrario (Tempo t) {
       if (t.leggiOra() >= oraInizio &&
           t.leggiOra() <= oraFine)
          return true;
       else {
          t.visualizza(true);
          return false;
       }
   }
}
```

Lo scopo della classe è semplice; si crea un'istanza della classe Accesso e si assegna l'orario di inizio lavoro e fine lavoro per mezzo dei metodi opportuni. Ogni volta che qualcuno tenta di passare dalla porta automatica, si invoca il metodo verificaOrario passandogli il tempo corrente; se siamo nei limiti dell'orario di lavoro, il metodo restituisce true, il che consentirà al resto del programma di aprire la porta, mentre in caso contrario visualizzerà l'ora corrente e restituirà false, impedendo in tal modo l'apertura. La cosa interessante è che il parametro t di verificaOrario può essere, oltre che un object-id a cui corrisponde un'istanza di Tempo, anche un object-id di Tempo2 o di una sua classe derivata o di qualsiasi altra classe abbia tra i suoi progenitori la classe Tempo. Questo rende la classe Accesso più resistente ai cambiamenti. Nel metodo verificaOrario non è possibile invocare direttamente i metodi di una classe derivata in quanto il compilatore effettua un controllo sulle operazioni e non può prevedere con quale istanza verrà invocato il metodo. Però in quest'esempio c'è il metodo visualizza che è presente sia nella classe Tempo sia nella classe Tempo2. Se al parametro t corrisponde un'istanza di Tempo2, quale metodo sarà invocato? Per chiarire bene questo punto bisogna accennare al binding.

Il *binding* è l'operazione con cui il linguaggio lega il metodo da invocare a un'istanza. Questa operazione può avvenire durante la fase di compilazione, *binding statico* o *early binding*, oppure durante l'esecuzione del programma *binding dinamico* o *late*

binding. Se Java effettuasse il binding durante la fase di compilazione, l'unico metodo che potrebbe essere invocato è `visualizza` della classe `Tempo`. Java invece utilizza il binding dinamico dei metodi, quindi, nel nostro esempio, viene invocato il metodo `visualizza` della classe `Tempo2`. Grazie al binding dinamico, la classe `Accesso` visualizzerà correttamente le istanze di `Tempo2`, o di qualsiasi sottoclasse si renda necessaria per il programma, anche se fosse stato scritto anni prima di quest'ultima. Tra l'altro è proprio grazie a questo meccanismo che il metodo che utilizziamo per la visualizzazione dei dati, `System.out.println`, consente di visualizzare qualsiasi cosa, ma questo lo vedremo meglio più in avanti.

9.8 Operatori sulle istanze

Nel paragrafo precedente si è detto che non si può invocare direttamente un metodo di una classe derivata da un object-id della classe genitrice. Visto che il binding dei metodi viene effettuato in fase di esecuzione, questa è una limitazione dovuta solo alla presenza del compilatore, che può essere aggirata tramite il *cast*. Un cast consente di modificare temporaneamente la classe di riferimento di un object-id. Questa operazione si effettua premettendo tra parentesi tonde il nome della classe a cui si desidera promuovere l'object-id alla variabile contenente l'object-id stesso. Rifacendosi all'esempio precedente, sul parametro `t` è possibile invocare il metodo `leggiCentesimi` di `Tempo2` usando la seguente notazione:

```
((Tempo2) t).leggiCentesimi();
```

Il compilatore permette di effettuare un cast solo verso classi derivate o genitrici, anche se in quest'ultimo caso sono perfettamente inutili. Se a `t` non corrisponde un'istanza di `Tempo2`, si ha un errore in fase di esecuzione, per cui, prima di fare un'operazione del genere, è necessario essere sicuri del tipo dell'istanza corrispondente all'object-id. A questo scopo si può usare l'operatore `instanceof` che consente di stabilire se su un'istanza è possibile effettuare un cast o meno. L'istruzione precedente può essere quindi scritta in modo più sicuro come segue:

```
if (t instanceof Tempo2)
   ((Tempo2) t).leggiCentesimi();
```

Notare che in questo esempio un'espressione del tipo

```
t instanceof Tempo
```

restituisce sempre `true`; infatti si può dire che un'istanza di una classe derivata è istanza anche della classe genitrice.

9.9 Costruttori e finalizzazione

Facendo ancora riferimento all'esempio precedente supponiamo che al momento di invocare `verificaOrario` non si abbia a disposizione un'istanza della classe `Tempo` ma solo un intero, `oraCorrente`, contenente appunto l'ora corrente. In tal caso è

necessario creare un'istanza di `Tempo`, assegnarle il contenuto di `oraCorrente` e quindi passarne l'object-id a `verificaOrario`. Per esempio:

```
Tempo perPassaggio = new Tempo();
perPassaggio.assegnaTempo (oraCorrente, 0, 0);
mioAccesso.verificaOrario (perPassaggio);
```

Questi passaggi sono assai macchinosi e introducono un nuovo nome, `perPassaggio`, che non serve ad altro che a complicare il compito di chi legge questo brano di codice. Possono essere evitati tramite l'uso di un *costruttore*. I costruttori sono metodi speciali che vengono invocati automaticamente subito dopo la creazione dell'oggetto e ai quali è possibile passare dei parametri direttamente nella `new`. Tali metodi vengono dichiarati nella classe come qualsiasi altro metodo, ma si distinguono dal fatto che il loro nome coincide con quello della classe e per il fatto che non dichiarano nessun tipo di dato da restituire. In una classe possono coesistere più costruttori che però debbono differire per numero e/o tipo degli argomenti. Dichiarando quindi il seguente costruttore nella classe `Tempo`

```
public Tempo (int ora, int minuto, int secondo) {
   assegnaTempo (ora, minuto, secondo);
}
```

l'esempio precedente diverrebbe

```
mioAccesso.verificaOrario (new Tempo(oraCorrente, 0, 0));
```

con evidenti vantaggi di concisione e leggibilità. Una volta definito questo costruttore però il compilatore non permetterà più di creare un'istanza della classe `Tempo` senza passare tre interi come argomenti, a meno di non definire un costruttore senza argomenti. Questo comportamento, che può sembrare bizzarro, consente di avere delle classi in cui non è permesso creare un oggetto senza fornire dei parametri al costruttore. Classi di questo tipo sono utili quando un oggetto deve essere legato a un altro oggetto o a un valore e questo legame rimane valido per tutta la durata dell'oggetto.

Se una classe ha uno o più costruttori, uno di essi deve essere invocato necessariamente anche quando viene creata un'istanza di una sottoclasse. Esso può essere invocato implicitamente nel caso ne esista uno che non ha argomenti, mentre in caso contrario deve essere invocato esplicitamente tramite la notazione illustrata nel successivo paragrafo. È quindi una buona norma, quando possibile, fornire sempre una classe di un costruttore senza argomenti in modo tale da semplificarne l'utilizzo in una sottoclasse.

È possibile inizializzare delle variabili d'istanza con valori costanti o anche richiamando metodi opportuni. Supponiamo di dichiarare una classe `Tempo3` come la seguente:

```
class Tempo3 {
   public Tempo oreMinSec;
   int centesimi;
   void assegnaTempo (int ora, int minuto, int secondo,
                      int cent) {
```

```
        oreMinSec.assegnaTempo (ora, minuto, secondo);
        centesimi = cent;
    }
    int leggiOra() {
        return oreMinSec.leggiOra();
    }
    ...
}
```

Questa classe fornisce più o meno le stesse funzionalità di `Tempo2` non sfruttando l'ereditarietà ma utilizzando una tecnica che è detta di *composizione*. Così come è scritta però questa classe non funziona, in quanto alla variabile `oreMinSec` non corrisponde nessuna istanza, per cui otteniamo un errore in fase di esecuzione non appena si invoca un metodo che le si riferisce. È quindi necessario istanziare `oreMinSec`. La classe diventa allora:

```
class Tempo3 {
    public Tempo oreMinSec = new Tempo ();
    int centesimi;
    void assegnaTempo (int ora, int minuto, int secondo,
                       int cent) {
        oreMinSec.assegnaTempo (ora, minuto, secondo);
        centesimi = cent;
    }
    int leggiOra() {
        return oreMinSec.leggiOra();
    }
    ...
}
```

Quando viene creato un oggetto vengono quindi controllate tutte le variabili d'istanza: se hanno un valore d'inizializzazione, questo viene assegnato loro, altrimenti prendono il valore 0 se sono di tipo primitivo e `null` se sono referenze a oggetti o ad array. A questo punto viene invocato il costruttore adeguato, se esiste.

Se due o più costruttori debbono condividere una parte consistente di codice, può tornare utile usare un *inizializzatore d'istanza*, *instance initializer*. Si tratta di un blocco d'istruzioni, esterno a tutti i metodi, che viene eseguito prima dell'invocazione di un qualsiasi costruttore. L'esempio precedente potrebbe quindi essere trasformato nel modo seguente:

```
class Tempo3 {
    public Tempo oreMinSec;
    int centesimi;

    {
        oreMinSec = new Tempo ();
    }
```

```
    void assegnaTempo (int ora, int minuto, int secondo,
                       int cent) {
        oreMinSec.assegnaTempo (ora, minuto, secondo);
        centesimi = cent;
    }
    int leggiOra() {
        return oreMinSec.leggiOra();
    }
    ...
}
```

Da notare che un inizializzatore d'istanza può contenere istruzioni di qualsiasi tipo, tranne l'istruzione `return`.

Si è detto che un oggetto viene eliminato automaticamente dal sistema quando non ha più nessun object-id che si riferisce a esso. Se un oggetto impegna una risorsa del sistema, come un file o una porta di comunicazione, è opportuno che essa venga rilasciata prima della distruzione dell'oggetto. A tale scopo si può definire un metodo con nome convenzionale `finalize`. Per esempio:

```
protected void finalize () {
    ...
}
```

Questo metodo può essere richiamato anche da programma come un qualsiasi altro metodo. Bisogna però fare attenzione che Java invoca automaticamente il metodo `finalize` solo appena prima di riutilizzare l'area allocata per l'oggetto: ciò significa che, non solo non è prevedibile il momento in cui viene invocato, ma che addirittura può non essere mai richiamato. A differenza dei costruttori, il metodo `finalize` di una sottoclasse non richiama automaticamente il metodo corrispondente della classe genitrice. Se questo risultasse necessario, bisogna codificare manualmente l'invocazione con la notazione descritta nel paragrafo successivo.

9.10 super

Abbiamo visto nel paragrafo precedente che in certi casi è necessario invocare metodi definiti nella classe genitrice. Questo può essere ottenuto tramite la parola riservata `super`. Essa referenzia come `this` l'istanza corrente, ma costringe l'interprete a fare il binding con i metodi della classe genitrice. È così possibile definire il metodo `visualizza` di `Tempo2` nel più conciso modo seguente:

```
public void visualizza (boolean ritornoACapo) {
    super.visualizza (false);
    System.out.print (separatore);
    if (ritornoACapo)
        System.out.println (centesimi);
```

```
    else
        System.out.print (centesimi);
}
```

Se nell'esempio precedente non avessimo utilizzato `super`, avremmo ottenuto un loop ricorsivo con conseguente errore di stack overflow in fase di esecuzione. L'invocazione di un costruttore della classe genitrice può avvenire solo a patto che sia la prima istruzione di un costruttore. Il costruttore della classe `Tempo2` può essere scritto come segue:

```
Tempo2 (int ora, int minuto, int secondo, int centesimo) {
    super (ora, minuto, secondo);
    this.centesimo = centesimo;
}
```

Nell'esempio precedente è stato utilizzato `this` per distinguere un parametro d'ingresso da una variabile d'istanza. Un altro caso in cui necessita l'uso di `this` è quando sia necessario passare l'oggetto corrente a un metodo di un'altra classe. Un esempio è l'invocazione del metodo `verificaOrario` della classe `Accesso` dall'interno di un metodo della classe `Tempo`, come, per esempio il seguente:

```
if (mioAccesso.verificaOrario (this))
```

9.11 Variabili e metodi di classe

Come si è detto, le variabili dichiarate in una classe esistono solo nell'istanza della classe ed è per questo motivo che vengono dette variabili d'istanza; esiste però la possibilità di dichiarare nella classe delle variabili che esistono indipendentemente da qualsiasi istanza. Tali variabili sono perciò dette *variabili di classe* o anche *variabili statiche* e sono, di fatto, delle variabili globali a cui però può venire limitato l'accesso come alle variabili d'istanza. La definizione di una variabile di classe si fa premettendo la parola riservata `static` alla normale dichiarazione di una variabile. Potremmo, per esempio, dichiarare statica la variabile `separatore` della classe `Tempo` nel modo seguente:

```
static public char separatore;
```

Questa modifica non è casuale, ma ha una sua utilità: all'interno di un programma, infatti, normalmente i tempi vengono rappresentati nello stesso modo, per cui è inutile dover assegnare il separatore per ogni nuova istanza creata. In questo modo è sufficiente assegnare il separatore una volta, diciamo all'inizio del programma, e tutte le istanze lo condivideranno. Una variabile di classe è visibile dai metodi della classe di cui fa parte nello stesso modo di una variabile d'istanza. Al di fuori della classe invece per accedere a essa, si può usare il nome stesso della classe invece del nome di un object-id: nella classe `ProvaTempo1`, per esempio, è possibile inserire la seguente istruzione:

```
Tempo.separatore = ':';
```

✔ **NOTA**

In Java è possibile riferirsi a una variabile di classe anche usando un object-id come se fosse una variabile d'istanza. Questa notazione può essere fuorviante, per cui se ne sconsiglia l'uso.

Nel caso in cui si voglia dichiarare un oggetto statico che necessita di particolari inizializzazioni, nasce il problema di quando effettuare questa creazione, in modo da non legare l'utilizzo della classe ad altri moduli. Per risolvere il problema, esiste la possibilità di inserire un blocco d'istruzioni, detto *inizializzatore statico* (*static initializer*), che può essere definito in ogni classe. Questo blocco viene eseguito una sola volta appena viene caricata la classe. Un inizializzatore statico è individuato da un blocco esterno a tutti i metodi preceduto dalla parola chiave `static`.

Per riassumere quanto detto con un esempio potremmo così modificare il modulo `ProvaTempo1` in `ProvaTempo2`.

```
class ProvaTempo2 {
   static private Tempo tempoStatico = new Tempo();
   public static void main (String argv[]) {
      System.out.println("Inizio");
      Tempo mioTempo;
      mioTempo = new Tempo();
      mioTempo.assegnaTempo(22,30,5);
      tempoStatico.assegnaTempo(1,2,3);
      mioTempo.visualizza(true);
      tempoStatico.visualizza(true);
   }
   static {
      Tempo.separatore = ':';
      System.out.println("Static");
   }
}
```

Nell'esempio abbiamo supposto che la variabile `separatore` della classe `Tempo` sia stata dichiarata statica come illustrato. Il programma, una volta compilato e lanciato in esecuzione, produrrà il seguente risultato:

```
Static
Inizio
22:30:5
1:2:3
```

Le variabili di classe vengono spesso usate per definire dei valori o degli oggetti costanti che risultino utili per semplificare il lavoro di programmazione. Per esempio nella classe `Math` fornita con il linguaggio è presente la seguente definizione

```
public final static double PI =  3.14159265358979323846;
```

per semplificare il trattamento del pi greco.

✔ **NOTA**

La parola riservata `final` *significa che il valore non è modificabile. Verrà trattata nel prossimo capitolo.*

Allo stesso modo in cui si dichiarano le variabili di classe, si possono dichiarare i *metodi di classe*. Anche in questo caso i metodi di classe vengono distinti da quelli d'istanza premettendo alla dichiarazione la parola riservata `static`. Con essi ovviamente non si possono trattare le variabili d'istanza, ma solo le variabili di classe a meno di non fornire un'istanza come parametro. Di fatto questi metodi si comportano in modo del tutto analogo alle funzioni del C e si utilizzano di solito dove è richiesto il comportamento di una funzione matematica: nella classe `Math`, per esempio, è definito il seguente metodo statico:

```
public static native double cos(double a);
```

✔ **NOTA**

La parola riservata `native` *indica che il metodo non è implementato in Java, ma direttamente in codice macchina. Ne parleremo nel prossimo capitolo.*

9.12 Classi astratte

In alcuni casi è comodo creare delle classi che servono come base da cui derivare altre sottoclassi, ma che da sole non hanno significato in quanto hanno una parte indefinita. Una classe di questo tipo viene detta *astratta* e può essere dichiarata tale premettendo la parola chiave `abstract` alla dichiarazione della classe: la caratteristica principale di una classe astratta è che da essa non è possibile creare un'istanza. Una classe astratta tipicamente avrà dei metodi astratti, dei metodi cioè che non hanno implementazione. In tal modo si obbligano le classi derivate a fornire un'implementazione del metodo adeguata. Anche in questo caso un metodo astratto deve avere la dichiarazione preceduta dalla parola chiave `abstract`. Un metodo astratto non può essere dichiarato in una classe non astratta, mentre una classe astratta può non avere metodi astratti. L'esempio riportato nell'ultimo paragrafo del capitolo chiarisce l'uso di questi meccanismi che, detti così, possono risultare un po' "astratti".

9.13 UML

L'UML (Unified Modelling Language) è un tipo di notazione che serve a modellare un sistema informativo usando un approccio orientato agli oggetti. Un modello UML viene descritto tramite l'uso di diversi tipi di diagrammi, divisi in 5 categorie, ciascuno dei quali descrive un diverso aspetto di un sistema informativo. Esistono diversi programmi che aiutano a realizzare i modelli UML in modo visuale, tra questi Umbrello (`http://uml.sourceforge.net`) è un prodotto gratuito che supporta i seguenti tipi di diagrammi:

Categoria	Diagrammi	
Diagrammi introduttivi	Casi d'uso	Mostra gli attori (persone o altri computer che usano il sistema), i casi d'uso (scenari in cui gli attori usano il sistema) e le loro relazioni.
Diagrammi di struttura statica	Classi	Mostra le classi e le relazioni fra esse.
Diagrammi d'interazione	Sequenza	Mostrano lo scambio di messaggi (cioè la chiamata ai metodi) tra diversi oggetti in un rapporto temporale preciso.
	Collaborazione	Mostra le interazioni che avvengono tra gli oggetti che partecipano a una situazione specifica.
Diagrammi di stato	Stato	Mostra gli stati, i cambi di stato e gli eventi in un oggetto.
	Attività	Mostra la sequenza di attività in un sistema.
Diagrammi d'implementazione	Componenti	Mostra i componenti software e le librerie usate nel sistema.
	Dislocamento	Mostra le istanze dei componenti durante l'esecuzione e le loro associazioni.

In un diagramma delle classi, ciascuna classe è rappresentata come un rettangolo diviso in tre parti sovrapposte: nella prima compare il nome della classe, di solito in grassetto, nella seconda compaiono gli attributi che la compongono, mentre nella terza compaiono le operazioni o i metodi che possono essere eseguiti sugli oggetti. Per esempio

Tempo
-ore : int -minuti : int -secondi : int +separatore : char
+assegnaTempo (ora : int, minuto : int, secondo : int) : int +leggiOra () : int +leggiMinuti () : int +leggiSecondi () : int +aggiungiOre (numOre : int) : void +visualizza (ritornoACapo : boolean) : void

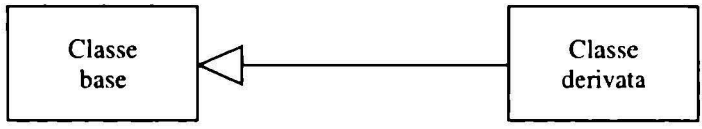
Come si vede la corrispondenza con le dichiarazioni Java è molto stretta, l'unica notazione che merita una spiegazione è l'uso del simbolo + per designare i membri pubblici e il simbolo - per designare i membri privati. Il simbolo usato per i membri protetti è #.

Un fattore molto importante nella descrizione di un'applicazione, e molto più difficile da desumere dalla semplice lettura dei sorgenti, è dato dalle relazioni che intercorrono tra le classi: UML prevede nel diagramma delle classi una notazione che permette di esplicitare diversi tipi di relazione. Quando si analizzano le relazioni tra classi, è più utile riuscire ad avere una visione d'insieme, piuttosto che riuscire a vedere i singoli membri di ciascuna classe, per cui in questi casi useremo dei rettangoli con i soli nomi delle classi. In ambienti di sviluppo reali i diagrammi vengono fatti solitamente usando programmi appositi che permettono vari livelli di "zoom", consentendo così di avere sia una visione globale che di dettaglio.

L'UML prevede le seguenti relazioni.

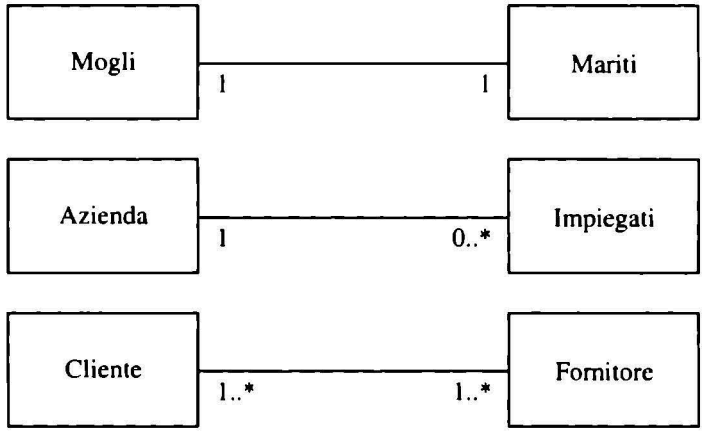
Generalizzazione

In UML, un'associazione di *generalizzazione* tra due classi le mette in una gerarchia che rappresenta il concetto di ereditarietà di una classe derivata da una classe di base. In UML le generalizzazioni sono rappresentate da una freccia che connette le due classi e punta alla classe di base.



Associazione

Le associazioni designano una relazione tra classi diverse. Le relazioni possono essere di tipo 1 a 1, 1 a molti e molti a molti e vengono rappresentate da una linea alle cui estremità è riportata la molteplicità.



Aggregazione

Le aggregazioni sono tipi particolari di associazione di tipo intero-parte. La rappresentazione grafica è una linea con un rombo vuoto sul lato dell'intero.

Composizione

Anche le composizioni sono associazioni di tipo intero-parte, ma a differenza delle aggregazioni, rappresentano un legame così forte da non permettere l'esistenza di una parte senza l'esistenza dell'intero. Le composizioni sono rappresentate da una linea con un rombo pieno sul lato dell'intero.

9.14 Un esempio riepilogativo

Nell'esempio che segue mettiamo in pratica i principali concetti visti in questo capitolo. Supponiamo quindi di volere implementare una classe che serva da contenitore per degli oggetti tale da poterli ordinare secondo criteri da stabilire. Per fare questo occorre prima precisare che in Java le classi, che non derivano esplicitamente da un'altra classe, derivano implicitamente dalla classe `Object` che verrà illustrata più in dettaglio nel Capitolo 11; ciò implica che qualsiasi classe Java deriva in maniera diretta o indiretta da `Object`. Definiamo dunque la classe `VettoreOrdinato` nel modo seguente:

```
abstract class VettoreOrdinato {
  private Object vettore[];
  private int maxElementi;
  private int curElementi;

  public VettoreOrdinato (int maxElementi) {
     this.maxElementi = maxElementi;
     vettore = new Object[maxElementi];
     curElementi = 0;
  }

  protected boolean aggiungi (Object elemento) {
     if (elemento != null && curElementi < maxElementi) {
        vettore[curElementi++] = elemento;
```

```
            return true;
        } else
            return false;
    }

    public Object leggi (int indice) {
        if (indice >= 0 && indice < curElementi)
            return (vettore[indice]);
        else
           return null;
    }

    public int maxElementi () { return maxElementi; }
    public int elementi () { return curElementi; }

    public void ordina () {
        int   s, i, j, num;
        Object temp;

        num = curElementi;
        for (s = num / 2; s > 0; s /= 2)
            for (i = s; i < num; i++)
                for (j = i - s; j >= 0; j -= s)
                    if (confronta (vettore[j], vettore[j + s])) {
                        temp = vettore[j];
                        vettore[j] = vettore[j + s];
                        vettore[j + s] = temp;
                    }
    }
    abstract protected boolean confronta (Object elemento1,
                                          Object elemento2);
}
```

Questa classe è stata dichiarata astratta in quanto, per poter funzionare, necessita di conoscere il criterio di ordinamento degli oggetti che deve contenere. Per motivi di semplicità, utilizziamo un vettore per contenere gli object-id degli oggetti: poiché un vettore deve essere dimensionato, abbiamo messo un costruttore con un argomento che indica il massimo numero di oggetti che può essere contenuto. Abbiamo poi dei metodi che servono per inserire e leggere gli elementi e per fornire indicazioni sul numero attuale e massimo di oggetti contenuti. Abbiamo infine il metodo `ordina` che esegue l'ordinamento tramite un algoritmo non molto efficiente, ma molto compatto, detto shell sort. Questo algoritmo utilizza il metodo `confronta` per stabilire l'ordinamento dei singoli oggetti: questo metodo è definito astratto in modo da obbligare la sottoclasse a implementare un metodo che svolga la funzione di confronto: esso dovrà restituire `true` se il primo argomento è maggiore del secondo, o meglio se il primo argomento si deve trovare in una posizione successiva al secondo nella sequenza di ordinamento, `false` altrimenti.

A questo punto, per provare il funzionamento di questa classe è necessario derivarne una sottoclasse che faccia riferimento a degli oggetti definiti. Deriviamo quindi la classe `VettoreTempo` destinata a contenere oggetti della classe `Tempo`. In quest'ultima classe però non abbiamo dei metodi che ci consentano di confrontare due tempi per stabilire un ordine. Aggiungiamo quindi il metodo `maggioreDi` in modo da non dover trattare direttamente con le variabili d'istanza. La classe `Tempo` finale diventa dunque la seguente:

```
class Tempo {
   protected int ore;
   protected int minuti;
   protected int secondi;
   static public char separatore;

   public Tempo () {
      ore = minuti = secondi = 0;
   }
   public Tempo (int ora, int minuto, int secondo) {
      this();
      assegnaTempo (ora, minuto, secondo);
   }
   public int assegnaTempo (int ora, int minuto,
                                        int secondo) {
      if (ora >= 0 && ora < 24 &&
          minuto >= 0 && minuto < 60 &&
          secondo >= 0 && secondo < 60) {
         ore = ora;
         minuti = minuto;
         secondi = secondo;
         return 0;
      } else
         return -1;
   }

   public int leggiOra () {
      return ore;
   }

   public int leggiMinuti () {
      return minuti;
   }

   public int leggiSecondi () {
      return secondi;
   }
```

```
    public void aggiungiOre (int numOre) {
        ore = ore + numOre;
        while (ore > 23)
            ore = ore - 24;
    }

    public void visualizza (boolean ritornoACapo) {
        System.out.print (ore);
        System.out.print (separatore);
        System.out.print (minuti);
        System.out.print (separatore);
        if (ritornoACapo)
            System.out.println (secondi);
        else
            System.out.print (secondi);
    }

    public boolean maggioreDi (Tempo t) {
        if (ore > t.ore ||
        ore == t.ore && minuti > t.minuti ||
            ore == t.ore && minuti == t.minuti && secondi
                                                > t.secondi)
            return true;
        else
           return false;
    }
}
```

Si noti che il costruttore senza argomenti assegna 0 a `ore`, `minuti` e `secondi`; l'operazione è inutile in quanto tali variabili vengono inizializzate a 0 automaticamente, ma l'esplicitazione dell'operazione rende più evidente il comportamento del costruttore. Per motivi analoghi nel costruttore con tre argomenti interi abbiamo innanzitutto invocato il costruttore senza argomenti per mezzo dell'istruzione `this()` per essere certi che l'istanza contenga i valori di default nel caso fallisca il metodo `assegnaTempo` a causa di valori incongruenti: questo tipo di invocazione deve essere sulla prima istruzione, analogamente all'invocazione con `super`.

Finalmente possiamo implementare la classe `VettoreTempo` nel modo seguente:

```
class VettoreTempo extends VettoreOrdinato {
    VettoreTempo () {
        super (10);
    }

    VettoreTempo (int elementi) {
        super (elementi);
    }
```

```
    protected boolean aggiungi (Object elemento) {
        return false;
    }

    protected boolean aggiungi (Tempo elemento) {
        return super.aggiungi(elemento);
    }

    protected boolean confronta (Object elemento1,
                                 Object elemento2) {
        return ((Tempo) elemento1).maggioreDi((Tempo)
                                              elemento2);
    }
}
```

In questa classe si è dichiarato un costruttore senza argomenti che dimensiona il contenitore per default a 10 elementi massimo e un costruttore che permette di stabilire la dimensione voluta. Per essere sicuri che questo contenitore contenga solo oggetti della classe `Tempo`, o oggetti di classi derivate, abbiamo sovrascritto il metodo `aggiungi (Object elemento)` in modo tale da impedire che vengano inseriti oggetti qualsiasi, e abbiamo definito il metodo `aggiungi (Tempo elemento)`. In quest'ultimo abbiamo dovuto usare `super` per evitare che venga richiamato il metodo che non effettua l'inserzione. Nel metodo `confronta` abbiamo dovuto dichiarare gli argomenti come object-id della classe `Object`, anche se siamo sicuri che corrisponderanno sempre a istanze di `Tempo`, poiché altrimenti il metodo non avrebbe corrisposto al metodo astratto dichiarato nella classe genitrice. Questo ci obbliga a usare dei cast che complicano un po' la lettura, ma che possono essere usati con tranquillità senza bisogno di controlli aggiuntivi.

Una schematizzazione delle relazioni tra le classi viene fornita dal seguente diagramma delle classi:

Infatti la classe `VettoreTempo` ha una *associazione 1 a molti* con la classe `Tempo`: il vettore è composto da più elementi ciascuno dei quali memorizza un tempo. La classe base `VettoreOrdinato` è una *generalizzazione* della classe derivata `VettoreTempo`.

A questo punto possiamo verificare il funzionamento delle classi descritte con una classe `ProvaVettore` come segue:

```
class ProvaVettore {
    public static void main (String argv[]) {
        VettoreTempo vt = new VettoreTempo (12);
        int ore, minuti, secondi;
```

```
    Tempo.separatore = ':';
    for (int i = 0; i < 12; i++) {
        ore = (int) (Math.random () * 24);
        minuti = (int) (Math.random () * 60);
        secondi = (int) (Math.random () * 60);
        vt.aggiungi (new Tempo (ore, minuti, secondi));
    }
        vt.ordina();
        for (int i = 0; i < vt.elementi(); i++)
          ((Tempo)vt.leggi (i)).visualizza(true);
    }
}
```

Questa classe crea un'istanza della classe `VettoreTempo` che può contenere 12 elementi e vi inserisce 12 istanze della classe `Tempo` con valori casuali. Per creare i valori casuali è stato usato il metodo statico `Math.random()` che fornisce un numero in doppia precisione compreso tra 0.0 e 1.0. Abbiamo dovuto utilizzare un cast sul metodo `leggi`, in quanto questo restituisce un `Object`. Va notato che nella classe `VettoreTempo` non avremmo potuto ridefinire un metodo `leggi` che restituisse un'istanza della classe `Tempo`, in quanto questo sarebbe stato diverso dal metodo della classe genitrice solo per il tipo restituito e questo non è permesso. Avremmo però potuto definire un metodo `leggiTempo` che restituisse un oggetto della classe `Tempo` richiamando il metodo `leggi` ed effettuando un cast opportuno.

Il risultato del programma precedente differisce ovviamente di volta in volta: uno dei possibili risultati è il seguente:

```
0:10:38
0:15:1
3:20:54
4:35:13
5:45:25
6:2:17
9:59:40
16:27:3
17:6:1
17:40:11
22:30:25
23:16:5
```

Domande di verifica

1. Dare una definizione del termine "classe" nella programmazione orientata agli oggetti. Esporre alcuni esempi di classi e oggetti.
2. Che cosa differenzia una variabile o un metodo dichiarato `public` da uno dichiarato `private`?

3. Esporre il concetto di "oggetto" in relazione a "classe"?
4. Come si istanzia un oggetto?
5. I metodi di una classe hanno visibilità sulle variabili della classe se queste sono dichiarate `private`?
6. Che cosa si intende per variabili o metodi d'istanza?
7. Come si può fare in modo che una classe sia visibile da qualsiasi altra classe?
8. Perché si usa il termine incapsulamento per indicare la caratteristica della programmazione a oggetti che permette di rendere visibili i dati solo ai metodi specificatamente preposti?
9. Cosa si intende per superclasse e classe derivata?
10. Fare un esempio in cui si modella la realtà attraverso un albero gerarchico di classi.
11. Spiegare il significato di polimorfismo nella programmazione a oggetti.
12. In Java si possono dichiarare nella stessa classe due metodi che si differenziano solamente per il valore di ritorno.
13. Spiegare con un esempio il significato di binding dinamico.
14. Cosa si intende per operazione di casting su un oggetto? Perché è utile l'operatore `instanceof`?
15. Che cosa è un metodo costruttore? A quale scopo è utile l'inizializzatore di istanza?
16. Spiegare i significati di `this` e `super`.
17. Che cosa differenzia le variabili di classe (o statiche) da quelle d'istanza?
18. Come si rappresentano generalizzazioni, associazioni, aggregazioni e composizioni nella notazione UML?

Esercizi

1. Scrivere una classe `Data` per la gestione delle date analoga a `Tempo`. Porre attenzione che nelle date non ha senso assegnare un valore di default nel caso di impostazione errata, per cui deve essere gestito anche il caso di data errata e data nulla.
2. Derivare dalla classe dell'esercizio precedente una classe `DataEOra` che permetta di contenere sia la data che l'ora.
3. Nella classe dell'esercizio precedente inserire un esempio di overload e uno di overriding.

4. Scrivere un programma con un array contenente istanze di oggetti di classi diverse e scrivere un metodo in grado di visualizzarne il contenuto.

5. Sareste in grado di dare una spiegazione al fatto che il metodo `System.out.println` riesce a visualizzare qualsiasi cosa? Se sì, provate a implementare un metodo analogo.

6. Nell'esempio riepilogativo alla fine del capitolo, modificare la classe `ProvaVettore` in modo che i valori da ordinare di ore, minuti e secondi siano richiesti e quindi immessi dall'utente.

7. Nell'esempio riepilogativo alla fine del capitolo, fare in modo che l'ordinamento del vettore avvenga prima in base ai secondi, poi ai minuti infine alle ore. Per cui si avrà che 23:11:15 è minore di 15:10:28.

8. Nell'esempio riepilogativo alla fine del capitolo, utilizzare per l'ordinamento del vettore il metodo del bubble sort nella sua forma più semplice (e meno efficiente) esaminato nel Capitolo 6.

9. Nell'esempio riepilogativo alla fine del capitolo, richiedere all'utente se desidera che siano considerati anche i centesimi di secondo. Nella soluzione si utilizzi la classe `Tempo2` derivata da `Tempo`, che includerà i centesimi e in cui saranno opportunamente sovrascritti alcuni metodi della classe generatrice.

10. Modificare l'esercizio precedente facendo sì che i valori da ordinare di ore, minuti, secondi ed eventualmente centesimi di secondo siano richiesti e quindi immessi dall'utente. In questo modo si potrà verificare che effettivamente l'ordinamento avviene correttamente anche per i centesimi di secondo.

Capitolo 10

Oltre le classi

Obiettivi didattici

- Classi trasversali: `interface`
- Classi contenitrici (collezioni) e interfacce `generics`
- Programmazione per componenti: `package`
- Modificatori di classe, interfaccia, metodo variabile
- Interfacce e classi annidate

10.1 Interfacce

Il meccanismo dell'ereditarietà permette di costruire dei modelli anche complessi che descrivono in modo accurato un problema o una realtà fisica, consentendo un ampio riutilizzo del codice. Le gerarchie di classi sono costruzioni logiche che ci aiutano a semplificare la comprensione del mondo, ma non sono valide universalmente. Per esempio, il visone è un mammifero carnivoro che appartiene alla classe dei mustelidi insieme a donnole e puzzole, ma è possibile inserirlo nella classe degli animali di cui si usa la pelle, insieme a coccodrilli e pitoni, oppure inserirlo nella classe degli animali che si allevano insieme a polli e trote. Una gerarchia di classi deve essere quindi organizzata in modo diverso secondo il tipo di problemi che s'intendono affrontare e tipicamente non esiste una gerarchia adatta a risolvere qualsiasi problema. Capita però che sia vantaggioso trattare oggetti di classi del tutto differenti in modo generale e omogeneo, in considerazione di alcune caratteristiche trasversali comuni che nulla hanno a che vedere con la gerarchia di classi utilizzata.

Supponiamo di avere la classe `Punto` con la seguente struttura.

```
class Punto
{
   int x;
   int y;
   public Punto (int x, int y) {
      this.x = x;
      this.y = y;
   }
   int leggiX() {
      return x;
   }
   int leggiY() {
      return y;
   }
   int distanza() {
      return (int) Math.sqrt(x*x + y*y);
   }
}
```

Questa classe serve per memorizzare dei punti di un piano mediante le coordinate `x` e `y`. Il metodo `distanza` stabilisce la distanza dall'origine, avvalendosi per il calcolo di `Math.sqrt(double d)` un metodo fornito con le classi del linguaggio, che restituisce un `double` contenente la radice quadrata del parametro in ingresso `d`, nell'esempio il valore di `x*x+y*y`.

È evidente come la classe `punto` non abbia molto in comune con la classe `Tempo` definita nel capitolo precedente, eppure negli oggetti di entrambe possiamo trovare una caratteristica comune: il fatto di poter essere ordinati secondo un criterio univoco. In effetti, si può desiderare di ordinare oggetti di un gran numero di classi, secondo criteri diversi. Abbiamo già visto un modo che permette di scrivere un algoritmo utilizzabile per l'ordinamento di oggetti di una qualsiasi classe. A tale fine è stato necessario dichiarare una classe specializzata per ogni classe che si intende ordinare, anche se tale classe contiene poco codice. Sarebbe invece molto più comodo dire che una qualsiasi classe è ordinabile se ha un metodo `maggioreDi` che consenta di confrontare due istanze e di stabilire quale delle due deve precedere l'altra. Ecco quindi che abbiamo delineato una specie di classe trasversale che accomuna classi diverse la cui unica caratteristica comune è quella di avere un metodo con lo stesso nome, o meglio, con la stessa firma. Potremmo poi avere una classe che ordina questa classe trasversale, senza la necessità di avere classi specializzate.

Per affrontare questo tipo di problemi, in Java è stato inserito il concetto di *interfaccia* (interface).

Le interfacce sono come le classi ma possono contenere solo variabili statiche e metodi astratti, cioè senza corpo, il che significa che si possono definire interfacce senza fare nessuna assunzione sull'implementazione. Le gerarchie delle interfacce sono totalmente disgiunte dalle gerarchie delle classi.

La dichiarazione di un'interfaccia è del tutto analoga a quella di una classe, basta sostituire la parola `class` con la parola `interface`; la sintassi è la seguente.

```
interface nome-interfaccia [extends int-genitrice [, int-genitrice ...]]
{
    [tipo1 nome-variabile1 = valore1;
    ...
    tipoN nome-variabileN = valoreN;]

    tipo-restituito1 metodo1(arg1);
    ...
    tipo-restituitoN metodoN(argN);
}
```

Per esempio

```
interface Ordinabile
{
    public boolean maggioreDi (Ordinabile o);
}
```

Come risulta evidente i metodi di un'interfaccia, nell'esempio ne è presente soltanto uno, hanno al posto del corpo un semplice punto e virgola. Non occorre usare la parola riservata `abstract` perché è implicita per tutti i metodi di un'interfaccia. Analogamente se avessimo dichiarato delle variabili, esse sarebbero state implicitamente statiche. Come si vede dalla sintassi, una variabile deve essere necessariamente inizializzata, perché le variabili d'interfaccia non possono essere modificate; in effetti, non sono variabili, ma piuttosto costanti simboliche. Vedremo nel seguito che è possibile dichiarare costanti di questo tipo anche al di fuori delle interfacce per mezzo della parola chiave `final`.

Un'interfaccia può estendere i metodi di un'altra interfaccia, esattamente come accade per le classi. Nel nostro esempio l'argomento del metodo `maggioreDi` è di tipo `Ordinabile`: ovviamente non si può creare un'istanza di un'interfaccia con l'istruzione `new`, visto che sarebbe vuota, per cui quando un oggetto è di un tipo corrispondente a un'interfaccia, significa che appartiene a una classe che *implementa* quell'interfaccia.

Una classe può *implementare* una o più interfacce dichiarandole esplicitamente e implementando i metodi dichiarati nelle interfacce stesse. In tal caso, gli oggetti di questa classe saranno riconosciuti anche come oggetti che implementano l'interfaccia. La dichiarazione d'implementazione all'interno di una classe si ottiene per mezzo della parola chiave `implements` seguita dal nome di una o più interfacce separate da virgole. Possiamo quindi dichiarare la classe `Punto` in modo che implementi `Ordinabile` nel modo seguente.

```
public class Punto implements Ordinabile
{
    int x;
    int y;
```

```
    public Punto (int x, int y) {
        this.x = x;
        this.y = y;
    }
    int leggiX() {
        return x;
    }
    int leggiY() {
        return y;
    }
    int distanza() {
        return (int) Math.sqrt(x * x + y * y);
    }
    public boolean maggioreDi (Ordinabile o) {
        if (o instanceof Punto)
            return distanza() > ((Punto) o).distanza();
        else
            return false;
    }
}
```

Nel metodo `maggioreDi` sappiamo solo che l'argomento implementa la classe `Ordinabile` e non c'è nessuna garanzia che appartenga alla classe `Punto`. Per questo motivo viene prima controllata l'appartenenza dell'argomento alla classe e quindi effettuato un cast. Nell'esempio è stato scelto come criterio d'ordinamento la distanza dall'origine.

Vediamo ora come viene modificata la classe `VettoreOrdinato` vista nel capitolo precedente in modo che possa funzionare con le interfacce.

```
public class VettoreOrdinato
{
  private Ordinabile vettore[];
  private int maxElementi;
  private int curElementi;

  public VettoreOrdinato (int maxElementi) {
    this.maxElementi = maxElementi;
    vettore = new Ordinabile[maxElementi];
    curElementi = 0;
  }

  public boolean aggiungi (Ordinabile elemento) {
    if (elemento != null && curElementi < maxElementi) {
       vettore[curElementi++] = elemento;
       return true;
    } else
```

```
        return false;
    }

    public Ordinabile leggi (int indice) {
      if (indice >= 0 && indice < curElementi)
         return (vettore[indice]);
      else
       return null;
    }

    public int maxElementi () { return maxElementi; }

    public int elementi () { return curElementi; }

    public void ordina () {
      int   s, i, j, num;
      Ordinabile temp;

      num = curElementi;
      for (s = num / 2; s > 0; s /= 2)
        for (i = s; i < num; i++)
           for (j = i - s; j >= 0; j -= s)
              if (vettore[j].maggioreDi (vettore[j + s])) {
                 temp = vettore[j];
                 vettore[j] = vettore[j + s];
                 vettore[j + s] = temp;
              }
    }
}
```

Si può notare in questa classe che, dal punto di vista del trattamento, non c'è nessuna differenza tra istanze di classi e istanze di classi che implementano interfacce. Scriviamo ora una classe che ci permetta di verificare il funzionamento di quanto visto. Il codice

```
public class TestOrdinamento2
{
   public static void main (String argv[]) {
      VettoreOrdinato vo = new VettoreOrdinato(10);
      vo.aggiungi (new Punto(30,40));
      vo.aggiungi (new Punto(300,400));
      vo.aggiungi (new Punto(3,4));
      vo.ordina();
      for (int i = 0; i < vo.elementi(); i++)
         System.out.println(((Punto)(vo.leggi
                                          (i))).distanza());
   }
}
```

produce il risultato

```
5
50
500
```

La soluzione è più flessibile rispetto a quella vista nel capitolo precedente, anche se abbiamo dovuto usare dei cast che rendono il codice più insicuro. Vedremo nel paragrafo successivo come, a partire dalla versione 5 di Java, sia possibile evitare questo tipo di insicurezza, permettendo al compilatore di verificare meglio la correttezza dei programmi.

10.2 Generic

Un compito molto comune che ci si trova ad affrontare quando si sviluppano programmi è quello di dover raggruppare un numero indefinito di oggetti dello stesso tipo. Per esempio potremmo dover raggruppare tutti gli articoli acquistati da un cliente per formare uno scontrino fiscale, oppure le parole che formano un testo o ancora i poligoni che formano un'immagine in tre dimensioni. Abbiamo già visto che gli array monodimensionali possono assolvere questo compito, ma hanno due limitazioni che possono renderne l'uso complicato: debbono essere dimensionati prima dell'utilizzo e permettono di ritrovare un elemento solo in base a un'associazione con un numero. In molti casi invece torna utile avere delle funzionalità in più, come, per esempio, disporre un vettore che cresce automaticamente in base alle necessità oppure poter associare un elemento a un nome anziché a un numero. In questi casi si creano delle classi apposite che svolgono questi compiti; classi di questo genere vengono chiamate *contenitori* (*container*) o collezioni (*collection*).

Vediamo allora come sia possibile implementare una classe `Vettore` molto semplice capace di contenere un numero indefinito di elementi.

```
public class Vettore {
   private Object vettore[] = new Object[0];

   public void metti (Object elemento, int posizione) {
      if (posizione >= vettore.length) {
         Object nuovo[] = new Object[posizione + 1];
         for (int i = 0; i < vettore.length; i++)
             nuovo[i] = vettore[i];
         vettore = nuovo;
      }
      vettore[posizione] = elemento;
   }

   public Object prendi (int posizione) {
      if (posizione >= vettore.length)
         return null;
```

```
        else
            return vettore[posizione];
    }
    int lunghezza() {
        return vettore.length;
    }
}
```

Si può notare come gli elementi vengano memorizzati in array di `Object`, che quando diventa di dimensione insufficiente viene sostituito con uno di capacità adeguata.

Questa classe può memorizzare qualsiasi tipo di oggetto, ma obbliga il programmatore a usare dei cast per utilizzare poi l'oggetto ricuperato. La correttezza dell'uso dei cast non è verificabile dal compilatore e questo rende i programmi soggetti a errori.

10.2.1 Evitare i cast

Per risolvere questo problema, dalla versione 5 di Java sono state introdotte le cosiddette *classi e interfacce generiche* (*generic*). Nella terminologia Java si parla semplicemente di generic perché tutto quello che riguarda questo argomento si applica ugualmente sia a classi che a interfacce, per cui risulta inutile e scomodo ripetere ogni volta "classe o interfaccia generica". Anche noi useremo questa convenzione, usando il termine inglese per evitare ambiguità e scegliendo arbitrariamente il genere maschile perché più comunemente usato.

Un generic è una classe/interfaccia legata in qualche modo a una o più classi diverse delle quali però non necessita conoscere alcuna caratteristica particolare. Le collezioni sono il campo di applicazione tipico dei generic.

Per dichiarare un generic è necessario fornire, tra parentesi angolari, < e >, un numero di *tipi parametrici formali* pari al numero di classi/interfacce diverse che il generic dovrà trattare. La sintassi è la seguente:

```
class nome-classe<nome-fittizio [,nome-fittizio ...]>
        [entends ...]

interface nome-classe<nome-fittizio [,nome-fittizio ...]>
        [implements ...]
```

Dato che non si sa a priori quali saranno realmente le classi/interfacce trattate, si dà loro un nome fittizio che servirà per referenziarle nel generic. Tornando all'esempio del `Vettore`, questa classe può essere trasformata in generic dichiarandola nel modo seguente

```
public class Vettore<T> {
    private Object vettore[] = new Object[0];

    public void metti (T elemento, int posizione) {
        if (posizione >= vettore.length) {
            Object nuovo[] = new Object[posizione + 1];
```

```
            for (int i = 0; i < vettore.length; i++)
                nuovo[i] = vettore[i];
            vettore = nuovo;
        }
        vettore[posizione] = elemento;
    }

    public T prendi (int posizione) {
        if (posizione >= vettore.length)
            return null;
        else
            return (T) vettore[posizione];
    }
    int lunghezza() {
        return vettore.length;
    }
}
```

In questa classe il tipo parametrico formale è stato chiamato `T` ed è stato sostituito in tutte le occasioni dove era stato usato `Object`; il vettore contenente gli object-id è però rimasto di tipo `Object` perché, pur essendo permesso creare variabili del tipo parametrico, non è permesso creare un array del tipo parametrico. Questo ci ha obbligato a usare un cast nel metodo `prendi` che però non inficia l'utilità dei generic: senza di essi infatti avremmo dovuto usare un cast ogni volta che prendiamo qualsiasi elemento di qualsiasi oggetto `Vettore` dichiarato nel programma.

Per usare un'istanza di un generic è necessario corredare il nome della classe con i *tipi parametrici attuali* che si vuole trattare in quel momento. Supponendo di voler creare un'istanza della classe `Vettore` adatta a contenere oggetti di tipo `String`, la dichiarazione sarà la seguente:

```
Vettore<String> v = new Vettore<String>();
```

L'istanza `v` accetterà quindi solo oggetti di tipo `String` nel metodo `metti` e restituirà oggetti di tipo `String` dal metodo `prendi`; ogni violazione di questa regola sarà segnalata dal compilatore con un apposito errore. Questi controlli sono utili ed evitano che vengano inclusi nel programma errori di battitura o di distrazione. Nei casi in cui sia necessario memorizzare un insieme di oggetti diversi tra loro, si può usare la seguente dichiarazione:

```
Vettore<Object> v = new Vettore<Object>();
```

Va detto che è possibile anche istanziare un generic senza fornire alcun parametro attuale e in tal caso l'istanza si comporta come se non fosse un generic e avesse tutti i tipi formali equivalenti a `Object`. La precedente dichiarazione può essere sostituita in modo quasi equivalente dalla seguente:

```
Vettore v = new Vettore();
```

Il motivo di questa "quasi equivalenza" verrà spiegato nel successivo paragrafo.

10.2.2 Assegnazione di generic

Quanto visto sui generic fino a questo punto è facilmente comprensibile; le cose però diventano un po' più complesse quando si tratta di capire in cosa differiscono due istanze dello stesso generic create con parametri diversi e come funzionano gli assegnamenti che le coinvolgono. L'importanza di questo argomento non riguarda tanto gli assegnamenti espliciti, che non sono di comune utilizzo tra generic, ma piuttosto gli assegnamenti impliciti, che vengono fatti tra parametri attuali e parametri formali nell'invocazione di un metodo: può infatti accadere che sia utile scrivere un metodo che riceva come argomento un'istanza di un generic creata con parametri differenti e per non avere errori in compilazione bisogna conoscere le regole di assegnamento.

Per affrontare questo argomento conviene fare prima due premesse:

- Java da sempre fornisce un'ampia libreria di collezioni, tra cui la classe `Vector` funzionalmente analoga alla classe `Vettore` degli esempi precedenti. Passando dalla versione 1.4 alla 5, tutte le collezioni sono state trasformate in generic, ma è stata conservata la compatibilità tra le classi vecchie e le nuove perché Java era già largamente diffuso e, senza compatibilità, migliaia di programmi sarebbero dovuti essere riscritti. Ecco perché è possibile istanziare un generic senza fornire parametri.
- Il meccanismo dei generic riguarda praticamente solo la compilazione e non introduce realmente un nuovo tipo di classe: se compiliamo le due versioni della classe `Vettore` viste nel paragrafo precedente e successivamente le decompiliamo, si vede che il byte code è sostanzialmente lo stesso e che la seconda include semplicemente delle informazioni in più al solo scopo di permettere al compilatore di fare i controlli opportuni. Quando invece si usa un'istanza di un generic, il compilatore si incarica di mettere i cast opportuni, analogamente a quanto faremmo noi usando una classe o interfaccia non generica, verificando però la correttezza delle operazioni e segnalando le condizioni di errore.

Come regola generale, a una variabile di un generic può essere assegnata soltanto un'istanza della stessa classe istanziata con gli stessi parametri. Per capire il perché di questa restrizione, supponiamo di poter scrivere le seguenti istruzioni:

```
Vettore<String> v1 = new Vettore<String>();
Vettore<Object> v2 = v1; // Errore in compilazione!
```

È evidente che, se questo fosse possibile, avremmo una sola istanza di `Vettore` nella quale possono essere inseriti oggetti qualsiasi usando la variabile `v2`: estraendo elementi di tipo diverso da `String` per mezzo della variabile `v1`, otterremo perciò un errore di cast. Naturalmente quanto detto è perfettamente valido anche nella situazione simmetrica.

```
Vettore<Object> v1 = new Vettore<Object>();
Vettore<String> v2 = v1; // Errore in compilazione!
```

In presenza di generic senza parametri però il compilatore "rilassa" i controlli e permette di compilare anche codice che può portare a errori, per questioni di compatibilità con le versioni precedenti: ecco allora che le seguenti istruzioni vengono compilate:

```
Vettore<String> v1 = new Vettore<String>();
Vettore v2 = v1; // OK!?!
```

Anche la situazione simmetrica è ammessa:

```
Vettore v1 = new Vettore();
Vettore<String> v2 = v1; // OK!?!
```

Questi strani comportamenti derivano da esigenze pratiche, ma il loro utilizzo è da evitare se si tiene a realizzare programmi robusti e facilmente comprensibili. Nella successiva trattazione ignoreremo quindi questo aspetto di compatibilità e, del resto, non è detto che in qualche futuro rilascio del linguaggio non vengano proibiti gli assegnamenti del tipo appena visti.

10.2.3 Carattere jolly

Resta il fatto che, per quanto detto finora, i generic dello stesso tipo non hanno un'organizzazione gerarchica in base ai parametri. In certi casi invece risulta comodo permettere che un metodo possa trattare generic dello stesso tipo istanziati con parametri diversi, magari solo per contarli o visualizzarli. Per permettere questo, è stata allora introdotta una notazione convenzionale in cui il carattere punto interrogativo (`?`) rappresenta un tipo qualsiasi. La seguente sintassi è perciò supportata:

```
Vettore<?> v1 = new Vettore<String>();
Vettore<?> v2 = new Vettore<Object>();
```

Il carattere `?` usato in questo contesto è detto *carattere jolly* (*wildcard*); ci si riferisce alle variabili `v1` e `v2` dell'esempio col termine di *Vettore di sconosciuti* (*Unknown*).

Il carattere jolly può essere usato nelle dichiarazioni, ma non nella creazione di istanze; la seguente sintassi infatti non viene compilata:

```
Vettore<?> v1 = new Vettore<?>(); // Errore in compilazione!
```

Per essere sicuri che non si possano generare errori di tipo, i metodi invocabili su un generic di sconosciuti sono limitati a quelli che non possono causare errori in alcun caso; quelli cioè che non hanno alcun tipo parametrico tra argomenti. Facendo riferimento alle dichiarazioni precedenti, le seguenti istruzioni sono permesse:

```
int n = v1.lunghezza();
System.out.println (v1.prendi(0));
```

La prima non tratta alcun tipo parametrico, mentre la seconda ne restituisce uno che però possiamo trattare con sicurezza fintanto che assumiamo che sia semplicemente un'istanza di `Object`.

Le seguenti operazioni invece non sono permesse per non rischiare di corrompere la coerenza delle istanze:

```
v1.metti("A");              // Errore in compilazione!
v2.metti(new Object());     // Errore in compilazione!
```

10.2.4 Limiti superiori

Abbiamo visto che se dichiariamo un generic con un carattere jolly, da esso possiamo aspettarci solo `Object` in sostituzione del tipo parametrico sconosciuto. Può capitare però che si sappia per certo che i tipi parametrici derivino da una qualche classe; in tal caso può diventare necessario invocare i metodi almeno di questa classe. Per capire bene questo punto, supponiamo di dover gestire gli articoli di un ferramenta: in generale essi avranno le caratteristiche più disparate ma anche alcuni tratti comuni come, per esempio, il fatto di avere un prezzo. Alcune classi del nostro progetto potrebbero essere le seguenti:

```
public abstract class Articolo {
   public abstract double costo();
   // ...
}

class Cavo extends Articolo {
   private double lunghezzaInMetri;
   private double costoAlMetro;

   public Cavo (double lunghezzaInMetri, double costoAlMetro) {
      this.lunghezzaInMetri = lunghezzaInMetri;
      this.costoAlMetro = costoAlMetro;
   }
   public double costo() {
      return lunghezzaInMetri * costoAlMetro;
   }
   // ...
}

class Martello extends Articolo {
   private int numeroPezzi;
   private double costoAlPezzo;
   public Martello (int numeroPezzi, double costoAlPezzo) {
      this.numeroPezzi = numeroPezzi;
      this.costoAlPezzo = costoAlPezzo;
   }
   public double costo() {
      return numeroPezzi * costoAlPezzo;
   }
   // ...
}
```

Supponiamo ora di voler scrivere un metodo `calcolaCosto` in grado di calcolare il costo di tutti gli articoli contenuti in un oggetto `Vettore` omogeneo, sia che esso contenga istanze di oggetti `Martello` che di oggetti `Cavo`. Per far questo le sintassi dei generic viste finora non sono sufficienti, perché, per quanto detto, se dichiarassimo il metodo nel modo seguente:

```
double calcolaCosto (Vettore<Articolo> v)
```

esso non potrebbe essere invocato con oggetti istanziati come `Vettore<Martello>` o `Vettore<Cavo>`; mentre se lo dichiarassimo nel modo seguente:

```
double calcolaCosto (Vettore<?> v)
```

saremmo costretti a usare un cast per invocare il metodo `costo` degli oggetti contenuti. Occorre quindi un nuovo formalismo che risolva questo problema e che è il seguente:

```
double calcolaCosto (Vettore<? extends Articolo> v)
```

In questo modo dichiariamo che il `Vettore` ricevuto come argomento contiene oggetti derivanti dalla classe `Articolo`: questo permette di poter invocare i metodi appartenenti a questa classe, anche se comunque impedisce di poter aggiungere oggetti di qualsiasi tipo al `Vettore`. In questo caso si dice che `Articolo` è il *limite superiore (upper bound)* del carattere jolly.

Il programma seguente mette in pratica quanto detto:

```
public class TestArticolo {
   public static void main (String argv[]) {
      Vettore<Martello> vm = new Vettore<Martello>();
      vm.metti (new Martello (10, 2.50), 0);
      vm.metti (new Martello (15, 3.50), 1);
      vm.metti (new Martello (15, 5.00), 2);
      System.out.println (calcolaCosto(vm));

      Vettore<Cavo> vc = new Vettore<Cavo>();
      vc.metti (new Cavo (100,  0.50), 0);
      vc.metti (new Cavo (150,  0.75), 1);
      System.out.println (calcolaCosto(vc));

   }
   public static double calcolaCosto (Vettore<? extends
                                                Articolo> v) {
      double risultato = 0;
      for (int i = 0; i < v.lunghezza(); i++)
         risultato += v.prendi (i).costo();
      return risultato;
   }
}
```

10.2.5 Metodi generici

Per quanto abbiamo visto finora, non è possibile aggiungere elementi a un oggetto della classe `Vettore` a meno di non conoscere esattamente il tipo di parametro con cui è istanziato. Naturalmente questo è corretto perché evita di corrompere la congruenza della collezione, ma in alcuni casi può risultare d'impaccio. Supponiamo di

voler riempire un oggetto della classe `Vettore`, istanziato con un qualsiasi parametro, col contenuto di un array monodimensionale di tipo opportuno. Per far questo ci vengono in aiuto i *metodi generici (generic methods)*, come il seguente:

```
public static <T> void riempiVettoreDaArray (T[] a,
                                              Vettore<T> v) {
   for (int i = 0; i < a.length; i++)
      v.metti (a[i], i);
}
```

Si dice generico un metodo nel quale si dichiarano tra parentesi angolari uno o più tipi parametrici di uno o più argomenti ed eventualmente del valore restituito. Metodi di questo tipo possono appartenere anche a classi o interfacce non generiche e, a differenza delle classi, per essere invocati non necessitano di una dichiarazione esplicita dei tipi parametrici che devono trattare poiché il compilatore li inferisce automaticamente dagli argomenti attuali. Per esempio, il seguente brano di codice è perfettamente legale e riempie l'oggetto `Vettore` con il contenuto di un array.

```
Vettore<Martello> vm = new Vettore<Martello>();
Martello ma[] = new Martello[3];
ma[0] = new Martello (10, 2.50);
ma[1] = new Martello (15, 3.50);
ma[2] = new Martello (15, 5.00);
riempiVettoreDaArray(ma, vm);
```

10.2.6 Limiti inferiori

Supponiamo ora di voler scrivere un metodo che copi un vettore in un altro e restituisca l'ultimo elemento copiato.

```
public static <T> T copia(Vettore<T> da, Vettore<T> a) {
   T ultimo = null;
   for (int i = 0; i < da.lunghezza(); i++)
      a.metti (ultimo = da.prendi(i), i);
   return ultimo;
}
```

A questo metodo è necessario fornire due oggetti `Vettore` parametrizzati con la stessa classe, ma questo risulta essere in generale troppo restrittivo. Per capire perché, supponiamo di avere i seguenti oggetti:

```
Vettore<String> vs = new Vettore<String>();
Vettore<Object> vo = new Vettore<Object>();
```

Abbiamo visto che l'assegnamento `vo = vs` è scorretto, ma non è scorretto assegnare gli elementi di `vs` a `vo`, in quanto un oggetto di tipo `String` è anche un oggetto di tipo `Object`. Quindi se permettiamo al metodo `copia` di copiare un oggetto `Vettore` parametrizzato con `String` in oggetto `Vettore` parametrizzato con `Object`, ampliamo le possibilità d'uso del metodo senza compromettere la congruenza dei

generic coinvolti. Più in generale, il metodo funziona correttamente quando l'oggetto `Vettore` di destinazione della copia è parametrizzato con una classe con cui è parametrizzato l'oggetto `Vettore` di partenza oppure con una sua superclasse. Questa condizione si esprime con l'uso un carattere jolly limitato questa volta inferiormente, poiché ci va bene una certa classe oppure una sua superclasse. Il limite inferiore *(lower bound)* si nota in modo analogo al limite superiore, sostituendo però la parola `extends` con la parola `super`. Il metodo diventa allora:

```
public static <T> T copia(Vettore<T> da, Vettore<? super T> a) {
   T ultimo = null;
   for (int i = 0; i < da.lunghezza(); i++)
      a.metti (ultimo = da.prendi(i), i);
   return ultimo;
}
```

Diventa allora legale la seguente invocazione:

```
Vettore<String> vs = new Vettore<String>();
Vettore<Object> vo = new Vettore<Object>();
String s = copia (vs, vo);
```

Va notato che questa maggiore libertà d'uso del metodo è possibile ottenerla anche usando un limite superiore: la seguente versione del metodo `copia` è quasi equivalente a quella appena vista:

```
public static <T> T copia(Vettore<? extends T> da, Vettore<T> a) {
   T ultimo = null;
   for (int i = 0; i < da.lunghezza(); i++)
      a.metti (ultimo = da.prendi(i), i);
   return ultimo;
}
```

La differenza tra le due risiede nel fatto che la prima prende come tipo parametrico di riferimento quello passato nel primo argomento, mentre la seconda usa quello del secondo argomento. In questo esempio la differenza è evidenziata dal tipo del valore restituito e difatti, ricordando l'esempio precedente, l'istruzione

```
String s = copia (vs, vo); // Errore con la seconda versione!
```

non viene compilata, poiché in questo caso il metodo restituisce un oggetto di tipo `Object`.

10.3 Package

Uno degli obiettivi che sono stati perseguiti nella progettazione di Java è quello di rendere possibile la programmazione per *componenti*. Questo approccio ha lo scopo di consentire la costruzione dei programmi mediante l'assemblaggio di parti prefabbricate, in modo analogo a come si costruisce un computer mettendo insieme dei cir-

cuiti integrati, anziché riprogettare ogni volta tutte le singole parti. Un problema nell'adattare questo approccio allo sviluppo del software, consiste nel fatto che assemblando parti sviluppate da fornitori diversi, diventa molto probabile avere delle collisioni sui nomi. Per questo motivo è stato introdotto il concetto del *name space* (spazio dei nomi). Il principio è abbastanza semplice: ogni classe è inserita in *package* così come, per esempio, ogni file di un sistema UNIX/LINUX, *Windows* o Macintosh appartiene a una directory. In tal modo non c'è nessun problema ad avere classi con nomi uguali, a patto che risiedano in package diversi. Gli esempi che abbiamo visto funzionano perché le classi illustrate, non avendo nessuna indicazione sul package di appartenenza, vengono poste nel package di default che è l'unico che non ha nome.

I package sono strutturati in modo gerarchico; per esempio, esiste il package `java` che ha dei sottopackage, come `java.lang`, `java.util`, ecc. Non ci sono limiti al numero di livelli che si possono generare. Poiché le classi hanno lo stesso nome dei file in cui sono contenute, è necessario evitare anche la collisione dei nomi dei file. Per questo motivo esiste una corrispondenza puntuale tra la struttura gerarchica dei package e la struttura delle directory in cui risiedono i file `.class`. Ecco quindi che il package `java.lang` risiede nella directory `java/lang` in un sistema operativo UNIX/LINUX, in `java\lang` in un sistema operativo Windows e in `java:lang` in un sistema operativo Macintosh. Per dare la massima flessibilità agli sviluppatori, è possibile avere più directory radice dei package che possono essere impostate in fase di esecuzione. Tipicamente queste directory sono contenute nella variabile d'ambiente `CLASSPATH`, anche se poi dipende molto dal sistema in uso. Supponendo quindi di lavorare sotto UNIX, che utilizza la variabile d'ambiente `CLASSPATH`, se questa contiene `/usr/lib/java`, le classi appartenenti al package `java.lang` vengono cercate nella directory `/usr/lib/java/java/lang`. Volendo avere più directory radice, è sufficiente inserirle in `CLASSPATH` separandole, nel caso UNIX, dal carattere `':'` (due punti). Per esempio, se `CLASSPATH` contiene un valore del tipo `/usr/lib/java:.:/home`, le classi appartenenti al package `java.lang` vengono ricercate prima nella directory `/usr/lib/java/java/lang`, quindi, se non sono state trovate, nella directory `./java/lang`, dove il punto significa la directory corrente, e infine se ancora l'interprete non è riuscito a individuarle, nella directory `/home/java/lang`.

Col tempo il numero di classi disponibili è aumentato in modo esponenziale. Per risparmiare spazio su disco e, al contempo, rendere più veloce il caricamento delle classi, è stata introdotta la possibilità di mettere nella variabile `CLASSPATH` non solo nomi di directory, ma anche nomi di file in formato ZIP. Il formato ZIP consente di memorizzare in un unico file un'intera sottodirectory, conservando tutte le informazioni sul nome dei file, sulla loro locazione e, ovviamente, sul loro contenuto, ma occupando molto meno spazio. Successivamente, per questo stesso scopo, è stato creato il formato JAR (Java ARchive) che è compatibile con il formato ZIP, ma include un file di testo col nome `META-INF/MANIFEST.MF` contenente informazioni sui package contenuti. Nelle più recenti release del JDK tutte le classi di base sono contenute nel file `rt.jar` e normalmente non è necessario includerlo nella variabile `CLASSPATH` in quanto la Java Virtual Machine lo carica sempre per default.

Per inserire una classe in un package, si usa l'istruzione `package` la cui forma generale è la seguente:

```
package nome-pkg1[.nome-pkg2[... .nome-pkgN]];
```

Può essere presente una sola istruzione di questo tipo all'interno di un file `.java` e, nel caso sia presente, deve apparire prima di ogni altra. Un package quindi è un'aggregazione di classi, in modo analogo a come le classi sono aggregazioni di dati e metodi. Il nome del package diventa quindi un *qualificatore* del nome delle classi in modo analogo a come il nome della classe o il nome di un'istanza della classe è un qualificatore dei nomi dei dati e dei metodi. Quando in una classe `C1` si dichiara o comunque si fa riferimento a una classe `C2` senza usare il qualificatore di package, si sottintende che `C1` e `C2` appartengono allo stesso package, a meno che la classe `C2` non sia stata importata con le istruzioni illustrate più avanti. È per questo motivo che abbiamo potuto ignorare il qualificatore di package fino a ora.

Le regole per la formazione dei nomi dei package sono le stesse che abbiamo visto per gli altri nomi; tutti i package forniti con Java sono per convenzione costituiti da sole lettere minuscole, in modo che risulti immediato distinguerli dai nomi delle classi, che invece iniziano sempre con una lettera maiuscola.

Per esempio, possiamo definire un package `ordinamento` in cui inserire la classe `VettoreOrdinato` e l'interfaccia `Ordinabile`. I file `VettoreOrdinato.java` e `Ordinabile.java` che le contengono debbono allora includere la seguente istruzione:

```
package ordinamento;
```

I file così modificati vengono compilati senza errori, ma i problemi sorgono quando si tenta di compilare la classe `Punto`, infatti quest'ultima appartiene ancora al package di default e, poiché non esiste più un'interfaccia in questo package di nome `Ordinabile`, il compilatore segnala l'errore. La classe `Punto` deve essere allora modificata.

```
public class Punto implements ordinamento.Ordinabile
{
   ...
   public boolean maggioreDi (ordinamento.Ordinabile o) {
      if (o instanceof Punto)
         return distanza() > ((Punto) o).distanza();
      else
         return false;
   }
}
```

Si tratta di un sistema molto efficiente per tenere distinte classi con funzionalità differenti e sviluppate da fornitori diversi; ecco perché tutte le classi di base fornite con il linguaggio sono contenute in package. L'utilizzo costante del qualificatore di package può risultare scomodo, in quanto tende ad allungare i nomi e quindi a renderli più

noiosi da scrivere e meno comprensibili a colpo d'occhio. Per questo motivo è possibile dichiarare all'inizio di un file `.java` le classi appartenenti a package diversi che si intendono usare, in modo tale che quando ci si riferisce a tali classi il nome del package sia implicito. Questa operazione è chiamata *importazione* di classi e si ottiene con il comando `import` la cui forma generale è la seguente:

```
import nome-pkg1[.nome-pkg2[... .nome-pkgN]].nome-classe;
```

Si possono usare in uno stesso file un numero qualsiasi di comandi `import`. Nel caso in cui si debbano usare molte classi di uno stesso package, si può usare un * (asterisco) al posto del nome della classe che permette di importare tutte le classi del package. Va notato che importare tutte le classi di un package invece delle sole necessarie, può rallentare la compilazione, ma non influisce né sull'efficienza né sulla dimensione della classe compilata. Ecco dunque come può essere modificato il programma di test precedente per lavorare correttamente con il package `ordinamento`:

```
import ordinamento.*; //oppure import ordinamento.VettoreOrdinato;

class TestOrdinamento3
{
  public static void main (String argv[]) {
    VettoreOrdinato vo = new VettoreOrdinato(10);
    vo.aggiungi (new Punto(30,40));
    vo.aggiungi (new Punto(300,400));
    vo.aggiungi (new Punto(3,4));
    vo.ordina();
    for (int i = 0; i < vo.elementi(); i++)
     System.out.println(((Punto)(vo.leggi (i))).distanza());
  }
}
```

Se si importano due o più package che contengono classi comuni, ci si deve riferire a queste con il nome completo del qualificatore di package, altrimenti si ottiene un errore in compilazione. Anche se una classe importata ha il nome del package implicito, è sempre permesso usare il nome completo del qualificatore di package.

Le classi principali sono incluse nel package `java` che a sua volta è suddiviso in sottopackage. In particolare il package `java.lang`, che contiene le classi necessarie anche per effettuare le elaborazioni più semplici, è sempre importato implicitamente all'interno di una qualsiasi classe. In pratica, in tutti gli esempi visti, è come se fosse stata presente sempre la linea

```
import java.lang.*;
```

il che ha permesso di utilizzare metodi presenti in questo package senza ulteriori qualificatori. Per esempio il nome completo dell'istruzione di visualizzazione, in effetti, è `java.lang.System.out.println`.

10.4 Modificatori

Abbiamo visto che in Java sono presenti dei *modificatori* che alterano appunto il comportamento o la funzionalità di alcuni costrutti. Ora che abbiamo esaminato le parti fondamentali del linguaggio, possiamo dare una panoramica completa su questi modificatori, anche se per qualcuno dovremo ancora rimandare la spiegazione. La Tabella 10.1 riporta tutti i modificatori e gli elementi cui possono essere applicati.

Tabella 10.1

	Classe	Interfaccia	Metodo	Variabile
`public`	×	×	×	×
`protected`	×	×	×	×
`private`	×	×	×	×
`abstract`	×	×	×	
`final`	×		×	×
`static`	×	×	×	×
`native`			×	
`transient`				×
`synchronized`			×	
`volatile`				×
`strictfp`	×	×	×	

10.4.1 Modificatori di accesso

È possibile modificare la visibilità di un dato o di un metodo appartenenti a una classe per mezzo dei modificatori `public`, `protected` e `private`. Questi modificatori di accesso possono essere premessi a una qualsiasi dichiarazione di dato o metodo, il che consente di avere quattro livelli di visibilità, cioè un livello per ogni modificatore più un livello di default se non si usa alcun modificatore. I quattro livelli di visibilità possono essere riassunti come segue:

- `private`: il dato o metodo è visibile solo all'interno della classe in cui è dichiarato;
- nessun modificatore: il dato o metodo è visibile anche da tutte le classi che fanno parte del package;
- `protected`: il dato o metodo è visibile anche al di fuori del package, ma solo dalle classi derivate dalla classe di appartenenza;
- `public`: il dato o metodo è visibile dovunque è visibile il package.

Questi modificatori sono tra loro mutuamente esclusivi, ma possono coesistere con qualsiasi altro modificatore illustrato nel seguito.

Il modificatore `public` può essere premesso anche alla dichiarazione di una classe o di un'interfaccia. Se non specificato, la classe, o interfaccia, nel suo insieme rimane visibile solo all'interno del package, per cui è inutile che abbia dei metodi pubblici.

I metodi di un'interfaccia hanno sempre la stessa visibilità dell'interfaccia stessa. Come si vede dalla Tabella 10.1, anche i modificatori `private` e `protected` possono essere premessi alla dichiarazione di una classe o di un'interfaccia: questo è possibile (e sensato) solo nel caso di *classi membro* e *interfacce membro*, vale a dire di classi e interfacce dichiarate all'interno di altre classi o interfacce. Questo argomento verrà illustrato nel paragrafo successivo.

10.4.2 Modificatori di classe e interfacce

Oltre ai modificatori `public`, `private` e `protected` appena visti, altri quattro tipi di modificatori possono essere applicati a una classe o a un'interfaccia. Il primo, `abstract`, che è stato visto nel capitolo precedente, premesso a una dichiarazione di classe fa in modo che essa non possa essere istanziata direttamente e che possa contenere metodi a loro volta `abstract`, cioè privi di corpo. Questo modificatore si può usare anche con le interfacce, ma essendo queste astratte di principio, in tal caso è superfluo.

Il secondo, `final`, invece indica che una classe è completa e da essa non è permesso derivare sottoclassi. Questo modificatore è stato pensato per fornire un più elevato livello di sicurezza su oggetti delicati che possono provocare problemi seri, ma nella norma non dovrebbe essere usato in quanto è difficile prevedere tutti i possibili utilizzi di una classe. È già successo che alcune classi fornite con il linguaggio, che in un primo tempo erano di tipo `final`, siano state rese derivabili in rilasci successivi. I modificatori `abstract` e `final` sono mutuamente esclusivi. Un'interfaccia non può essere dichiarata `final`.

Il modificatore `static` può essere usato in dichiarazioni di classi e interfacce membro, di cui parleremo in seguito.

Il modificatore `strictfp` modifica il comportamento di tutte le espressioni in virgola mobile contenute nella classe o nell'interfaccia. Java usa i formati in virgola mobile specificati dall'ANSI/IEEE Standard 754-1985. In alcuni casi però può usare internamente un formato a *esponente esteso* (extended-exponent) per garantire la correttezza dei calcoli. In generale ciò non comporta problemi e migliora le prestazioni dei programmi, ma in casi particolari può accadere che un programma si aspetti esattamente il comportamento previsto nello standard IEEE 754, anche se i risultati ottenuti sono meno accurati. In tale caso il modificatore `strictfp` garantisce l'utilizzo nei calcoli del solo formato standard IEEE 754.

10.4.3 Modificatori di metodi

Il modificatore `abstract`, come abbiamo detto, può essere applicato anche a metodi presenti in classi astratte, il che rende possibile dichiarare un metodo senza fornirne l'implementazione. È utilizzabile anche in questo caso, come per le classi, il modifi-

catore `final` che rende impossibile sovrascrivere il metodo da parte di una classe derivata. Non ha molto senso definire un metodo `final` in una classe `final`, anche se il compilatore lo permette.

Il modificatore `native` dà la possibilità di dichiarare un metodo senza corpo in modo analogo al modificatore `abstract`, ma con un significato molto diverso. Tale modificatore indica, infatti, che il corpo del metodo è implementato in un linguaggio diverso da Java, come, per esempio, C, C++, FORTRAN o assembler. I modificatori `abstract` e `native` sono mutuamente esclusivi.

Rimangono altri tre modificatori di cui il primo, `static`, è già stato visto nel capitolo precedente, il secondo, `synchronized`, verrà illustrato in un capitolo successivo dove vengono trattati i *thread*, mentre l'ultimo, `strictfp`, ha significato analogo a quello illustrato nel paragrafo precedente, restringendo però il campo di applicazione a un singolo metodo anziché a un intera classe.

10.4.4 Modificatori di variabili

Delle variabili `static` abbiamo già parlato nel capitolo precedente e ci occuperemo del modificatore `volatile` sempre nel capitolo dedicato ai *thread*.

Il modificatore `final` fa in modo che una variabile di un tipo primitivo o un object-id non sia modificabile dopo l'assegnazione. Si osservi che in questo secondo caso il fatto di rendere non modificabile l'object-id lascia modificabile il contenuto dell'oggetto, istanza di classe o array che sia.

Normalmente le variabili di tipo `final` vengono inizializzate all'atto della definizione, ma è possibile ritardare l'assegnazione a un momento successivo a patto che esso sia unico e avvenga in fase di costruzione dell'istanza (per variabili d'istanza) o della classe (per variabili di classe). Quando una variabile viene dichiarata `final` senza che sia inizializzata, si parla di variabile *blank final*.

In pratica, se si dichiara `final` una variabile d'istanza, il suo assegnamento deve essere effettuato:

- come un'inizializzazione all'atto della definizione, oppure
- una sola volta nell'inizializzatore d'istanza, oppure
- una sola volta in tutti i costruttori dell'istanza.

Se invece si dichiara `final` una variabile di classe (statica), il suo assegnamento deve essere effettuato:

- come un'inizializzazione all'atto della definizione, oppure
- nell'inizializzatore statico.

L'utilizzo del modificatore `final` è l'unico modo che si ha in Java per dichiarare delle costanti simboliche. Abbiamo già visto che, per esempio, nella classe `Math` esiste la costante seguente con il valore di pi greco:

```
public final static double PI = 3.14159265358979323846;
```

Il modificatore `final` impedisce che un oggetto possa modificare questo valore, creando scompiglio nel programma. Le costanti in generale dovrebbero essere dichiarate `static`, in quanto non ha senso occupare memoria in ogni istanza con un valore che non si può modificare.

L'ultimo modificatore, `transient`, si utilizza per indicare quelle variabili d'istanza il cui valore non è importante per stabilire lo stato dell'oggetto. Verrà trattato nel capitolo dove si trattano gli oggetti persistenti.

10.5 Interfacce e classi annidate

Nelle applicazioni reali capita spesso di avere delle classi e/o interfacce che interagiscono l'una con l'altra in modo molto stretto. Abbiamo visto che i package permettono di raggruppare un insieme di classi volte a risolvere uno stesso tipo di problema. A volte però si può avere una classe (o un'interfaccia) che ha senso solo se vista in relazione a un'altra classe (o interfaccia). Per rendere evidenti nel codice dei programmi situazioni di questo tipo, Java permette di dichiarare classi e interfacce all'interno di altre classi o interfacce. In tali casi si parla di classi e interfacce *annidate* (nested). Per contrasto, classi e interfacce dichiarate esternamente a qualsiasi altra classe o interfaccia sono dette *di massimo livello* (top-level). Desideriamo sottolineare subito che queste caratteristiche non modificano il modello concettuale di programmazione a oggetti visto finora, ma servono solo a migliorare la leggibilità dei programmi. I vantaggi derivanti dall'utilizzo di queste tecniche appariranno evidenti quando affronteremo le interfacce utente grafiche.

10.5.1 Interfacce annidate

Si dice *interfaccia annidata* un'interfaccia la cui dichiarazione viene fatta nel corpo di un'altra. Le interfacce annidate hanno le stesse caratteristiche di quelle top-level, l'unica particolarità è che per riferirsi a esse da fuori dell'interfaccia che le contiene, è necessario premettere il nome di tale interfaccia seguito da un punto. Di fatto, possiamo considerare un'interfaccia annidata come membro di un'altra e per questo motivo si dice anche *interfaccia membro* (member interface).

Per fare un esempio, supponiamo di avere bisogno di un oggetto che si comporti come un array, ma che permetta di memorizzare due oggetti in corrispondenza di ciascun indice. Supponiamo inoltre di voler definire un'interfaccia minima per oggetti di questo tipo, in modo che l'implementazione reale possa evolvere in modi diversi.

I metodi da definire in quest'interfaccia, chiamiamola `ArrayDiCoppie`, sono sostanzialmente due: uno per inserire due oggetti in corrispondenza di un determinato indice e l'altro per ricuperarli. Per la definizione del primo dei due metodi è naturale prevedere tre argomenti, cioè l'indice e i due oggetti:

```
public void mettiA (int indice, Object o1, Object o2);
```

La definizione del secondo metodo, chiamiamolo `prendiA`, presenta invece un problema dato che non è possibile per un metodo restituire più di un oggetto. La soluzione più semplice consiste nel restituire un oggetto che permetta d'accedere ai due oggetti

desiderati. Dato che stiamo definendo un'interfaccia per rendere possibile qualsiasi implementazione, ha senso definire l'oggetto restituito da `prendiA` in termini d'interfaccia. La descrizione completa di `ArrayDiCoppie` potrebbe quindi essere la seguente:

```
public interface ArrayDiCoppie {
   public interface Coppia {
      public Object oggetto1();
      public Object oggetto2();
   }
   public void mettiA (int indice, Object o1, Object o2);
   public Coppia prendiA (int indice);
}
```

Appare chiaro in questo caso che l'interfaccia `Coppia` (o meglio `ArrayDiCoppie.Coppia`) ha significato solo come completamento dell'interfaccia `ArrayDiCoppie`, mentre dichiarare `Coppia` come interfaccia top-level non avrebbe reso evidente questo stretto legame.

Una possibile implementazione di `ArrayDiCoppie` è contenuta nelle seguenti due classi:

```
class Coppia implements ArrayDiCoppie.Coppia {
   Object o1;
   Object o2;
   public Coppia (Object o1, Object o2) {
      this.o1 = o1;
      this.o2 = o2;
   }
   public Object oggetto1 () {
      return o1;
   }
   public Object oggetto2 () {
      return o2;
   }
}

public class CoppieArray implements ArrayDiCoppie {
   int dimensione;
   Coppia array[];

   public CoppieArray(int dimensione) {
      this.dimensione = dimensione;
      array = new Coppia[dimensione];
   }
   public void mettiA (int indice, Object o1, Object o2) {
      array[indice] = new Coppia (o1, o2);
   }
```

```
    public ArrayDiCoppie.Coppia prendiA (int indice) {
        return array[indice];
    }
}
```

Nei metodi che ricuperano le coppie di oggetti non è stato messo alcun controllo che l'indice fornito come argomento abbia un valore tale da non generare errori. Così facendo la classe si comporta in modo del tutto analogo a un normale array. Possiamo provare il funzionamento delle classi scritte con il seguente programma:

```
public class TestArray {
  static void test (ArrayDiCoppie a) {
    a.mettiA(0, "Primo Oggetto1", "Primo Oggetto 2");
    a.mettiA(1, "Secondo Oggetto1", "Secondo Oggetto 2");
    a.mettiA(2, "Terzo Oggetto1", "Terzo Oggetto 2");
    for (int i = 0; i < 3; i++)
      System.out.println ("Elemento " + i + "=" +
           a.prendiA(i).oggetto1() + "," +
           a.prendiA(i).oggetto2());
  }
  public static void main (String argv[]) {
    CoppieArray a = new CoppieArray(3);
    test (a);
  }
}
```

Notare come il metodo statico `test`, che esegue le elaborazioni, sia completamente indipendente dall'implementazione della classe `CoppieArray` e funzioni solo utilizzando la definizione dell'interfaccia `ArrayDiCoppie`.

Un'interfaccia annidata è sempre implicitamente `public` e quindi è visibile quanto l'interfaccia top-level che la contiene. Il compilatore segnala un errore se si cerca di dichiarare un'interfaccia annidata di tipo `private` o `protected`.

È ammesso dichiarare un'interfaccia all'interno di una classe in modo del tutto analogo a quanto appena visto. In tal caso l'interfaccia annidata è ovviamente sempre visibile dai membri della classe, ma possiamo usare i modificatori `private` o `protected` per cambiare la sua visibilità dalle altre classi.

10.5.2 Classi annidate

Si dice *classe annidata* una classe la cui dichiarazione compare nel corpo di un'altra classe o interfaccia. Esistono diversi tipi di classi annidate, e precisamente:

- classi interne (inner classes);
- classi annidate non interne (non-inner classes);
- classi locali (local classes);
- classi anonime (anonymous classes).

Una classe interna è una classe annidata dichiarata al di fuori dei metodi della classe che la contiene. La particolarità di questo tipo di classe è che essa può essere istanziata solo all'interno di un'istanza della classe che la contiene (*istanza contenitrice* o *enclosing instance*) e, in cambio, le sue istanze possono accedere in modo trasparente a tutti i membri dell'istanza contenitrice.

La classe `CoppieArray` vista precedentemente può allora essere modificata nel modo seguente:

```
public class CoppieArray2 implements ArrayDiCoppie {
  int dimensione;
  class Coppia implements ArrayDiCoppie.Coppia {
    Object o1;
    Object o2;
    public Coppia (Object o1, Object o2) {
       this.o1 = o1;
       this.o2 = o2;
    }
    public Object oggetto1 () {
       return o1;
    }
    public Object oggetto2 () {
       return o2;
    }
    public void info () {
      System.out.println ("Dimensione array=" + dimensione);
    }
  }
  Coppia array[];

  public CoppieArray2(int dimensione) {
    this.dimensione = dimensione;
    array = new Coppia[dimensione];
  }
  public void mettiA (int indice, Object o1, Object o2) {
    array[indice] = new Coppia (o1, o2);
  }
  public ArrayDiCoppie.Coppia prendiA (int indice) {
    return array[indice];
  }
}
```

Notare che nella classe `Coppia` abbiamo aggiunto il metodo `info()` per mostrare come sia possibile accedere alla variabile `dimensione` che esiste solo nell'istanza della classe contenitrice. In caso di ambiguità di nomi tra membri della classe interna e quelli della classe contenitrice, è possibile far riferimento al `this` dell'istanza contenitrice premettendo a tale parola riservata il nome della classe contenitrice. Per

esempio la visualizzazione del metodo `info()` potrebbe essere scritta nel modo seguente:

```
System.out.println ("Dimensione array=" +
      CoppieArray2.this.dimensione);
```

Abbiamo detto che una classe interna può essere istanziata solo dall'interno di un'istanza della classe contenitrice, ma questo non vuol dire che le sue istanze non possano essere viste all'esterno della classe contenitrice. Supponiamo di scrivere il seguente brano di codice all'esterno della classe `CoppieArray2`:

```
CoppieArray2.Coppia c;
c = new CoppieArray2.Coppia(null, null); // Errata !!
```

La prima riga è assolutamente legale, mentre la seconda provoca un errore in fase di compilazione. Per renderla corretta è necessario fornire l'istanza contenitrice all'atto della creazione; questo può essere fatto nel modo seguente:

```
CoppieArray2 a = new CoppieArray2(3);
CoppieArray2.Coppia c;
c = a.new Coppia(null, null); // Corretta
```

Da quanto detto si capisce che, stante il fatto che una classe interna risulta sempre come `public` per la classe che la contiene, si possa limitarne la visibilità all'esterno usando i soliti modificatori di accesso. Per le classi interne è ammesso anche il modificatore `private` e in questo caso la classe interna sarà visibile solo dalla classe contenitrice.

Le classi annidate non interne si dichiarano nello stesso modo delle classi interne, ma la loro dichiarazione deve essere preceduta dal modificatore `static`. Questo fa in modo che le istanze possano essere create indipendentemente da qualsiasi istanza della classe contenitrice. Di fatto, sono del tutto equivalenti alle classi top-level, in modo analogo a come le interfacce annidate sono equivalenti alle interfacce top-level.

Nella classe `ProvaArray2` vista prima possiamo dichiarare `static` la classe annidata `Coppia` alla sola condizione di eliminare il metodo `info()`; in questo caso l'espressione

```
c = new CoppieArray2.Coppia(null, null); // Corretta
```

diventa corretta anche al di fuori della classe contenitrice.

Classi interne e classi annidate non interne vengono anche dette *classi membro* (member classes). Il linguaggio Java permette di dichiarare una classe anche all'interno di un metodo. Una classe dichiarata in questo modo è visibile solo all'interno del metodo in cui è dichiarata e per questo motivo viene detta *classe locale* (local class). Una classe locale dichiarata in un metodo d'istanza ha le stesse proprietà di una classe interna con, in più, la possibilità di accedere anche alle variabili locali di tipo `final` del metodo. Analogamente, una classe locale dichiarata in un metodo statico ha le stesse proprietà di una classe annidata non interna e anche in questo caso può accedere alle variabili locali di tipo `final` del metodo.

Per fare un esempio, possiamo modificare la classe `TestArray` usata precedentemente in modo che l'inserzione e il ricupero di elementi nell'array di coppie di oggetti sia controllato in modo da non causare errori in fase di esecuzione:

```
public class TestArray {
  private interface Pippo {
    public boolean ordinable();
  }
  static void test (ArrayDiCoppie a) {
    a.mettiA(0, "Primo Oggetto1", "Primo Oggetto 2");
    a.mettiA(1, "Secondo Oggetto1", "Secondo Oggetto 2");
    a.mettiA(2, "Terzo Oggetto1", "Terzo Oggetto 2");
    for (int i = 0; i < 3; i++)
      System.out.println ("Elemento " + i + "=" +
           a.prendiA(i).oggetto1() + "," +
           a.prendiA(i).oggetto2());
  }
  public static void main (String argv[]) {
    final int dim = 3;
    class MieCoppieArray extends CoppieArray {
      MieCoppieArray () {
        super(dim);
      }
      public void mettiA (int indice, Object o1, Object o2) {
        if (indice >= 0 && indice < dimensione)
          super.mettiA(indice, o1, o2);
      }
      public ArrayDiCoppie.Coppia prendiA (int indice) {
        if (indice >= 0 && indice < dimensione)
          return super.prendiA(indice);
        else
          return null;
      }
    };
    MieCoppieArray a = new MieCoppieArray();
    test (a);
  }
}
```

La classe `MieCoppieArray` ha un costruttore senza argomenti che inizializza la sua superclasse basandosi su una variabile `final` del metodo in cui è dichiarata. La sua dichiarazione termina con un punto e virgola come qualsiasi altra dichiarazione all'interno di un metodo. Il fatto che la nuova classe sia totalmente invisibile dal metodo `test` non crea alcuna difficoltà, in quanto quest'ultimo ha per argomento l'interfaccia implementata, implicitamente per ereditarietà, da `MieCoppieArray`.

In quest'esempio il nome della classe locale è usato una sola volta. Per non obbligare i programmatori a dover pensare dei nomi inutilmente, Java permette la creazione di un particolare tipo di classe locale, detto *classe anonima* (anonymous class). Una classe anonima viene definita e istanziata nello stesso momento. La sintassi Java prevede, infatti, che dopo un'invocazione di una new su una qualsiasi classe sia possibile mettere un blocco con delle dichiarazioni che la specializzano. Le classi anonime, non avendo nome, non possono avere neppure costruttori propri ma al massimo un inizializzatore d'istanza e/o di classe. Per istanziare una classe anonima quindi è necessario invocare un costruttore esistente della classe da cui deriva. Vediamo come può diventare il metodo `main` della classe `TestArray` con l'utilizzo di una classe anonima:

```
public static void main (String argv[]) {
  final int dim = 3;
  CoppieArray a = new CoppieArray(dim) {
    public void mettiA (int indice, Object o1, Object o2) {
      if (indice >= 0 && indice < dimensione)
         super.mettiA(indice, o1, o2);
    }
    public ArrayDiCoppie.Coppia prendiA (int indice) {
      if (indice >= 0 && indice < dimensione)
        return super.prendiA(indice);
      else
        return null;
    }
  };
  test (a);
}
```

Da notare che è stato necessario usare di nuovo l'unico costruttore di `CoppieArray`.

✔ **NOTA**

Le classi e le interfacce annidate sono implementate al momento attuale creando delle classi fittizie, i cui nomi vengono creati automaticamente dal compilatore basandosi sui nomi delle classi coinvolte e su dei numeri progressivi, usando il carattere dollaro ($) per effettuare le concatenazioni. Per questo motivo è opportuno non usare questo carattere nei nomi delle proprie classi.

Domande di verifica

1. Dare una definizione del termine "interfaccia" nella programmazione orientata agli oggetti. Esporre alcuni esempi di interfacce.
2. Che cosa significa "una classe implementa una o più interfacce"?
3. Esporre il concetto di classe *container* (o *collection*).
4. Come si dichiarano classi e interfacce generiche e perché vi si fa riferimento nei parametri formali a *nomi fittizi*?

5. Quali restrizioni si devono rispettare nell'assegnamento di un'istanza a una variabile generic?
6. A quale scopo si utilizza il carattere punto interrogativo (`?`) quando si lavora con generic? Può essere utilizzato nelle dichiarazioni e anche nelle creazione di istanze?
7. Cosa si intende per limite superiore del carattere jolly? E inferiore?
8. Quale utilità hanno i metodi generici? Come si riconoscono?
9. Qual è la funzione dei package nella programmazione per componenti e come sono strutturati? Cosa si intende e qual è l'utilità del *name space*?
10. Quali sono e a cosa servono i modificatori di accesso, di classe, di interfaccia, di variabile?
11. Elencare tutti i modificatori e per ognuno di essi indicare se può essere usato a livello di classe, interfaccia, metodo o variabile.
12. Quale utilità hanno le interfacce e le classi annidate?

Esercizi

1. Modificare `TestOrdinamento` in modo che le ascisse e le ordinate dei tre punti siano richieste all'utente.
2. Implementare una classe `generic` in grado di contenere coppie di oggetti qualsiasi.
3. Modificare la classe `Tempo` del capitolo precedente in modo tale che implementi l'interfaccia `Ordinabile` e usare la classe `VettoreOrdinato` presentata in questo capitolo per ordinarne un certo numero di istanze.
4. Creare un package `prova.pkg1` e dopo aver inserito la classe `Tempo` in tale package, ripetere l'esercizio precedente.
5. Creare un package `prova.pkg2` e dopo aver inserito la classe `Tempo` in tale package, scrivere una classe che utilizzi sia `prova.pkg1.Tempo` che `prova.pkg2.Tempo`.
6. Verificare la differenza di visibilità tra metodi e variabili con visibilità di default e `protected` di classi appartenenti a uno stesso package e classi appartenenti a package diversi.
7. Supponiamo di avere diversi oggetti che stanno su una scrivania, per esempio forbici, penne matite, gomme da cancellare e un portapenne. Scrivere una classe che rappresenti ciascun oggetto e scrivere un metodo che calcoli il peso del portapenne qualora contenga tutti gli oggetti sulla scrivania. L'esercizio deve essere fatto in modo che aggiungendo un nuovo tipo di oggetti, per esempio i righelli, il calcolo del peso funzioni ancora senza dover modificare il codice già scritto. Per ottenere il risultato avvalersi prima dell'ereditarietà e quindi delle interfacce. Evidenziare vantaggi e svantaggi dei due metodi.
8. Rivedere le classi usate fino a questo capitolo, cercando possibili utilizzi di classi e interfacce annidate.

Capitolo 11

Classi standard

Obiettivi didattici

- Metaclasse `Class`
- Classe `Object`
- Gestione delle stringhe (costruttori, confronti, estrazione di caratteri, conversioni, altri metodi)
- Collezioni (`Vector`, iteratori, `for-each`)
- Classi involucro (`Number`, numeri interi, numeri in virgola mobile, `BigDecimal`, `BigInteger`, *autoboxing*)
- Funzioni matematiche
- Enumerali

Come abbiamo visto le classi, che rappresentano il mezzo grazie al quale il progettista software realizza i programmi, risiedono in package che le raggruppano in base a criteri di affinità funzionale. Molte classi vengono comunque fornite con il linguaggio per la risoluzione dei problemi più comuni o per l'interfacciamento con il sistema operativo. Java è corredato da un numero notevole di classi; nella distribuzione del JDK 1.6 (Java 6), per esempio, sono presenti oltre 200 package tra i quali menzioniamo i principali in ordine alfabetico:

`java.applet`	Applet.
`java.awt`	Interfaccia grafica (Abstract Windows Toolkit).
`java.beans`	Classi legate allo sviluppo di Java Beans.
`java.beans.beancontext`	Classi legate allo sviluppo di Java Beans.

`java.io`	Classi per l'input-output.
`java.lang`	Classi fondamentali per il linguaggio.
`java.math`	Classi per calcoli di numeri interi e decimali in qualsiasi precisione.
`java.net`	Classi per applicazioni di rete.
`java.rmi`	Classi per RMI (Remote Method Invocation).
`java.security`	Gestione della sicurezza.
`java.sql`	Accesso ai RDBMS.
`java.text`	Classi per gestire testi, date, numeri e messaggi.
`java.util`	Classi per funzioni di utilità generali.
`javax.accessibility`	Definisce un contratto tra componenti dell'interfaccia utente e una tecnologia che fornisce accesso a quei componenti.
`javax.swing`	Interfaccia grafica scritta totalmente in Java e indipendente dal sistema operativo sottostante.
`org.omg.CORBA`	Mappatura delle API OMG CORBA.

In totale sono presenti oltre 7500 tra interfacce e classi. L'illustrazione di tutte queste funzionalità, oltre a richiedere una pubblicazione a volumi con aggiornamenti mensili, va oltre le finalità del libro per cui vedremo alcune tra le classi più importanti al solo scopo di rendere possibile lo sviluppo di programmi completi. La documentazione è comunque disponibile con i rilasci del linguaggio in formato elettronico (tipicamente in formato HTML), e anzi è opportuno abituarsi a consultare questo tipo di documentazione per avere informazioni allineate con le ultime release e con eventuali nuovi package.

In questo capitolo vedremo inoltre alcuni costrutti del linguaggio che non è stato possibile illustrare precedentemente perché implicano un uso più o meno mascherato di oggetti. L'uso di questi costrutti non è assolutamente necessario, ma hanno lo scopo di semplificare la scrittura dei programmi e di renderli più facilmente leggibili.

✔ NOTA

Nel corso dei vari rilasci del linguaggio, sono stati fatti numerosi cambiamenti nelle classi di corredo, sia perché sono state trovate soluzioni migliori per determinati problemi, sia per adattarsi meglio alle continue evoluzioni del mondo informatico. Esiste quindi un certo numero di classi e metodi che sono stati sostituiti da altri più adatti a svolgere il proprio compito. Le classi e i metodi 'antiquati' non sono stati eliminati dalle distribuzioni per consentire la compatibilità dei programmi già sviluppati, ma il loro uso viene deprecato. Per fare in modo che i programmatori non utilizzino più classi e metodi deprecati, il compilatore segnala con un messaggio d'avvertimento quando una classe ne fa uso. È eventualmente possibile avere notizie più precise su quali sono le classi e/o i metodi mediante opzioni di compilazione.

Il package più importante è `java.lang` che contiene anche alcune classi intrinsecamente legate al linguaggio, senza le quali nessun programma potrebbe funzionare. In particolare, la classe `Object` è la radice di tutte le classi e di tutti gli array, per cui i suoi metodi sono disponibili in tutti gli oggetti. In questo capitolo vedremo rapidamente alcune classi, principalmente del package `java.lang`, che ci servono per illustrare più chiaramente le caratteristiche di Java che ancora rimangono da esaminare. A una prima lettura, alcuni argomenti possono risultare un po' ostici, ma non è il caso di preoccuparsi e conviene comunque proseguire la lettura riservandosi una seconda rilettura e/o consultazione in un momento successivo.

11.1 La classe `Class`

Le istanze di questa classe, appartenente al package `java.lang`, descrivono le classi, le interfacce e i tipi di array in modo tale che un programma Java possa gestirle in fase di esecuzione. Potrebbe quindi essere definita una *metaclasse* in quanto si tratta di una classe che descrive le altre classi. Non esiste un costruttore pubblico in `Class` poiché le istanze possono essere create solo dall'interprete. Ogni volta che viene caricata una nuova classe, una nuova interfaccia oppure viene creato un array differente per tipo e/o dimensioni da qualsiasi altro già esistente, la Virtual Machine crea un'istanza della classe `Class` con le informazioni appropriate. Un programma utente non può quindi creare direttamente nuove istanze di `Class`, ma può ricuperarle e leggerne il contenuto. Esaminiamo dunque alcuni metodi.

- `public static Class forName(String name)`: restituisce l'oggetto di tipo `Class` associato alla classe o all'interfaccia il cui nome corrisponde a quello specificato come argomento. `name` deve essere completo del qualificatore di package se la classe, o interfaccia, specificata non appartiene al package di default. Se non esiste alcuna classe o interfaccia col nome specificato, questo metodo dà un errore o meglio *getta un'eccezione*, come vedremo in seguito.
- `public Object newInstance()`: crea un'istanza della classe rappresentata dall'oggetto corrente come se fosse istanziata tramite un'invocazione di `new` senza argomenti. Questo metodo può gettare diversi tipi di eccezione nel caso in cui la creazione dell'istanza non possa essere eseguita, per esempio se l'istanza corrente rappresenta un'interfaccia, una classe astratta, una classe priva di costruttore senza argomenti ecc.
- `public String getName()`: restituisce il nome della classe, completo del qualificatore di package. Se l'oggetto descrive un'array, la stringa restituita inizierà con un numero di parentesi quadre aperte `[` pari al numero di dimensioni dell'array e sarà seguita da una lettera indicante il tipo dell'array secondo la Tabella 11.1.
- `public boolean isInterface()`: restituisce `true` se l'oggetto descrive una interfaccia, `false` altrimenti.
- `public boolean isArray()`: restituisce `true` se l'oggetto descrive un array, `false` altrimenti.

- `public Class getSuperclass()`: restituisce un oggetto con la descrizione della classe genitrice. Nel caso tale metodo sia applicato a un oggetto che descrive un'interfaccia, un tipo primitivo oppure la classe `Object`, viene restituito il valore `null`. Se invece viene applicato a un oggetto che descrive un array, viene restituito l'oggetto che descrive la classe `Object`.

Tabella 11.1 Codifica del tipo di array in `getName()`

Carattere	Tipo
B	byte
C	char
D	double
F	float
I	int
J	long
L*nome-della-classe*	Classe o interfaccia
S	short
Z	boolean

Dalla versione 1.1 di Java è stato aggiunto il package `java.lang.reflect` che, insieme a `Class`, permette non solo di poter conoscere dettagliatamente in fase di esecuzione tutte le caratteristiche di qualsiasi classe e oggetto, ma anche di poterne invocare qualsiasi costruttore, leggerne le variabili statiche e d'istanza e invocare i suoi metodi. Questo tipo di funzionalità, detta *riflessione* (reflection), è stata aggiunta per permettere la creazione di strumenti di sviluppo scritti direttamente in Java, come debugger, ispettori di classi e oggetti ecc., per permettere la creazione di oggetti permanenti, di cui parleremo in seguito, e per la creazione dei cosiddetti JavaBean, che sono dei componenti riusabili da ambienti di sviluppo grafici senza bisogno di scrittura di codice.

11.2 La classe `Object`

La classe `Object`, inclusa nel package `java.lang`, è la radice di qualsiasi altra classe, compresa la classe `Class`. Se una classe non dichiara esplicitamente la propria discendenza da alcuna classe, è implicitamente assunto che discenda da `Object`. Tutti gli oggetti, compresi gli array, implementano i metodi della classe `Object`. Vediamo alcuni metodi della classe.

- `public final Class getClass()`: restituisce un oggetto con la descrizione della classe a cui appartiene l'oggetto. A tale oggetto sono applicabili i metodi visti nel paragrafo precedente.

- `public boolean equals(Object obj)`: abbiamo visto che se confrontiamo due oggetti con l'operatore di uguaglianza ==, vengono confrontate le referenze e non il contenuto dell'oggetto. Questo metodo serve invece per confrontare il contenuto e deve essere ridefinito in tutte quelle classi i cui oggetti possono essere confrontati fra loro. Questo metodo deve avere le stesse proprietà dell'operatore == e cioè *riflessività* (`x.equals(x)` dà sempre `true`), *simmetria* (`x.equals(y)` dà sempre lo stesso risultato di `y.equals(x)`), *transitività* (se `x.equals(y)` e `y.equals(z)` restituiscono `true`, allora anche `x.equals(z)` restituisce `true`) e *consistenza* (`x.equals(y)` dà sempre lo stesso risultato fino a che `x` e/o `y` non vengono modificate). Inoltre se viene confrontato un oggetto non `null` con un oggetto `null`, il risultato deve essere `false`. Nel caso che questo metodo non sia ridefinito nelle sottoclassi, viene eseguito il metodo definito nella classe `Object` che effettua un semplice confronto sulle referenze per mezzo dell'operatore ==. Poiché gli array non hanno ridefinito questo metodo, la sua invocazione su di essi corrisponde al confronto delle referenze e non del contenuto.
- `public int hashCode()`: restituisce un intero che rappresenta il codice *hash* del contenuto dell'oggetto. Due oggetti che confrontati col metodo `equals` risultano uguali debbono restituire lo stesso intero all'interno della stessa esecuzione, ma non è garantito che due oggetti diversi restituiscano interi diversi. Poiché due array con lo stesso contenuto possono non risultare uguali se confrontati con il metodo `equals`, allora anche i loro codici hash possono differire.
- `public String toString()`: restituisce una stringa contenente una "rappresentazione testuale" dell'oggetto. Anche questo metodo deve essere ridefinito in modo da fornire una rappresentazione significativa dell'oggetto. L'implementazione definita nella classe `Object` restituisce una stringa contenente il nome della classe d'appartenenza seguito da una 'a' commerciale `@` (at) e dalla rappresentazione esadecimale del codice hash dell'oggetto. Va sottolineato che questo metodo viene richiamato dai metodi a noi ben noti `print` e `println` quando l'argomento è un oggetto.
- `protected Object clone()`: così come l'operatore == confronta le referenze e non il contenuto dell'oggetto, l'operatore = effettua una copia della referenza e non del contenuto. Questo metodo serve appunto per creare una copia del contenuto, un *clone*, e può essere ridefinito all'interno delle classi. Quando una classe derivata ridefinisce questo metodo, esso *dovrebbe* agire in modo tale che, dato un oggetto `x` della classe `C` e data l'istruzione `C y=(C)x.clone()`, le espressioni `y!=x`, `y.getClass()==x.getClass()` e `y.equals(x)` restituiscano `true`. Va notato che abbiamo dovuto utilizzare un cast per assegnare la copia di `x` a `y` perché questo metodo restituisce sempre un oggetto della classe `Object`. Se il metodo non è ridefinito, allora l'implementazione definita nella classe `Object` verifica per prima cosa se la classe da duplicare implementa l'interfaccia `Cloneable` e se questo non è vero getta un'eccezione. L'interfaccia `Cloneable` non ha metodi e serve solo per dividere l'universo delle classi di un programma in due categorie, quelle che possono essere clonate da quelle che non possono. Se l'oggetto appartiene a una classe che implementa l'interfaccia `Cloneable` allora il metodo crea

una nuova istanza e assegna a ogni elemento della copia il valore dell'originale per mezzo dell'operatore =. Questo significa che gli oggetti contenuti non vengono a loro volta clonati, per cui questo tipo di copia viene detto *copia superficiale* (shallow copy). Quando si ha bisogno che una classe effettui una *copia profonda* (deep copy), vale a dire che vengano clonati anche gli oggetti contenuti (e gli oggetti contenuti negli oggetti contenuti e così via), è necessario o ridefinire il metodo `clone` o, meglio, definirsene un altro, il cui nome potrebbe essere per esempio `deepClone`, che esegua l'operazione richiesta. Questo problema della copia profonda o superficiale non si pone per le variabili di tipo primitivo in quanto esse vengono copiate totalmente. Gli array sono considerati come appartenenti a una classe che implementa l'interfaccia `Cloneable`, per cui a oggetti di questo tipo può essere applicato il metodo `clone`. Una variabile array e un suo clone verificano solo le prime due condizioni riportate precedentemente mentre non soddisfano la terza, congruentemente con quanto detto sul metodo `equals` applicato agli array.

✔ **NOTA**

Se il lettore prova un esempio che coinvolge il metodo `clone`*, otterrà un errore in compilazione in quanto i metodi che possono gettare eccezioni di un certo tipo debbono essere trattati in modo particolare, analogamente a quanto visto per il metodo* `System.in.read()`*. Per fare una prova allora si può usare un'espressione del genere*

```
Tempo t1 = new Tempo;
       try {
          Tempo t2 = (Tempo)t1.clone();
       } catch (Exception e) {
          System.out.println ("Classe non clonabile");
       }
```

Anche in questo caso è stato necessario un cast per effettuare l'assegnamento. Nell'esempio riportato non esiste il problema della copia profonda o superficiale in quanto i tipi primitivi vengono duplicati totalmente.

11.3 La gestione delle stringhe

Una variabile di tipo `char` consente di memorizzare un singolo carattere. I caratteri singoli non hanno un grande utilizzo, molto più spesso i programmi debbono trattare insiemi di più caratteri che formano parole o frasi. Ci si riferisce a tali insiemi come a *stringhe* di caratteri. Java mette a disposizione due classi, incluse nel package `java.lang`, che permettono di trattare le stringhe, e precisamente `String` e `StringBuffer`. La differenza fra le due consiste sostanzialmente nel fatto che il contenuto di un oggetto della prima classe non è modificabile mentre il contenuto di un oggetto della seconda può essere alterato. Entrambe queste classi sono dichiarate `final` per motivi di ottimizzazione, per cui non è possibile derivare nuove sottoclassi da esse.

Il fatto che esistano due classi distinte, non legate neppure da parentele di ereditarietà, atte a svolgere compiti analoghi è dovuto alla stretta integrazione tra la classe `String` e il linguaggio che permette un utilizzo più semplice delle stringhe. Quando scriviamo un'istruzione come la seguente

```
System.out.println ("Essere o non essere");
```

Java considera l'espressione tra i doppi apici, `"Essere o non essere"`, un'istanza della classe `String`. Ora un'espressione di questo tipo è una costante letterale, come abbiamo già detto nel primo capitolo, per cui è naturale che non possa essere modificata. Poiché una costante letterale è un'istanza della classe `String`, a essa possono essere applicati tutti i metodi della classe. Per esempio il metodo `length()` restituisce un intero contenente il numero di caratteri che compongono la stringa: ecco allora che il codice seguente

```
System.out.println ("Essere o non essere".length());
```

è perfettamente legale e visualizza 19.

Una stringa può essere inizializzata in numerosi modi, il che significa che ha numerosi costruttori, tra cui

- `public String()`: costruisce una stringa vuota.
- `public String(String value)`: costruisce una stringa con il contenuto uguale a `value`.
- `public String(StringBuffer buffer)`: costruisce una stringa il cui contenuto è uguale a quello dell'istanza di `StringBuffer buffer`.
- `public String(char[] value)`: costruisce una stringa contenente gli stessi caratteri contenuti in `value`.
- `public String(char[] value, int offset, int count)`: costruisce una stringa contenente `count` caratteri corrispondenti a quelli contenuti in `value` a partire dall'elemento all'indice `offset`.
- `public String(byte[] ascii)`: costruisce una stringa i cui caratteri sono la conversione degli elementi di tipo `byte` contenuti nell'array `ascii`.
- `public String(byte[] ascii, int offset, int count)`: costruisce una stringa contenente `count` caratteri corrispondenti a quelli contenuti in `ascii` a partire dall'elemento all'indice `offset`.

La classe `StringBuffer` non ha tutti questi costruttori, ma avendone uno che ha come argomento un oggetto di tipo `String`, può usufruire di riflesso di quelli di quest'ultima classe. Volendo per esempio creare un oggetto di tipo `StringBuffer` con il contenuto di un'array di caratteri, si può usare la seguente istruzione:

```
char c[] = { 'a', 'b', 'c' };
StringBuffer sb = new StringBuffer (new String (c));
```

Va detto che non capita molto spesso di usare direttamente gli oggetti della classe `StringBuffer`, e che il loro utilizzo più comune avviene in modo nascosto al programmatore, come ci renderemo conto in seguito. Vediamo ora come si possono trattare le stringhe.

11.3.1 Confronti

Come abbiamo detto, non si può in generale confrontare il contenuto di due oggetti mediante l'operatore d'uguaglianza ==. Nella classe `String` è stato ridefinito il metodo `public boolean equals(Object obj)` della classe `Object` che ha quindi l'argomento di tipo `Object` e non `String`. Restituisce `true` se l'argomento è di tipo `String` e se il suo contenuto è uguale a quello dell'istanza corrente, `false` altrimenti.

Esistono inoltre altri due metodi per confrontare stringhe. Il primo, `public boolean equalsIgnoreCase(String str)` esegue un confronto come il metodo `equals`, ma a differenza di quest'ultimo, non considera diversi i caratteri che, pur rappresentando una stessa lettera, sono uno maiuscolo e uno minuscolo. Anche il metodo `public int compareTo(String str)` esegue un confronto come il metodo `equals` con la differenza che restituisce un intero che vale 0 se `str` è uguale all'istanza corrente, un numero positivo se `str` precede in ordine alfabetico l'istanza corrente e un numero negativo se `str` segue in ordine alfabetico l'istanza corrente. Questo metodo permette quindi di mettere in sequenza stringhe diverse in ordine alfabetico e potrebbe essere utilizzato per l'esempio del vettore ordinato visto precedentemente. L'esempio seguente

```
String str1 = "Marco";
String str2 = new String(str1);
System.out.println (str1 == str2);
System.out.println (str1.equals(str2));
System.out.println (str1.equals("marco"));
System.out.println (str1.equalsIgnoreCase("marco"));
System.out.println (str1.compareTo("Andrea"));
System.out.println (str1.compareTo("Marco"));
System.out.println (str1.compareTo("Ugo"));
```

mostra la differenza dei metodi mostrati e produce la seguente visualizzazione:

```
false
true
false
true
12
0
-8
```

✔ **NOTA**

Il fatto che le stringhe non possano essere modificate permette all'interprete di effettuare numerose ottimizzazioni. Una di queste consiste nel fatto di riutilizzare lo stesso spazio di memoria per stringhe diverse con lo stesso contenuto. Per questo motivo può accadere, e con la Virtual Machine usata per il test degli esempi di questo libro accade, che il codice seguente

```
String str1 = "Marco";
String str2 = "Marco";
System.out.println (str1 == str2);
```

visualizzi `true`. *Vedere casi come questo potrebbe indurre a pensare che l'operatore* `==` *si comporti in modo analogo al metodo* `equals` *mentre in realtà non è vero.*

11.3.2 Estrazione di caratteri

Spesso è necessario analizzare il contenuto di una stringa. La classe `String` mette a disposizione molti metodi per effettuare questo genere d'elaborazione. Il metodo più semplice da usare è `public char charAt(int i)` che restituisce l'`i`-esimo carattere contenuto nella stringa. L'argomento di questo metodo può assumere un qualsiasi valore compreso tra 0 e la lunghezza della stringa meno 1, altrimenti si ottiene un'eccezione in fase di esecuzione. Per conoscere la lunghezza della stringa, può essere utilizzato il metodo `public int length()`. Se volessimo quindi scoprire quante volte compare il carattere `X` nella stringa `str`, potremmo usare il codice seguente

```
String str = "dsXdskòXkòXòldkxX";
int conto = 0;
for (int i = 0; i < str.length(); i++)
   if (str.charAt(i) == 'X')
      conto++;
System.out.println (conto);
```

La classe `String` consente anche di estrarre un insieme contiguo di caratteri da una stringa. Ci si riferisce a tale insieme di caratteri solitamente con il termine di *sottostringa*. Per effettuare tale operazione sono disponibili due metodi. Il primo, `public String substring(int start)`, restituisce una nuova stringa che comprende tutti i caratteri a partire da quello in posizione `start` fino alla fine della stringa originaria. Il secondo `public String substring(int start, int end)`, restituisce una nuova stringa che comprende i caratteri a partire da quello in posizione `start` fino a quello in posizione `end-1`.

```
String str = "foresta";
System.out.println (str.substring(2));
System.out.println (str.substring(3, 6));
```

visualizza

```
resta
est
```

L'esempio appena riportato evidenzia come i metodi `substring` non modifichino la stringa originale.

Nel caso sia necessario trovare se in una stringa è presente un certo carattere e, se sì, in che posizione, si può usare il ciclo visto in precedenza, ma la classe `String` mette a disposizione dei metodi che consentono di effettuare questo tipo d'operazione in modo più semplice.

- `public int indexOf(int c)`: restituisce l'intero `i` più piccolo tale che `this.charAt(i)==c` sia vera. Se la stringa non contiene alcun carattere uguale a `c`, il metodo restituisce `-1`.
- `public int indexOf(int c, int start)`: restituisce l'intero `i` più piccolo tale che `i>=start && this.charAt(i)==c` sia vera. Se la stringa non contiene alcun carattere che soddisfi l'espressione precedente, il metodo restituisce `-1`.

Altri due metodi consentono di effettuare lo stesso tipo operazione su sottostringhe invece che su caratteri.

- `public int indexOf(String str)`: restituisce l'intero `i` più piccolo tale che `this.substring(i,i+str.length()).equals(str)` sia vera. Se la stringa non contiene nessuna sottostringa uguale a `str`, il metodo restituisce `-1`.
- `public int indexOf(String str, int start)`: restituisce l'intero `i` più piccolo tale che `i>=start && this.substring(i,i+str.length()).equals(str)` sia vera. Se la stringa non contiene alcun carattere che soddisfi l'espressione precedente, il metodo restituisce `-1`.

I quattro metodi appena visti scandiscono la stringa da sinistra a destra, trovando così gli indici più bassi che soddisfano le condizioni richieste. Esistono quattro metodi analoghi che eseguono le stesse elaborazioni ma scandiscono la stringa da destra a sinistra, trovando in tal modo gli indici più alti che soddisfano le richieste. Questi quattro metodi sono

```
public int lastIndexOf(int c)
public int lastIndexOf(int c,int start)
public int lastIndexOf(String str)
public int lastIndexOf(String str,int start)
```

11.3.3 Conversioni tra stringhe e array

Abbiamo visto alcuni costruttori che inizializzano una stringa partendo da un array di tipo `char` o `byte`. In particolare la possibilità di convertire degli array di `byte` in stringhe e viceversa è molto utile perché i metodi per le operazioni di input/output utilizzano generalmente vettori di questo tipo. I metodi seguenti si occupano della funzione opposta, ossia di trasferire il contenuto di una stringa in un array.

- `public void getChars(int start,int end,char dst[],int dstStart)`: copia il contenuto della stringa a cui è applicato nell'array `dst`. La stringa viene copiata a partire dal carattere in posizione `start` fino al carattere in posizione `end-1`. I caratteri della stringa vanno a riempire gli elementi dell'array

`dst` a partire da quello con indice `dstStart`. L'array deve essere sufficientemente capiente, altrimenti si ottiene un'eccezione.

- `public byte[] getBytes()`: restituisce un array di `byte` corrispondente ai caratteri contenuti nella stringa.
- `public char[] toCharArray()`: restituisce un array di `char` i cui elementi sono i caratteri della stringa. L'array restituito è una copia, per cui modificandone gli elementi non si modifica il contenuto della stringa.

11.3.4 Conversione di tipi in stringhe

Quando si ha la necessità di visualizzare dei messaggi per l'utente, può capitare di dover convertire dei dati non stringa in una stringa. Abbiamo visto che nella classe `Object` c'è un metodo, `toString`, che tutte le classi derivate dovrebbero implementare proprio a questo scopo. Purtroppo i tipi primitivi non sono oggetti, per cui a essi non si può applicare `toString`. Per rimediare a questo esistono i seguenti metodi statici che convertono i tipi primitivi in stringhe.

- `public static String valueOf(boolean b)`: restituisce la stringa `"true"` o la stringa `"false"` a seconda dell'argomento.
- `public static String valueOf(char c)`: restituisce una stringa di lunghezza 1 contenente il carattere `c`.
- `public static String valueOf(int i)`: restituisce una stringa contenente la rappresentazione decimale del numero `i`.
- `public static String valueOf(long l)`: restituisce una stringa contenente la rappresentazione decimale del numero `l`.
- `public static String valueOf(float f)`: restituisce una stringa contenente la rappresentazione decimale del numero `f`. Se il numero è più piccolo di 10^{-3} o è più grande di 10^{7}, la stringa restituita contiene la rappresentazione in formato esponenziale.
- `public static String valueOf(double d)`: restituisce una stringa contenente la rappresentazione decimale del numero `d`. Se il numero è più piccolo di 10^{-3} o è più grande di 10^{7}, la stringa restituita contiene la rappresentazione in formato esponenziale.
- `public static String valueOf(Object obj)`: questo metodo è implementato solo per uniformità, ma di fatto la sua invocazione è equivalente a `obj.toString()`.
- `public static String valueOf(char[] data)`: anche questo metodo è implementato per uniformità e la sua invocazione è equivalente a `new String(data)`.
- `public static String valueOf(char[] data, int offset, int count)`: altro metodo implementato per uniformità e la sua invocazione è equivalente a `new String(data, offset, count)`.

11.3.5 Altri metodi

Riportiamo di seguito altri utili metodi disponibili nella classe `String`.

- `public boolean startsWith(String prefix)`: restituisce `true` se la stringa `prefix` corrisponde alle prime lettere della stringa a cui il metodo è applicato, `false` altrimenti. Questo metodo restituisce `true` sia se la stringa `prefix` è vuota sia se ha lo stesso contenuto della stringa a cui è applicato il metodo.
- `public boolean startsWith(String prefix, int ts)`: questo metodo è equivalente all'espressione `this.substring(ts).startsWith(prefix)`.
- `public boolean endsWith(String suffix)`: restituisce `true` se la stringa `suffix` corrisponde alle ultime lettere della stringa a cui il metodo è applicato, `false` altrimenti. Questo metodo restituisce `true` sia se la stringa `suffix` è vuota sia se ha lo stesso contenuto della stringa a cui è applicato il metodo.
- `public String concat(String str)`: restituisce una nuova stringa formata dalla concatenazione della stringa a cui è applicato il metodo con `str`. Per esempio, l'espressione `"Oggi è ".concat("lunedì")` restituisce la stringa `"Oggi è lunedì"`.
- `public String replace(char oldChar, char newChar)`: restituisce una nuova stringa che differisce da quella a cui è applicato il metodo per il fatto che tutte le occorrenze di `oldChar` sono sostituite con `newChar`.
- `public String toLowerCase()`: restituisce una nuova stringa che differisce da quella a cui è applicato il metodo per il fatto che tutte le lettere maiuscole che compaiono all'interno sono trasformate in lettere minuscole.
- `public String toUpperCase()`: restituisce una nuova stringa che differisce da quella a cui è applicato il metodo per il fatto che tutte le lettere minuscole che compaiono all'interno sono trasformate in lettere maiuscole.
- `public boolean regionMatches(int ts, String o, int os, int len)`: questo metodo restituisce `true` o `false` equivalentemente all'espressione `s1.substring(ts,ts+len).equals(s2.substring(os,os+len))`.
- `public boolean regionMatches(boolean iC, int ts, String o, int os, int len)`: questo metodo restituisce `true` o `false` equivalentemente all'espressione `s1.substring(ts,ts+len).equalsIgnoreCase(s2.substring(os,os+len))` se `iC` vale `true`, altrimenti si comporta come il metodo precedente.
- `public String trim()`: restituisce una nuova stringa in cui sono stati tolti eventuali caratteri bianchi o di controllo dalla testa e dalla coda.
- `public String intern()`: Java mantiene un pool di oggetti stringa pronti per essere utilizzati. Quando si invoca questo metodo, viene verificato se esiste un oggetto del pool uguale a quello su cui il metodo è invocato. In caso affermativo, viene restituita la referenza a tale oggetto, altrimenti la stringa viene aggiunta al pool e ne viene restituita la referenza. Viene utilizzato solo per motivi di ottimizzazione.

11.3.6 La classe StringBuffer

Come detto, la classe StringBuffer si usa saltuariamente in modo esplicito ma alcuni dei suoi metodi sono utilizzati implicitamente molto frequentemente.

```
public StringBuffer append(boolean b);
public StringBuffer append(char c);
public StringBuffer append(int i);
public StringBuffer append(long l);
public StringBuffer append(float f);
public StringBuffer append(double d);
public StringBuffer append(Object obj);
public StringBuffer append(char[] str);
public StringBuffer append(char[] str, int offset, int len);
public StringBuffer append(String str);
```

Questi metodi permettono di appendere in fondo a un oggetto di tipo StringBuffer un oggetto di qualsiasi tipo. Si può notare che questa serie di metodi ha gli argomenti corrispondenti a quelli dei metodi valueOf visti in precedenza, con in più uno per appendere un oggetto di tipo String. Quest'ultimo metodo teoricamente è inutile perché il metodo che ha come argomento un Object funziona correttamente anche per le String, ma da un punto di vista pratico consente delle prestazioni migliori.

Tutti questi metodi append consentono di costruire una stringa come la concatenazione di stringhe diverse. Se, per esempio, avessimo il giorno del mese in una variabile intera e il mese in una variabile stringa, come nel codice seguente

```
int giorno = 17;
String mese = "Giugno";
```

e volessimo mettere in una stringa un messaggio indicante la data odierna, potremmo usare un'espressione del tipo

```
String dataOggi = new StringBuffer()
                      .append("Oggi è il ")
                      .append(giorno)
                      .append(" di ")
                      .append(mese)
                      .toString();
```

Analizziamo questa espressione che ha un aspetto un po' complesso. Per prima cosa viene creato un oggetto di tipo StringBuffer vuoto, dopodiché sull'oggetto appena creato si applica il primo metodo append ottenendo così un oggetto contenente "Oggi è il ". Su questo oggetto così modificato si invoca ancora il metodo append ottenendo in tal modo che l'istanza contenga "Oggi è il 17" e così via fino all'ultimo metodo append che ci consente di avere l'istanza di StringBuffer contenente "Oggi è il 17 di Giugno". Su tale istanza invochiamo infine il metodo toString che restituisce una stringa con il contenuto dell'istanza di StringBuffer che può essere così assegnata a dataOggi.

Questa operazione, chiamata *concatenazione di stringhe*, è molto usata in modo particolare per comporre dei messaggi per l'utente, per cui i progettisti di Java hanno pensato di semplificarne l'utilizzo mediante l'operatore +. Ecco così che l'espressione precedente può essere scritta più semplicemente nel modo seguente

```
String dataOggi = "Oggi è il " + giorno + " di " + mese;
```

La maggiore leggibilità di questa seconda notazione è evidente. La capacità dei metodi `append` di poter appendere praticamente qualsiasi cosa rende possibile usare la concatenazione di stringhe in molte circostanze. Supponiamo per esempio di voler ottenere una stringa contenente la rappresentazione di un `double` e di non ricordare il metodo adatto. Allora il codice seguente può servire allo scopo.

```
double d = 1.23;
String dStringa = "" + d;
```

Nel terzo capitolo, parlando del fatto che una costante stringa non può essere spezzata su più righe, abbiamo visto che più costanti stringa possono essere concatenate con l'operatore +. Ora siamo in grado di capire come è implementata questa caratteristica.

11.4 Collezioni

Come accennato nel precedente capitolo, nello sviluppo di programmi, specialmente a oggetti, capita spesso di aver bisogno di 'contenitori' di oggetti. Il linguaggio mette già a disposizione gli array per questo scopo, ma non sempre essi risultano sufficientemente flessibili. Ecco quindi che sono state inserite, tra le librerie di corredo del linguaggio, una serie di classi che implementano diversi tipi di "collezioni", come, per esempio, vettori dinamici e liste concatenate. Questa libreria è tanto ricca e articolata da essere definita "Collection Framework". Per permettere che i diversi tipi di contenitori possano interagire l'uno con l'altro, è stata creata una gerarchia di interfacce nel package `java.util` che raggruppa le differenti classi collezione a seconda delle funzionalità offerte. La radice della gerarchia è l'interfaccia `Collection<E>` che raggruppa quindi qualsiasi tipo di collezione. Questa interfaccia deriva a propria volta dall'interfaccia `java.lang.Iterable<E>`, che vedremo più avanti, e in realtà non è implementata direttamente da nessuna classe presente nella libreria di Java, ma vengono fornite implementazioni delle sue interfacce derivate, come `Set<E>` e `List<E>`. La differenza tra queste due interfacce consiste nel fatto che la prima non permette di avere due o più elementi uguali tra loro, mentre la seconda non ha questa limitazione. Da `Set<E>` deriva l'interfaccia `SortedSet<E>` che mantiene gli elementi in un determinato ordine dipendente dalle caratteristiche degli oggetti contenuti.

Tutte le classi di utilizzo generale che implementano `Collection<E>` hanno, per convenzione, un costruttore senza argomenti che consente la creazione di una collezione vuota e un costruttore che ha un'altra collezione come argomento: quest'ultimo permette di fatto di poter "travasare" tutti gli elementi di una collezione in un'altra, dello stesso tipo o di tipo diverso. Il linguaggio non permette di rendere obbligatorio il rispetto di questa convenzione perché nelle interfacce non si possono specificare i costruttori, nondimeno tutte le classi presenti nel framework la rispettano.

Prima di illustrare brevemente i metodi definiti in `Collection<E>`, va detto che alcuni di essi implementano operazioni opzionali, operazioni cioè che possono non essere applicabili a una particolare implementazione di collezione: naturalmente questi metodi sono in ogni caso definiti per fare in modo che la classe rispetti il contratto imposto dall'interfaccia, ma quando non sono applicabili o non fanno nulla o gettano un'eccezione. La lista completa dei metodi è la seguente:

- `boolean add(E e)`: assicura che l'oggetto specificato come argomento sia incluso nella collezione. Restituisce `true` se l'elemento viene aggiunto realmente alla collezione, `false` altrimenti: quest'ultima circostanza si verifica solo se la collezione non permette il contenimento di elementi uguali e l'argomento è già presente nella collezione. Questa operazione è opzionale.
- `boolean addAll(Collection<? extends E> c)`: questo metodo è equivalente all'invocazione del metodo `add(E e)` per ciascun elemento della collezione specificata come argomento. Restituisce `true` se almeno un elemento viene aggiunto alla collezione, `false` altrimenti. Anche questa operazione è opzionale.
- `void clear()`: rimuove ogni elemento dalla collezione. Questa operazione è opzionale.
- `boolean contains(Object o)`: restituisce `true` se l'elemento fornito come argomento è contenuto nella collezione.
- `boolean containsAll(Collection<?> c)`: restituisce `true` se tutti gli elementi contenuti nella collezione fornita come argomento sono contenuti nella collezione, `false` altrimenti.
- `boolean equals(Object o)`: questo metodo fa parte della classe `Object`, per cui potrebbe non essere ridefinito, tuttavia le collezioni normalmente lo ridefiniscono per permette il confronto tra collezioni sulla base del contenuto.
- `int hashCode()`: anche questo metodo fa parte della classe `Object`, per cui potrebbe non essere ridefinito, tuttavia le collezioni normalmente lo ridefiniscono per permettere la distinzione di liste diverse sulla base del contenuto.
- `boolean isEmpty()`: restituisce `true` se la collezione è vuota, `false` altrimenti.
- `Iterator<E> iterator()`: restituisce un oggetto della classe `Iterator<E>` che permette di scorrere gli elementi di una collezione, come vedremo meglio più avanti. Questo metodo è l'unico dell'interfaccia `Iterable<E>`.
- `boolean remove(Object o)`: rimuove l'oggetto specificato come argomento dalla collezione. Restituisce `true` se l'elemento viene trovato e rimosso, `false` altrimenti. Questa operazione è opzionale.
- `boolean removeAll(Collection<?> c)`: questo metodo è equivalente all'invocazione del metodo `remove(Object o)` per ciascun elemento della collezione specificata come argomento. Restituisce `true` se almeno un elemento viene rimosso dalla collezione, `false` altrimenti. Anche questa operazione è opzionale.

- `boolean retainAll(Collection<?> c)`: fa in modo che nella collezione rimangano solo gli oggetti contenuti anche nella collezione passata come argomento. Questa operazione è opzionale.
- `int size()`: restituisce il numero di elementi presente nella collezione.
- `Object[] toArray()`: restituisce un array di `Object` contenente tutti gli oggetti contenuti nella collezione.
- `<T> T[] toArray(T[] a)`: questo metodo è analogo al precedente, ma permette di specificare il tipo dell'array che viene richiesto fornendone uno opportuno come argomento: se quest'ultimo è sufficientemente capace da contenere tutti gli elementi della collezione, esso stesso viene riempito con detti elementi e restituito, altrimenti ne viene creato uno nuovo. Se il tipo del vettore specificato non è adatto a contenere gli oggetti della collezione, viene gettata un'eccezione.

11.4.1 La classe `Vector<E>`

La classe `Vector<E>`, che implementa l'interfaccia `List<E>` e appartiene al package `java.util`, mette a disposizione le funzionalità di un array ma, a differenza di quest'ultimo, non necessita di alcun dimensionamento. Il funzionamento è abbastanza semplice e simile a quello che abbiamo sviluppato nel capitolo precedente; la classe mantiene al suo interno un array di una certa capacità e man mano che gli oggetti vengono aggiunti, le loro referenze vengono memorizzate in esso. Quando la capacità dell'array non è più sufficiente, viene creato un nuovo array più capace e in esso vengono ricopiati tutti gli elementi del vecchio.

I costruttori a disposizione di questa classe sono i seguenti

- `public Vector()`: costruisce un oggetto della classe che ha una capacità interna di default di 10 elementi. Un array creato con questo metodo ha un *incremento di capacità* uguale a 0, il che significa che ogni volta che viene creato un nuovo array per contenere gli oggetti perché quello attuale non è più sufficiente, la capacità del nuovo viene impostata al doppio della capacità dell'attuale.
- `public Vector(int initialCapacity)`: costruisce un oggetto della classe che ha una capacità interna di default di `initialCapacity` elementi. Anche un array creato con questo metodo ha un incremento di capacità uguale a 0.
- `public Vector(int initialCapacity, int capacityIncrement)`: costruisce un oggetto della classe che ha una capacità interna di default di `initialCapacity` elementi. Un array creato con questo metodo ha un incremento di capacità fisso uguale a `capacityIncrement`.
- `public Vector (Collection<? extends E> c)`: costruisce un oggetto della classe e inserisce al suo interno tutti gli elementi contenuti nell'istanza passata come argomento.

Bisogna distinguere la *capacità* di un oggetto di tipo `Vector` dalla sua *misura*: la prima infatti indica un parametro interno all'oggetto che non ha interesse per il programmatore se non per questioni di ottimizzazione, mentre la seconda indica il numero

di elementi effettivamente contenuti nell'oggetto. Due metodi restituiscono due interi che rappresentano queste grandezze: il primo `public final int capacity()` restituisce la capacità, mentre il secondo `public int size()` restituisce il numero di elementi.

Un oggetto della classe `Vector<E>` dispone di tutte le operazioni disponibili nelle interfacce `Collection<E>` e `List<E>`, comprese quelle opzionali, ma possiede anche altri metodi, sia per motivi storici che per rendere più agevole il suo uso.

Altri metodi per aggiungere e togliere degli elementi da un oggetto di tipo `Vector<E>` sono:

- `public void addElement(E obj)`: aggiunge `obj` dopo l'ultimo elemento.
- `public void insertElementAt(E obj, int index)`: inserisce `obj` nella posizione indicata dall'indice `index`. Tutti gli elementi che si trovavano in una posizione maggiore o uguale a `index` vengono spostati in avanti di una posizione. Se `index` è maggiore della misura del vettore o è minore di 0 viene gettata un'eccezione.
- `public void setElementAt(E obj, int index)`: sostituisce l'elemento in posizione `index` con `obj`. Se `index` è maggiore della misura del vettore o è minore di 0 viene gettata un'eccezione.
- `public void removeElementAt(int index)`: viene rimosso l'elemento che si trova in posizione `index`. Tutti gli elementi che si trovavano in una posizione maggiore a `index` vengono spostati indietro di una posizione. Se `index` è maggiore della misura del vettore o è minore di 0 viene gettata un'eccezione.
- `public boolean removeElement(E obj)`: se il `Vector` contiene l'oggetto `obj`, esso viene rimosso con le modalità del metodo precedente e il metodo restituisce `true`, altrimenti il vettore non viene modificato e il metodo restituisce `false`.
- `public void removeAllElements()`: rimuove tutti gli elementi dal `Vector`.

Per ricuperare gli elementi da un `Vector` sono disponibili i seguenti metodi:

- `public E elementAt(int index)`: restituisce l'elemento a posizione `index`;
- `public E firstElement()`: restituisce il primo elemento;
- `public E lastElement()`: restituisce l'ultimo elemento.

Ora che abbiamo esaminato i principali metodi della classe `Vector`, vediamo un esempio.

```
import java.util.Vector;

class SuiVettori
{
  String p1, p2;
  public static void main (String argv[]) {
    Vector<String> v = new Vector<String>(3);
    System.out.println("capacità         >" + v.capacity());
```

```
    System.out.println("n.elementi       >" + v.size());
    v.addElement ("a");
    v.addElement ("b");
    v.addElement ("d");
    v.addElement ("e");
    v.insertElementAt ("c", 2);
    System.out.println("capacità         >" + v.capacity());
    System.out.println("n.elementi       >" + v.size());
    System.out.println("primo  elemento >" +
                       v.firstElement());
    System.out.println("ultimo elemento >" +
                       v.lastElement());
    for (int i = 0; i < v.size(); i++)
      System.out.println("elemento a " + i + "     >" +
                         v.elementAt(i));
  }
}
```

Va notata l'istruzione `import java.util.Vector` senza la quale il compilatore segnala un errore perché non trova la classe `Vector<E>`. Il programma produce la seguente visualizzazione:

```
capacità         >3
n.elementi       >0
capacità         >6
n.elementi       >5
primo  elemento >a
ultimo elemento >e
elemento a 0     >a
elemento a 1     >b
elemento a 2     >c
elemento a 3     >d
elemento a 4     >e
```

Dalla classe `Vector<E>` deriva la classe `Stack<E>` che implementa un coda LIFO (Last-In.First-Out).

11.4.2 Iteratori

L'operazione eseguita più frequentemente sulle collezioni è la scansione sequenziale dei suoi elementi. Guardando la descrizione dell'interfaccia `Collection<E>`, si vede che non esistono metodi adatti a eseguire direttamente questo tipo di operazione, ma che è necessario ottenere, tramite il metodo `iterator()`, un oggetto che implementa l'interfaccia `Iterator<E>` e usare quello per la scansione. Questa interfaccia definisce tre metodi e precisamente:

- `boolean hasNext()`: restituisce `true` se c'è almeno un elemento ancora da scandire, false altrimenti;

- `E next()`: restituisce l'elemento successivo;
- `void remove()`: rimuove dalla collezione l'ultimo elemento ricuperato con il metodo `next()`: questa operazione è opzionale.

Un oggetto che implementa questa interfaccia viene detto appunto un *iteratore*. La scansione sequenziale di una collezione può quindi essere fatta facilmente come nel seguente esempio:

```
public static void stampaCollezione (Collection<?> c) {
      for (Iterator<?> i = c.iterator(); i.hasNext(); )
         System.out.println (i.next());
}
```

Ci si può chiedere perché i metodi dell'interfaccia `Iterator<E>` non siano stati inclusi direttamente nell'interfaccia `Collection<E>`, evitando così la necessità di dover creare un nuovo oggetto per ogni scansione: il motivo è che un'implementazione di questo tipo equivarrebbe ad aver associato a ogni collezione uno e un solo iteratore e che questo non è sufficientemente flessibile e può causare errori difficili da individuare. Per esempio supponiamo di dover scrivere un metodo che segnali se una collezione contiene elementi duplicati o meno:

```
public static boolean haDuplicati (Collection<?> c) {
   Object o;
   boolean trovato;
   for (Iterator<?> i = c.iterator(); i.hasNext(); ) {
      o = i.next();
      trovato = false;
      for (Iterator<?> j = c.iterator(); j.hasNext(); )
         if (o.equals (j.next()))
            if (trovato)
               return true;
            else
               trovato = true;
   }
   return false;
}
```

Un algoritmo di questo tipo non può essere implementato usando un solo iteratore.

11.4.3 Costrutto `for-each`

Come si può notare dagli esempi precedenti, l'impostazione di un ciclo per effettuare la scansione di una collezione è sempre formalmente uguale, ma risulta un po' prolissa e non si adatta perfettamente all'impostazione del costrutto `for`: infatti l'istruzione d'incremento è sempre vuota e l'avanzamento del cursore viene fatto in un'istruzione

dentro il ciclo. Per snellire questa sintassi, in Java 5 è stata introdotta una nuova forma di costrutto `for`, chiamato *for-each* (per-ciascuno), pensata esplicitamente per fare scansioni con iteratori, ma usabile anche con array.

La sintassi è la seguente:

```
for (dichiarazione : collezione) istruzione;
```

Tra le parentesi si possono distinguere due parti separate dal carattere due punti (:). Nella seconda parte compare un oggetto che implementa l'interfaccia `Iterable<E>` oppure un array, mentre nella prima si dichiara una variabile di tipo opportuno che dovrà contenere tutti i valori della collezione. Come già accennato, l'interfaccia `Iterable<E>` dichiara un unico metodo `Iterator<E> iterator()`. Usando questo costrutto, il precedente metodo per riconoscere se una collezione ha elementi duplicati diventa:

```
public static boolean haDuplicati (Collection<?> c) {
   boolean trovato;
   for (Object i : c) {
      trovato = false;
      for (Object j : c)
         if (i.equals (j))
            if (trovato)
               return true;
            else
               trovato = true;
   }
   return false;
}
```

Per riassumere quanto visto, riportiamo un esempio dove riprendiamo l'oggetto `Vettore<T>` visto nel capitolo precedente e gli facciamo implementare la classe `Iterable<T>`.

```
import java.util.Iterator;

class VI<T> implements Iterator<T> {
   private Vettore<T> v;
   private int indice;
   VI (Vettore<T> v) {
      this.v = v;
   }
   public boolean hasNext () {
      return indice < v.lunghezza();
   }
   public T next() {
      if (indice < v.lunghezza())
       return v.prendi (indice++);
```

```
     else
        return null;
   }
   public void remove() {
   }
}

public class Vettore<T> implements Iterable<T> {
   private Object vettore[] = new Object[0];

   public void metti (T elemento, int posizione) {
      if (posizione >= vettore.length) {
         Object nuovo[] = new Object[posizione + 1];
         for (int i = 0; i < vettore.length; i++)
             nuovo[i] = vettore[i];
         vettore = nuovo;
      }
      vettore[posizione] = elemento;
   }

   public T prendi (int posizione) {
      if (posizione >= vettore.length)
         return null;
      else
         return (T) vettore[posizione];
   }
   int lunghezza() {
      return vettore.length;
   }
   public Iterator<T> iterator() {
      return new VI<T>(this);
   }

   public static void main (String argv[]) {
      Vettore<String> v = new Vettore<String>();
      v.metti ("A", 1);
      v.metti ("B", 2);
      v.metti ("C", 3);
      v.metti ("E", 5);
     for (String s : v)
        System.out.println (s);
   }
}
```

11.5 Classi involucro

Come abbiamo visto, i tipi primitivi non appartengano alla gerarchia degli oggetti per motivi prestazionali. Questo fatto normalmente non crea problemi ma in alcuni casi risulta scomodo. Per esempio un'istanza della classe `Vector` può contenere istanze di qualsiasi oggetto ma non variabili di tipi primitivi, il che in certe situazioni potrebbe invece tornare utile. Per risolvere questo tipo di problema, nel package `java.lang` sono disponibili alcune classi che servono da involucro dei tipi primitivi, vale a dire classi che hanno un tipo primitivo come unica variabile d'istanza e il cui scopo principale è quello di permettere un trattamento omogeneo di oggetti e tipi primitivi. Le classi involucro vengono istanziate sempre e solo assegnando loro un valore, che poi non può essere più modificato; per questo motivo gli oggetti di queste classi possono essere considerati costanti.

Le classi involucro presenti sono le seguenti:

- `Boolean`
- `Byte`
- `Character`
- `Double`
- `Float`
- `Integer`
- `Long`
- `Number`
- `Short`

Vediamo ora più da vicino alcune di queste classi.

11.5.1 Classe `Number`

Questa è una classe astratta che è la radice delle classi che implementano i tipi numerici, e cioè `Byte`, `Double`, `Float`, `Integer` e `Long`. `Number` dispone dei seguenti metodi che sono ridefiniti in tutte le classi concrete derivate

- `public byte byteValue()`: restituisce il valore del numero come `byte`, eventualmente arrotondato o troncato;
- `public short shortValue()`: restituisce il valore del numero come `short`, eventualmente arrotondato o troncato;
- `public abstract int intValue()`: restituisce il valore del numero come `int`, eventualmente arrotondato o troncato;
- `public abstract long longValue()`: restituisce il valore del numero come `long`, eventualmente arrotondato o troncato;
- `public abstract float floatValue()`: restituisce il valore del numero come `float`, eventualmente arrotondato;

- `public abstract double doubleValue()`: restituisce il valore del numero come `double`, eventualmente arrotondato.

11.5.2 Classi involucro per numeri interi

Le classi seguenti derivano tutte dalla classe `Number` e servono come involucro per i tipi interi

- `Byte`
- `Integer`
- `Long`
- `Short`

Ognuna di queste classi ha definite due costanti, `MIN_VALUE` e `MAX_VALUE`, che contengono rispettivamente i valori minimo e massimo che possono essere contenuti.

Per il resto, ognuna ridefinisce tutti i metodi della classe `Number` e della classe `Object`. Le classi `Integer` e `Long` hanno definiti alcuni metodi statici di utilità. Daremo uno sguardo solamente alla classe `Integer` tenendo conto che la classe `Long` ha dei metodi statici analoghi.

La classe `Integer` ha i seguenti costruttori

- `public Integer(int i)`: costruisce un oggetto assegnandogli il valore `i`;
- `public Integer(String s)`: costruisce un oggetto assegnandogli il valore rappresentato in formato decimale dalla stringa `s`.

Oltre a ridefinire tutti i metodi della classe `Number` e della classe `Object`, la classe `Integer` dispone dei seguenti metodi statici di utilità.

- `public static String toString(int i, int radix)`: restituisce una rappresentazione in formato stringa in base `radix` del numero `i`. `radix` può assumere un qualsiasi valore compreso tra le costanti statiche `Character.MIN_RADIX` e `Character.MAX_RADIX`. Nel caso in cui `radix` non sia tra i valori predetti, il numero viene convertito in base 10. I simboli usati per la rappresentazione dei numeri consistono nelle 10 cifre numeriche e in tutte e 26 le lettere minuscole dell'alfabeto; da questo si evince che `RADIX` non può eccedere 36.
- `public static String toString(int i)`: restituisce una rappresentazione in formato stringa in base 10 del numero `i`.
- `public static String toHexString(int i)`: restituisce una rappresentazione in formato stringa in base 16 del numero `i`.
- `public static String toOctalString(int i)`: restituisce una rappresentazione in formato stringa in base 8 del numero `i`.
- `public static String toBinaryString(int i)`: restituisce una rappresentazione in formato stringa in base 2 del numero `i`.

- `public static int parseInt(String s, int radix)`: restituisce un intero di valore corrispondente a quello rappresentato in base `radix` nella stringa `s`. `radix` può assumere valori analoghi a quelli illustrati nel metodo `public static String toString(int i, int radix)`.
- `public static int parseInt(String s)`: restituisce un intero di valore corrispondente a quello rappresentato in base `10` nella stringa `s`.
- `public static Integer valueOf(String s)`: corrisponde all'invocazione `new Integer(Integer.parseInt(s))`.
- `public static Integer valueOf(String s, int radix)`: corrisponde all'invocazione `new Integer(Integer.parseInt(s, radix))`.

11.5.3 Classi involucro per numeri in virgola mobile

Le classi seguenti derivano tutte dalla classe `Number` e servono come involucro per i tipi in virgola mobile

- `Float`
- `Double`

I numeri in virgola mobile sono memorizzati in Java con il formato IEEE 754, come specificato nel documento *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985 (IEEE, New York). Questo formato prevede tre valori speciali che servono per trattare i risultati delle divisioni per 0. Ciò significa che la divisione per 0 non genera eccezioni nel caso venga effettuata su dati in virgola mobile. Entrambe le classi hanno definite, oltre le costanti `MIN_VALUE` e `MAX_VALUE` che hanno un significato analogo a quelle già viste, le seguenti costanti

- `public static final float NEGATIVE_INFINITY`
- `public static final double NEGATIVE_INFINITY`: indica il valore "infinito negativo" ed è definito come l'espressione `-1.0f/0.0f` nella classe `Float` e `-1.0d/0.0d` nella classe `Double`.
- `public static final float POSITIVE_INFINITY`
- `public static final double POSITIVE_INFINITY`: indica il valore "infinito positivo" ed è definito come l'espressione `1.0f/0.0f` nella classe `Float` e `1.0d/0.0d` nella classe `Double`.
- `public static final float NaN`
- `public static final double NaN`: indica il valore di numero indeterminato, o "non numero" (Not-a-Number), ed è definito come l'espressione `0.0f/0.0f` nella classe `Float` e `0.0d/0.0d` nella classe `Double`.

Le prime due costanti possono essere utilizzate per riconoscere rispettivamente se un numero negativo o positivo è stato diviso per 0 con espressioni del tipo:

```
if (d == Double.NEGATIVE_INFINITY)
   System.out.println ("negativo diviso per 0");
else if (d == Double.POSITIVE_INFINITY)
   System.out.println ("positivo diviso per 0");
```

Questo approccio non può essere usato per riconoscere un valore NaN in quanto, essendo indeterminato, è diverso da qualsiasi valore, anche da se stesso. L'espressione x!=x è vera se e solo se x contiene un valore NaN e l'espressione (x<y)==!(x>=y) è falsa se e solo se x o y contengono un valore NaN.

Per trattare questi valori particolari, sono definiti nelle classi Float e Double i seguenti metodi:

- public boolean isNaN(): restituisce true se l'istanza corrente contiene un valore NaN, false altrimenti.
- public boolean isInfinite(): restituisce true se l'istanza corrente contiene un valore NEGATIVE_INFINITY o POSITIVE_INFINITY, false altrimenti.
- public static boolean isNaN(float v)
- public static boolean isNaN(double v): restituisce true se l'argomento passato contiene un valore NaN, false altrimenti.
- public static boolean isInfinite(float v)
- public static boolean isInfinite(double v): restituisce true se l'argomento contiene un valore NEGATIVE_INFINITY o POSITIVE_INFINITY, false altrimenti.

11.5.4 Classi BigDecimal e BigInteger

Le classi BigDecimal e BigInteger servono per contenere numeri con un numero praticamente illimitato di cifre. Si differenziano l'una dall'altra per il fatto che la prima può gestire numeri con la virgola, mentre la seconda può gestire solo numeri interi. Esse non sono classi involucro, ma vengono menzionate in questo paragrafo perché discendono entrambe da Number come le classi precedenti. In effetti queste classi appartengono al package java.math e sono state aggiunte in un secondo momento per risolvere problemi particolari, come lo scambio di dati con i database e calcoli per scopi crittografici. Come le stringhe, anche gli oggetti di queste classi sono immutabili.

Gli oggetti di entrambe le classi permettono di effettuare calcoli aritmetici solo tra oggetti della stessa classe; è possibile però creare un oggetto di classe BigDecimal a partire da uno di classe BigInteger ed eseguire la conversione inversa specificando il metodo di arrotondamento desiderato.

11.5.5 Autoboxing

Le classi involucro permettono di superare il problema della differenza tra tipi primitivi e oggetti, ma possono risultare noiose da usare e complicare la stesura dei programmi. Per questo motivo in Java 5 è stata introdotta nel linguaggio un'integrazione

più stretta tra gli oggetti di queste classi e i tipi primitivi, chiamata *autoboxing* (autoinscatolamento). In pratica il compilatore analizza il codice e crea automaticamente un oggetto di una classe involucro opportuna (autoboxing) oppure esegue l'operazione contraria (unboxing) quando risulta necessario.

Per fare un esempio, creiamo un oggetto della classe `Vector<Integer>`, riempiamolo con numeri interi e passiamolo a un metodo che ha il compito di stabilire se ciascun numero nel vettore è pari o dispari; il codice è il seguente:

```
import java.util.Vector;

public class AutoBoxing {
   public static void main (String argv[]) {
      Vector<Integer> v = new Vector<Integer>();
      for (int i = 1; i < 10; i++)
         v.addElement (i);
      trovaDispari(v);
   }

   static void trovaDispari (Vector<Integer> v) {
      for (int i : v)
         if ((i & 1) == 1)
            System.out.println ("dispari = " + i);
         else
            System.out.println ("pari    = " + i);
   }
}
```

Come si vede grazie all'autoboxing l'uso di un oggetto `Vector<Integer>` diventa semplice come usare un normale array. Non bisogna dimenticare però che vengono eseguite delle operazioni nascoste che possono influire in modo drastico sulle prestazioni. Il codice precedente è del tutto equivalente al seguente:

```
import java.util.Vector;

public class AutoBoxing {
   public static void main (String argv[]) {
      Vector<Integer> v = new Vector<Integer>();
      for (int i = 1; i < 10; i++)
         v.addElement (new Integer(i));
      trovaDispari(v);
   }

   static void trovaDispari (Vector<Integer> v) {
      for (Integer i : v)
         if ((i.intValue() & 1) == 1)
            System.out.println ("dispari = " + i);
         else
```

```
            System.out.println ("pari     = " + i);
    }
}
```

La differenza di prestazioni normalmente non è rilevante per applicazioni normali, ma è opportuno evitare l'autoboxing in presenza di calcoli lunghi dove la velocità d'esecuzione è un fattore critico.

11.6 Funzioni matematiche

La classe `Math` del package `java.lang` è formata da una serie di metodi statici per il calcolo delle funzioni matematiche e trigonometriche. Riportiamo di seguito l'elenco delle variabili di classe e dei metodi statici senza commenti, rimandando il lettore ai manuali del linguaggio nel caso di necessità.

```
public static final double E = 2.7182818284590452354;
public static final double PI = 3.14159265358979323846;
public static native double sin(double a);
public static native double cos(double a);
public static native double tan(double a);
public static native double asin(double a);
public static native double acos(double a);
public static native double atan(double a);
public static double toRadians(double angdeg);
public static double toDegrees(double angrad);
public static native double exp(double a);
public static native double log(double a);
public static native double sqrt(double a);
public static native double IEEEremainder(double f1,double f2);
public static native double ceil(double a);
public static native double floor(double a);
public static native double rint(double a);
public static native double atan2(double a, double b);
public static native double pow(double a, double b);
public static int round(float a);
public static long round(double a);
public static synchronized double random();
public static int abs(int a);
public static long abs(long a);
public static float abs(float a);
public static double abs(double a);
public static int max(int a, int b);
public static long max(long a, long b);
public static float max(float a, float b);
public static double max(double a, double b);
```

```
public static int min(int a, int b);
public static long min(long a, long b);
public static float min(float a, float b);
public static double min(double a, double b)
```

11.7 Enumerali

Nella stesura di programmi spesso è necessario rappresentare oggetti del mondo, siano essi astratti o concreti, per i quali non è necessario specificare alcuna caratteristica particolare, ma che fanno semplicemente parte di un elenco più o meno fissato. Alcuni esempi di oggetti di questo tipo sono i giorni della settimana, i colori, le carte da gioco di un mazzo, ecc. Elenchi di questo tipo vengono chiamate *enumerazioni*; prima dell'uscita Java 5 venivano gestite semplicemente assegnando dei numeri progressivi a delle costanti numeriche il cui nome ricalcava quello dell'oggetto rappresentato. Per esempio i giorni della settimana potevano essere rappresentati nel modo seguente:

```
public static final int LUNEDI    = 0;
public static final int MARTEDI   = 1;
public static final int MERCOLEDI = 2;
public static final int GIOVEDI   = 3;
public static final int VENERDI   = 4;
public static final int SABATO    = 5;
public static final int DOMENICA  = 6;
```

Questo sistema però presenta diversi inconvenienti: è molto prolisso e nel caso di elenchi molto lunghi diventa difficile da manutenere; visualizzando un elemento si vede semplicemente un numero, ma il difetto maggiore è che è insicuro. Supponiamo di avere un metodo che accetta come argomento un giorno della settimana, il compilatore non può fornire alcuna garanzia che l'argomento abbia un valore adeguato, compreso cioè tra 0 e 6, per cui si possono inserire degli errori nel programma che verranno evidenziati in un momento imprevedibile durante l'uso.

Per ovviare a questo problema, in Java 5 è stato inserito un nuovo tipo, chiamato `enum`, che permette di semplificare la dichiarazione appena vista nel modo seguente:

```
public enum Giorno {
                    LUNEDI, MARTEDI, MERCOLEDI, GIOVEDI,
                    VENERDI, SABATO, DOMENICA };
```

Abbiamo detto che è un tipo nuovo, ma in realtà si tratta di una vera e propria classe che, grazie a una gestione integrata nel linguaggio analoga a quella vista per l'autoboxing, offre delle funzionalità non disponibili in altri linguaggi che pure hanno un tipo enumerale. Nell'esempio precedente quindi `Giorno` è una classe che deriva da `java.lang.Enum<E extends Enum<E>>` e che ha definiti sette membri statici corrispondenti ai giorni della settimana. Ciascun membro dispone di tutti i metodi della classe `Enum` e in più ne ha uno statico, `values()`, che restituisce un array contenente tutti i membri. Vediamo ora un esempio che riassume le principali caratteristiche di questo tipo.

```
public class Enumerali {
   public enum Giorno {
               LUNEDI, MARTEDI, MERCOLEDI, GIOVEDI,
               VENERDI, SABATO, DOMENICA };

   public static void main (String argv[]) {
      for (Giorno g : Giorno.values())
         vedi (g);
   }
   public static void vedi (Giorno g) {
      switch (g) {
      case LUNEDI:
         System.out.println ("Lunedì=" + g);
         break;
      case MARTEDI:
         System.out.println ("Martedì=" + g);
         break;
      case MERCOLEDI:
         System.out.println ("Mercoledì=" + g);
         break;
      case GIOVEDI:
         System.out.println ("Giovedì=" + g);
         break;
      case VENERDI:
         System.out.println ("Venerdì=" + g);
         break;
      case SABATO:
         System.out.println ("Sabato=" + g);
         break;
      case DOMENICA:
         System.out.println ("Domenica=" + g);
         break;
      }
   }
}
```

Dall'esempio si può notare che è molto semplice eseguire un'iterazione su tutti i valori dell'insieme tramite il costrutto `for-each` e che è possibile anche trattare gli elementi usando il costrutto `switch` come se fossero numeri interi. La sorpresa però giunge quando si visualizzano i singoli elementi, infatti non vediamo un semplice numero, ma il nome dell'elemento stesso. Questa "magia" è realizzata dal compilatore che aggiunge delle operazioni in modo da semplificare al massimo il compito del programmatore. Essendo `Giorno` una classe a tutti gli effetti, il metodo `vedi(Giorno g)` accetterà solo argomenti di questo tipo e quindi non saranno accettati enumerali di altro tipo né tanto meno numeri interi.

Il fatto che gli enumerali siano implementati come classi permette di ottenere funzionalità neppure immaginabili con altri linguaggi: le classi infatti possono essere estese e così pure gli enumerali. Supponiamo, per esempio, di voler enumerare dei colori e di voler associare a ciascuno di essi le sue componenti RGB per la visualizzazione su uno schermo di computer: basta estendere un tipo enum aggiungendo le informazioni necessarie ed eventualmente i metodi per ricuperare tali informazioni. Nell'esempio seguente viene creato un enumerale esteso con i 7 colori base e ne viene fatta la visualizzazione.

```
public class EnumeraliEstesi {
   public enum Colore {
         ROSSO (255, 0, 0),
         VERDE (0, 255, 0),
         BLU (0, 0, 255),
         GIALLO (255, 255, 0),
         CIANO (0, 255, 255),
         MAGENTA (255, 0, 255),
         BIANCO (255, 255, 255);
      private final int verde;
      private final int rosso;
      private final int blu;
      private Colore (int r, int g, int b) {
         rosso = r;
         verde = g;
         blu = b;
      }
      public int rosso() { return rosso; }
      public int verde() { return verde; }
      public int blu()   { return blu; }
   };

   public static void main (String argv[]) {
      for (Colore c : Colore.values())
         System.out.println ("colore " + c + "=("
                                          + c.rosso() + ","
                                          + c.verde() + ","
                                          + c.blu() + ")");
   }
}
```

✔ NOTA

Volendo capire cosa fa realmente il compilatore, abbiamo compilato il programma `Enumerali` *e poi lo abbiamo "decompilato" per vedere le istruzioni realmente impiegate. Il risultato è riportato nel listato seguente, nel quale abbiamo inserito*

manualmente la classe MyEnum *e l'abbiamo sostituita a* Enum *per poterla rendere compilabile: Java infatti non permette di derivare direttamente da questa classe. Da notare che sono state usate delle* classi interne *ovvero classi dichiarate all'interno di un'altra classe: questa caratteristica verrà illustrata più avanti.*

```
class MyEnum {
   private String nome;
   private int ordinale;
   protected MyEnum (String nome, int ordinale) {
      this.nome = nome;
      this.ordinale = ordinale;
   }
   public final int ordinal() {
      return ordinale;
   }
   public String toString() {
      return nome;
   }
}

public class EnumD {
   public static final class Giorno extends MyEnum {
      public static final Giorno LUNEDI;
      public static final Giorno MARTEDI;
      public static final Giorno MERCOLEDI;
      public static final Giorno GIOVEDI;
      public static final Giorno VENERDI;
      public static final Giorno SABATO;
      public static final Giorno DOMENICA;
      private static final Giorno $VALUES[];

      private Giorno(String s, int i) {
         super(s, i);
      }
      public static final Giorno[] values() {
         return (Giorno[])$VALUES.clone();
      }

      static {
         LUNEDI = new Giorno("LUNEDI", 0);
         MARTEDI = new Giorno("MARTEDI", 1);
         MERCOLEDI = new Giorno("MERCOLEDI", 2);
         GIOVEDI = new Giorno("GIOVEDI", 3);
         VENERDI = new Giorno("VENERDI", 4);
         SABATO = new Giorno("SABATO", 5);
```

```
        DOMENICA = new Giorno("DOMENICA", 6);
        $VALUES = (new Giorno[] {
            LUNEDI, MARTEDI, MERCOLEDI, GIOVEDI,
            VENERDI, SABATO, DOMENICA
        });
    }
}

public static void main(String argv[]) {
    Giorno arr$[] = Giorno.values();
    int len$ = arr$.length;
    for(int i$ = 0; i$ < len$; i$++) {
        Giorno g = arr$[i$];
        vedi(g);
    }
}

static class _cls1 {
    static final int $SwitchMap$E$Giorno[];
    static {
       $SwitchMap$E$Giorno = new int[Giorno.values().length];
        try {
            $SwitchMap$E$Giorno[Giorno.LUNEDI.ordinal()] = 1;
        } catch(NoSuchFieldError ex) { }
        try {
           $SwitchMap$E$Giorno[Giorno.MARTEDI.ordinal()] = 2;
        } catch(NoSuchFieldError ex) { }
        try {
         $SwitchMap$E$Giorno[Giorno.MERCOLEDI.ordinal()] = 3;
        } catch(NoSuchFieldError ex) { }
        try {
           $SwitchMap$E$Giorno[Giorno.GIOVEDI.ordinal()] = 4;
        } catch(NoSuchFieldError ex) { }
        try {
           $SwitchMap$E$Giorno[Giorno.VENERDI.ordinal()] = 5;
        } catch(NoSuchFieldError ex) { }
        try {
            $SwitchMap$E$Giorno[Giorno.SABATO.ordinal()] = 6;
        } catch(NoSuchFieldError ex) { }
        try {
          $SwitchMap$E$Giorno[Giorno.DOMENICA.ordinal()] = 7;
        } catch(NoSuchFieldError ex) { }
    }
}
```

```
    public static void vedi(Giorno g) {
        switch(_cls1.$SwitchMap$E$Giorno[g.ordinal()]) {
        case 1:
            System.out.println ("Lunedì=" + g);
            break;
        case 2:
            System.out.println ("Martedì=" + g);
            break;
        case 3:
            System.out.println ("Mercoledì=" + g);
            break;
        case 4:
            System.out.println ("Giovedì=" + g);
            break;
        case 5:
            System.out.println ("Venerdì=" + g);
            break;
        case 6:
            System.out.println ("Sabato=" + g);
            break;
        case 7:
            System.out.println ("Domenica=" + g);
            break;
        }
    }
}
```

Domande di verifica

1. Qual è la funzione della classe `Class`? Esiste un costruttore pubblico in `Class`? Perché?
2. Descrivere i metodi principali della classe `Object`?
3. Quali sono le due classi per la gestione delle stringhe e in quale package sono contenute?
4. Si possono confrontare due oggetti mediante l'operatore di uguaglianza? Come si fa?
5. Indicare i metodi per l'estrazione di caratteri da stringa e le conversioni di stringhe.
6. Cosa si intende per *collection framework*?
7. Spiegare l'utilità della classe `Vector`.
8. Quali strumenti sono resi disponibili per lavorare con le collezioni?

9. Perché sono utili le classi involucro? Cosa si intende per *autoboxing*?
10. Cosa possiede `Math` e in quale package è contenuta?
11. Fare un esempio di utilizzo del tipo `enum`.

Esercizi

1. Scrivere un metodo `equals` per la classe `Tempo`.
2. Rendere la classe `Tempo` clonabile.
3. Scrivere un metodo `toString` per la classe `Tempo` e visualizzare una sua istanza mediante `System.out.println`.
4. Scrivere un programma che concateni ad una stringa i primi cinque caratteri di una seconda stringa immessa dall'utente.
5. Scrivere un programma che confronti due stringhe, limitatamente ai primi cinque caratteri e, successivamente, visualizzi il risultato del confronto.
6. Scrivere un programma che data la seguente assegnazione alla stringa `esercizio`

```
esercizio:=
  '1234567890abcdefghilmopqrstvuzABCDEFGHILMNOPQRSTVUZ';
```

sposti i caratteri numerici dopo le lettere minuscole e prima delle lettere maiuscole, in modo che la stringa assuma il valore

```
abcdefghilmopqrstvuz1234567890ABCDEFGHILMNOPQRSTVUZ
```

7. Scrivere un programma che, richieste all'utente le stringhe `frase`, `parola1` e `parola2`, controlli se in `frase` è contenuta `parola1` e, in tal caso, sostituisca tutte le sue occorrenze con `parola2`.
8. Scrivere un programma che, richiesta all'utente una stringa, controlli se è palindroma. [Una stringa si dice palindroma se si legge nello stesso modo da sinistra verso destra e da destra verso sinistra. Sono esempi di stringhe palindrome: ANNA, radar, anilina].
9. Scrivere un programma che, richiesta all'utente una stringa, controlli se vi compaiono almeno due caratteri uguali consecutivi.
10. Scrivere un programma che permetta di ordinare *n* stringhe.
11. Scrivere un programma che richieda all'utente un carattere e una stringa e calcoli quindi il numero di occorrenze del carattere nella stringa.
12. Scrivere un programma che, letta una stringa composta di sole lettere maiuscole (A..Z), visualizzi il numero delle vocali e quello delle consonanti.
13. Scrivere un programma che, letta una stringa composta di sole lettere maiuscole (A..Z), visualizzi la lettera che appare più volte.

14. Scrivere un programma che, letta una stringa composta di sole cifre (0..9), visualizzi accanto ad ogni cifra il numero di volte che questa compare nella stringa.

15. Scrivere un programma che richiesta all'utente una stringa ne visualizzi una seconda ottenuta dalla prima sostituendo tutte le lettere minuscole con delle maiuscole.

16. Scrivere un programma che richiesta all'utente una stringa ne visualizzi una seconda ottenuta dalla prima sostituendo tutte le lettere maiuscole con delle minuscole e viceversa.

17. Riscrivere la classe `VettoreOrdinato` utilizzando un oggetto della classe `Vector` anziché un array.

18. Fare in modo che `VettoreOrdinato` ordini una serie di numeri in virgola mobile e li visualizzi usando un ciclo `for-each`.

19. Calcolare varie funzioni matematiche (`sen`, `cos` ecc.) dei valori contenuti nel vettore dell'esercizio precedente usando l'autoboxing.

20. Scrivere un programma che mostri i mesi dell'anno con il relativo numero di giorni utilizzando i tipi `enum`.

Capitolo 12

Gestione delle eccezioni

Obiettivi didattici

- Gestione degli errori
- Classe `Throwable`
- `try`, `catch`, `finally`
- Classi `Error` ed `Exception`
- Clausola `throws` nella dichiarazione dei metodi
- Rigetto esplicito con `throw`

Nei capitoli precedenti, si è parlato spesso di errori che possono avvenire durante la fase di esecuzione del programma. Questi errori, detti a *run-time*, avvengono quando un'istruzione pur essendo scritta correttamente, non è in grado di svolgere il proprio compito per un qualche motivo contingente. Nella maggior parte dei linguaggi normalmente esistono due tipi di errore: il primo tipo raggruppa tutti quelli che possono avvenire anche in situazioni del tutto normali, come per esempio trovare la fine di un file durante la sua lettura, mentre il secondo raggruppa tutti quelli che sono sintomo di una situazione sicuramente anormale, come per esempio l'accesso a un elemento di un array con un indice minore di 0.

Gli errori del primo tipo vengono segnalati di solito per mezzo di un codice restituito al programma e il programmatore ha la facoltà di gestire l'errore oppure di ignorarlo. Gli errori del secondo tipo invece causano, nel migliore dei casi, la terminazione immediata dell'esecuzione mentre, nel caso peggiore, possono creare delle anomalie nel funzionamento del programma di difficile individuazione. Dato che Java ha l'obiettivo di essere un linguaggio portabile e robusto, è stato necessario trovare un sistema

migliore per la gestione degli errori, tale da permettere un trattamento omogeneo sia degli errori di sistema come di quelli definiti a livello di programma e da integrarsi adeguatamente con la metodologia a oggetti.

In un programma Java ogni volta che un'istruzione non porta a termine il proprio compito, viene creato un oggetto di una classe derivata dalla classe `Throwable` contenente delle informazioni riguardo il tipo di problema che non ha permesso l'esecuzione corretta. Ci si riferisce a questi oggetti col termine di *eccezioni*. Si usa "eccezione" al posto di "errore" in quanto quest'ultimo termine tende a indicare uno sbaglio da parte del programmatore, mentre il termine eccezione abbraccia tutti quegli eventi che non dovrebbero accadere in una situazione normale, eventi appunto eccezionali. Si dice che viene *gettata* (thrown) un'eccezione in quanto essa deve essere *catturata* (catch) da un brano di codice del programma affinché la situazione eccezionale sia gestita correttamente. Se il programmatore non ha previsto nessun brano di codice per la gestione dell'eccezione, l'interprete blocca l'esecuzione del programma con un messaggio opportuno.

12.1 Istruzioni `try` e `catch`

Quando ci si aspetta che un brano di codice possa generare un'eccezione, esso va racchiuso in un blocco speciale, detto blocco `try`. Dopo un blocco di questo tipo devono essere sempre definiti uno o più blocchi `catch` e, opzionalmente, un blocco `finally`. La forma generale del costrutto `try-catch-finally` è quindi la seguente:

```
try {
    try-istruzione-1;
    [try-istruzione-2; ]
     ...
    [try-istruzione-N; ]
}
[catch (Classe-eccezione1 e1) {
    [catch1-istruzione-1; ]
     ...
    [catch1-istruzione-N; ]
}
catch (Classe-eccezione2 e2) {
    [catch2-istruzione-1; ]
     ...
    [catch2-istruzione-N; ]
}


catch (Classe-eccezioneN eN) {
    [catchN-istruzione-1; ]
```

```
    [catchN-istruzione-N; ]
} ]
[finally {
    [finally-istruzione-1; ]
     ...
    [finally-istruzione-N; ]
}]
```

Un blocco `try` deve essere seguito sempre almeno da un blocco `catch` oppure da un blocco `finally`. Nel caso in cui sia definito almeno un blocco `catch` il blocco `finally` è opzionale. Vediamo dunque come interagiscono questi blocchi.

Quando un'istruzione all'interno di un blocco `try` fallisce, viene creata un'eccezione, cioè un oggetto di una classe derivata dalla classe `Throwable` dipendente dal tipo di problema che ha causato il fallimento. Per esempio una divisione tra interi in cui il divisore è uguale a zero, provocherà la creazione di un oggetto della classe `ArithmeticException`, mentre l'accesso a un array con indice minore di 0 provocherà la creazione di un oggetto della classe `ArrayIndexOutOfBoundsException`.

Al momento della creazione dell'eccezione, l'esecuzione del codice viene interrotta e l'interprete controlla se esiste un blocco `catch` che abbia un argomento corrispondente al tipo di eccezione gettata. Il blocco `catch` ha una struttura simile a quella di un metodo, il cui argomento deve appartenere alla classe dell'eccezione gettata oppure a una sua superclasse. Se l'interprete trova un blocco `catch` il cui argomento è compatibile con l'eccezione, esegue il codice racchiuso tra le sue parentesi graffe, altrimenti considera l'eccezione non catturata.

Quando un'eccezione non viene catturata, il metodo viene terminato immediatamente come se fosse eseguito un `return`, tramandando però l'eccezione al metodo chiamante, il quale può catturare a sua volta l'eccezione o ignorarla con un meccanismo analogo a quello visto. Se non viene trovato nessun blocco `catch` adatto alla gestione dell'eccezione, l'interprete blocca l'esecuzione del programma segnalando un opportuno messaggio d'errore. Per chiarire questo meccanismo, osserviamo l'esempio seguente

```
class Eccezioni
{
  int array1[] = { 100, 200 };
  int array2[];

  int metodo1 (int arr[], int ind, int div) {
    int Return = 0;
    try {
       Return = arr[ind] / div;
    } catch (ArithmeticException e) {
     System.out.println ("metodo1: Eccezione aritmetica:" + e);
    }
    System.out.println ("Fine metodo1");
    return Return;
  }
```

```
  public static void main (String argv[]) {
    Eccezioni ecc = new Eccezioni();

    try {
       ecc.metodo1(ecc.array1, 0, 0);
    } catch (ArrayIndexOutOfBoundsException e) {
       System.out.println ("Eccezione indice array:" + e);
    } catch (ArithmeticException e) {
       System.out.println ("main: Eccezione aritmetica:" + e);
    }
    try {
       ecc.metodo1(ecc.array1, -1, 1);
    } catch (ArrayIndexOutOfBoundsException e) {
       System.out.println ("main: Eccezione indice array:" + e);
    } catch (ArithmeticException e) {
       System.out.println ("main: Eccezione aritmetica:" + e);
    }
    try {
       ecc.metodo1(ecc.array2, 1, 1);
    } catch (ArrayIndexOutOfBoundsException e) {
       System.out.println ("main: Eccezione indice array:" + e);
    } catch (ArithmeticException e) {
       System.out.println ("main: Eccezione aritmetica:" + e);
    }
    System.out.println ("Fine main");
  }
}
```

La classe `Eccezioni` ha due variabili d'istanza, la prima delle quali è un array d'interi inizializzato con due elementi, mentre la seconda è un array d'interi non inizializzato. Essa contiene un metodo d'istanza, `metodo1`, che ha tre parametri e precisamente un array d'interi `arr`, un intero `ind` e un intero `div`. Il metodo divide l'elemento a indice `ind` di `arr` per `div`. Questa istruzione è inserita in un blocco `try` a cui segue un solo blocco `catch` dove viene catturata l'eccezione di tipo `ArithmeticException`, che non fa altro che visualizzare un messaggio. Nella classe è presente anche il metodo di classe `main` che crea un'istanza della classe `Eccezioni` e invoca tre volte il suo metodo `metodo1` con argomenti differenti.

Ogni invocazione è racchiusa in un blocco `try` cui seguono due blocchi `catch` dove vengono catturate le eccezioni di tipo `ArrayIndexOutOfBoundsException` e `ArithmeticException` che, anche in questo caso, non fanno altro che visualizzare un messaggio. La prima volta che viene invocato il metodo `metodo1`, viene passato un valore 0 per il parametro `div`. Questo valore provoca un'eccezione nel blocco `try` che viene catturata dal blocco `catch` seguente, il quale esegue la `println` che si trova al suo interno. L'esecuzione continua dopo il blocco e viene così eseguita anche la `println` che si trova prima dell'istruzione `return`. Il controllo ritorna al metodo `main` che invoca di nuovo il metodo `metodo1`, questa volta con il parametro `ind`

uguale a -1. In questo caso viene generata un'eccezione di tipo differente che però non trova nessun blocco `catch` adatto al suo trattamento tra quelli che seguono il blocco `try` in cui si è verificata. Per questo motivo l'esecuzione del metodo viene bloccata e il controllo viene restituito immediatamente al metodo `main`.

Poiché l'invocazione di `metodo1` è contenuta in un blocco `try` e poiché uno dei blocchi `catch` che la seguono è adatto al trattamento dell'eccezione provocata dal metodo invocato, vengono eseguite le istruzioni di quel blocco. Dopo la visualizzazione, l'esecuzione prosegue dopo l'ultimo blocco `catch` e si passa quindi alla terza invocazione di `metodo1`. In questo caso viene passato come argomento un array che non è stato inizializzato, per cui viene generata un'eccezione di tipo `NullPointerException` che non viene catturata né nel metodo `metodo1` né nel metodo `main`. Allora l'interprete blocca l'esecuzione del programma con un messaggio opportuno e l'ultima `println` non viene mai eseguita. In definitiva quindi il programma precedente produce la seguente visualizzazione.

```
metodo1: Eccezione aritmetica:java.lang.ArithmeticException:
/ by zero
Fine metodo1
main: Eccezione indice
array:java.lang.ArrayIndexOutOfBoundsException: -1
Exception in thread "main" java.lang.NullPointerException
        at Eccezioni.metodo1(Eccezioni.java:9)
        at Eccezioni.main(Eccezioni.java:35)
```

Questo modo di gestire le eccezioni è estremamente pulito ed evita il rischio di avere nell'interprete uno stato inconsistente, poiché ogni istruzione che provoca un'eccezione risulta a tutti gli effetti come non eseguita e non genera nessun effetto collaterale.

Va sottolineato che le istruzioni del blocco `catch` sono al di fuori del blocco `try` a cui si riferiscono per cui, se si prevede che una di esse possa generare un'eccezione, essa va inserita in un nuovo blocco `try`. Per esempio, possiamo modificare il blocco `catch` incluso nel metodo `metodo1` nel modo seguente.

```
} catch (ArithmeticException e) {
   System.out.println ("metodo1: Eccezione aritmetica:" + e
   System.out.println ("Premi return per continuare");
   try {
      System.in.read();
   } catch (Exception e2) {  }
}
```

Notare che è stato necessario dare un nome diverso all'eccezione dichiarata come argomento della seconda `catch`, poiché in quel punto è visibile ancora l'argomento della prima `catch`, se avessimo usato lo stesso nome avremmo avuto collisione di nomi.

Il costrutto `try-catch-finally` è implementato sullo stack, in modo analogo a come vengono richiamati i metodi e gestite le istruzioni composte, per cui non solo è possibile avere dei blocchi `catch` annidati, come in questo caso, ma anche dei blocchi `try` annidati.

12.2 finally

Come abbiamo visto, un'eccezione può provocare una modifica del flusso del programma in modi anche difficilmente prevedibili. In alcune situazioni è necessario assicurarsi che certe istruzioni vengano eseguite in tutti i casi, indipendentemente dal fatto che si verifichino eccezioni e di che tipo. Ciò risulta utile quando si allocano delle risorse che è opportuno liberare prima dell'uscita, oppure si aprono dei file all'inizio del metodo.

Il blocco `finally` supplisce a tale necessità. Esso può essere posto direttamente dopo un blocco `try` oppure dopo l'ultimo blocco `catch` e, se il flusso arriva al blocco `try` relativo, le istruzioni in esso contenute vengono eseguite in tutti i casi, anche se nel blocco `try` o in uno dei blocchi `catch` precedenti compare l'istruzione `return`. Modificando nell'esempio precedente il metodo `metodo1`

```
int metodo1 (int arr[], int ind, int div) {
  int Return = 0;
  try {
    Return = arr[ind] / div;
  } catch (ArithmeticException e) {
    System.out.println ("metodo1: Eccezione aritmetica:" + e);
  } finally {
    System.out.println ("metodo1:finally 1");
  }

  try {
    return Return;
  } finally {
    System.out.println ("Fine metodo1");
  }
}
```

il programma esegue la seguente visualizzazione

```
metodo1: Eccezione aritmetica:java.lang.ArithmeticException:
/ by zero
metodo1:finally 1
Fine metodo1
metodo1:finally 1
main: Eccezione indice
array:java.lang.ArrayIndexOutOfBoundsException: -1
metodo1:finally 1
Exception in thread "main" java.lang.NullPointerException
        at Eccezioni.metodo1(Eccezioni.java:9)
        at Eccezioni.main(Eccezioni.java:41)
```

Come si vede, il primo blocco `finally` viene eseguito tre volte, tante quante viene invocato il metodo `metodo1`, mentre il secondo blocco `finally` viene eseguito una sola volta, nell'unico caso in cui il flusso del programma arriva al secondo blocco `try` e nonostante esso contenga un'istruzione `return`.

12.3 Tipi di eccezioni

Abbiamo detto che un'eccezione è un oggetto che appartiene a una classe derivata dalla classe `Throwable`, che a sua volta deriva da `Object`, e appartiene al package `java.lang`. Dalla classe `Throwable` derivano le classi `Error` ed `Exception`. Le classi che derivano da `Error` sono usate per la segnalazione di problemi cui un programmatore non può porre rimedio, come un errore interno dell'interprete o la mancanza di memoria centrale. Le classi che derivano da `Exception` viceversa sono usate per segnalare problemi che possono essere gestiti dal programmatore. Questa distinzione è stata fatta affinché sia possibile catturare un'eccezione qualsiasi, senza preoccuparsi del suo tipo, con un'istruzione come la seguente

```
catch (Exception e)
```

Le eccezioni derivanti da costrutti del linguaggio utilizzati in modo scorretto, come quelle viste negli esempi precedenti, derivano tutte dalla classe `RuntimeException` che a sua volta deriva da `Exception`. Il programmatore può derivare le eccezioni di cui necessita da qualsiasi classe di eccezioni ritiene opportuno e nel seguito vedremo come fare per gettarle.

Oltre alla suddivisione delle eccezioni in base alla gerarchia delle classi, esse sono distinte in due tipi: le eccezioni *controllate* (checked) e quelle *non controllate* (unchecked). Quando un'istruzione può provocare un'eccezione di tipo controllato, il compilatore si assicura che essa sia inclusa in un blocco `try` e che sia definito un blocco `catch` adatto alla sua gestione, altrimenti segnala un errore in compilazione. Abbiamo sperimentato questa situazione quando abbiamo introdotto il metodo `System.in.read`. Le istruzioni che possono provocare un'eccezione non controllata possono trovarsi anche fuori di un blocco `try` senza che il compilatore segnali alcun problema. Appartengono al gruppo delle eccezioni non controllate tutte quelle che derivano dalla classe `Error` e dalla classe `RuntimeException`, mentre tutte le altre appartengono al gruppo delle eccezioni controllate.

Il controllo della gestione delle eccezioni da parte del compilatore obbliga il programmatore a tenere conto degli errori che possono verificarsi in un modo o in un altro. D'altra parte per non infarcire i programmi di blocchi `try` inutili, che renderebbero più complessa l'individuazione dei blocchi `try` utili, sono state tolte da questo meccanismo di controllo le eccezioni che riguardano le istruzioni più comuni e quelle che non potrebbero essere gestite in alcun caso.

12.4 throws

Nel capitolo dedicato a una prima presentazione dei metodi, abbiamo introdotto un metodo `leggi` utile per leggere dati in modo generico dal terminale (Cfr. Paragrafo 7.4).

```
static int leggi(boolean num) {
   byte b[] = new byte[9];
   int Return = 0;
   try {
      System.in.read(b);
   } catch (Exception e) { }
   if (num)
      for (int i = 0; i < b.length; i++)
         if (b[i] >= '0' && b[i] <= '9')
            Return = Return * 10 + b[i] - '0';
         else
            break;
   else
      if (b[0] >= ' ')
         Return = b[0];
   return Return;
}
```

In `leggi` viene invocato `System.in.read` che è incluso nelle librerie di base e legge un array di `byte` appunto dal terminale; esso però può provocare un'eccezione `IOException`, che è di tipo controllato, per cui abbiamo dovuto inserire l'invocazione all'interno di un blocco `try`. Il metodo `leggi` ha lo scopo di sostituire la chiamata diretta di `System.in.read` in modo da evitare di interpretare ogni volta il contenuto dell'array di byte letto. Nell'implementazione riportata però, quando un programma invoca `leggi` non si ha la possibilità di sapere se la lettura sia andata a buon fine o meno. Visto che `leggi` sostituisce `System.in.read`, esso dovrebbe essere in grado di gettare le eccezioni `IOException`, in modo tale che il metodo che lo invoca sia consapevole di eventuali problemi. Lo si ottiene tramite la clausola `throws` posta dietro la dichiarazione di un metodo. La forma generale è

```
dichiarazione-metodo (lista-argomenti) throws Classe-eccezione1
                                            [,Classe-eccezione2
                                             ...
                                            [,Classe-eccezioneN]]
```

L'istruzione `throws` posta dopo la dichiarazione di un metodo dichiara al compilatore che esso è in grado di gettare eccezioni di uno o più tipi; in questo caso non è più obbligatorio inserire le istruzioni che possono generare eccezioni dei tipi dichiarati dopo la `throws` in blocchi `try`, in quanto la responsabilità della gestione di tali eccezioni è trasferita al metodo chiamante. L'esempio citato potrebbe essere quindi modificato nel modo seguente (Cfr. Paragrafo 7.4):

```
import java.io.*;

class Input
{
   public static void main (String argv[]) {
      int inp = 0;
      do {
         try {
            inp = leggi(true);
         } catch (Exception e) { }
         System.out.print ("hai inserito ");
         System.out.println (inp);
      } while (inp != 0);
      do {
         try {
            inp = leggi(false);
         } catch (Exception e) { }
         System.out.print ("hai inserito ");
         System.out.println ((char)inp);
      } while (inp != 0);
   }

   static int leggi(boolean num) throws IOException {
      byte b[] = new byte[9];
      int Return = 0;
      System.in.read(b);
      if (num)
         for (int i = 0; i < b.length; i++)
            if (b[i] >= '0' && b[i] <= '9')
               Return = Return * 10 + b[i] - '0';
            else
               break;
      else
         if (b[0] >= ' ')
            Return = b[0];
      return Return;
   }
}
```

Nell'esempio abbiamo messo in grado il metodo `leggi` di gettare eccezioni del tipo `IOException`. Poiché questo tipo di eccezioni è definito nel package `java.io`, abbiamo dovuto inserire un'istruzione di `import` opportuna. Avremmo potuto usare la classe `Exception` invece di `IOException` dopo la clausola `throws`, ma in questo modo avremmo rischiato di lasciar "rimbalzare" eccezioni da gestire localmente e avremmo fornito meno informazioni sul tipo di problemi che potrebbero derivare dall'utilizzo del metodo a chi ne legge il listato. L'utilizzo della clausola `throws` ha

permesso di richiamare `System.in.read` senza includerlo in un blocco `try`, ma obbliga l'inserimento dell'invocazione del metodo `leggi` in un blocco `try`, a meno che il metodo da cui è richiamato non abbia a sua volta una clausola `throws`. Potremmo quindi eliminare tutti i blocchi `try` riscrivendo il metodo `main` dell'esempio nel modo seguente.

```
public static void main (String argv[]) throws Exception {
   int inp = 0;
   do {
      inp = leggi(true);
      System.out.print ("hai inserito ");
      System.out.println (inp);
   } while (inp != 0);
   do {
      inp = leggi(false);
      System.out.print ("hai inserito ");
      System.out.println ((char)inp);
   } while (inp != 0);
}
```

12.5 throw

Nell'esempio precedente abbiamo visto che il metodo `leggi` più che gettare un'eccezione, la fa *rimbalzare*, in quanto l'eccezione viene gettata da un metodo contenuto. In questi casi si dice che l'eccezione è gettata in modo *implicito*. Un metodo però può gettare eccezioni in modo *esplicito* per mezzo dell'istruzione `throw` la cui forma generale è

`throw` *`oggetto-eccezione;`*

dove `oggetto-eccezione` deve essere un oggetto appartenente a una classe derivata da `Throwable`. Con l'istruzione `throw` si getta un'eccezione che segue le regole generali viste fino a ora. Questo significa che la precedente istruzione può essere lanciata o dall'interno di un blocco `try` seguito da un blocco `catch` adatto al trattamento della classe cui appartiene `oggetto-eccezione`, oppure all'interno di un metodo che ha specificato la classe cui appartiene `oggetto-eccezione` dopo una clausola `throws`. Bisogna prestare attenzione che le parole riservate `throw` e `throws` differiscono solo per la esse finale, ma hanno significati assai diversi.

12.6 Un esempio riepilogativo

Riprendiamo ora l'esempio della classe `Tempo` che ci ha accompagnato durante tutto il Capitolo 9. In essa sono definiti un costruttore e un metodo, `assegnaTempo`, che hanno come argomenti tre interi che rappresentano rispettivamente le ore, i minuti e i secondi. Nel caso del metodo, esso restituisce un valore 0 se i tre argomenti indicano

un tempo valido, mentre restituisce -1 nel caso i tre argomenti non siano congruenti con la definizione di tempo. Il metodo chiamante quindi può verificare attraverso il codice restituito se l'operazione è andata a buon fine o meno. Questo sistema però non è applicabile al caso del costruttore, in quanto esso restituisce per definizione solo la nuova istanza creata. In effetti all'interno del costruttore viene richiamato il metodo `assegnaTempo` senza neppure controllarne il codice restituito. Ora che conosciamo il meccanismo delle eccezioni di Java possiamo fare di meglio.

Per prima cosa deriviamo da `Exception` una classe che rappresenti le eccezioni della classe `Tempo`. Per il momento limitiamoci all'esempio più semplice, scrivendo nel file `TempoException.java` la classe seguente

```
public class TempoException extends Exception
{
}
```

A questo punto possiamo modificare la classe `Tempo`.

```
public class Tempo
{
   protected int ore;
   protected int minuti;
   protected int secondi;
   static public char separatore;

   public Tempo () {
      ore = minuti = secondi = 0;
   }
   public Tempo (int ora, int minuto, int secondo)
                        throws TempoException {
      this();
      assegnaTempo (ora, minuto, secondo);
   }
   public void assegnaTempo (int ora, int minuto, int secondo)
                        throws TempoException  {
      if (ora >= 0 && ora < 24 &&
          minuto >= 0 && minuto < 60 &&
          secondo >= 0 && secondo < 60) {
         ore = ora;
         minuti = minuto;
         secondi = secondo;
      } else
         throw new TempoException();
   }
   /* ... */
   /* Altri metodi della classe Tempo */
   /* ... */
}
```

Come si vede, il metodo `assegnaTempo` in questo caso non restituisce più un codice, ma invece getta un'eccezione mentre il costruttore con tre argomenti si limita a rimbalzare questa eccezione. Si ottiene così un comportamento analogo sia per il costruttore che per il metodo. Possiamo verificarne il comportamento per mezzo della classe seguente.

```
public class EccTempo
{
   public static void main (String argv[]) {
      try {
        Tempo t = new Tempo (99,99,99);
      } catch (TempoException e) {
         System.out.println ("Errore: " + e);
      }
   }
}
```

Come si vede, abbiamo dovuto inserire l'invocazione del costruttore con tre argomenti in un blocco `try`, altrimenti il compilatore avrebbe segnalato un errore. Eseguendo la classe precedente, otteniamo la seguente visualizzazione

```
Errore: TempoException
```

poiché il costruttore è stato invocato con tre numeri che non rappresentano un tempo. Possiamo ottenere dei risultati ancora migliori modificando di poco la classe `TempoException`, così per esempio:

```
public class TempoException extends Exception
{
   String msg;
   TempoException () {
      msg = "";
   }
   TempoException (String msg) {
      this.msg = msg;
   }
   public String toString () {
      return "Errore nella classe Tempo: " + msg;
   }
}
```

In questa nuova versione abbiamo inserito un nuovo costruttore che consente di memorizzare un messaggio nella variabile d'istanza `msg`. Abbiamo poi ridefinito il metodo `toString`, che fa parte della classe `Object`, in modo tale da far visualizzare un messaggio personalizzato contenente anche `msg`. Con questa nuova classe, possiamo modificare l'istruzione `throw` contenuta nella classe `Tempo` nel modo seguente:

```
throw new TempoException("argomenti incongruenti "
                                + ora + ","
                                + minuto + ","
                                + secondo);
```

Ricompilando le classi e rilanciando il programma si ottiene ora la seguente visualizzazione

```
Errore: Errore nella classe Tempo: argomenti incongruenti
99,99,99
```

che sicuramente dà maggiori indicazioni riguardo il tipo di problema verificatosi. Non c'è limite alla quantità di informazioni che si possono aggiungere a un'eccezione in quanto essa altro non è che un oggetto.

Domande di verifica

1. Qual è il significato di *gettare* e *catturare* un'eccezione?
2. A cosa è utile il costrutto `try-catch-finally`? Qual è la sua sintassi?
3. A quale necessità supplisce il blocco `finally`?
4. Da quale classe derivano e a cosa sono utili rispettivamente `Error` ed `Exception`?
5. Come si può fare in modo che un metodo sia in grado di accettare eccezioni di uno o più tipi?
6. Cose si può rigettare un'eccezione in modo esplicito?

Esercizi

1. Il metodo `Math.sqrt(double d)` non getta un'eccezione nel caso `d` abbia un valore negativo, ma invece restituisce `NaN`. Scrivere una propria versione di `sqrt` che getti un'eccezione `ArithmeticException` nel caso venga invocata con argomento negativo.
2. Scrivere un metodo che restituisce `true` se l'argomento passato appartiene a una classe clonabile e `false` altrimenti, utilizzando il meccanismo delle eccezioni.
3. Fare in modo che la classe `TempoException` sia un'eccezione non controllata, anziché un'eccezione controllata.
4. Riprendere l'esempio riepilogativo del Capitolo 6 e riscriverlo utilizzando il meccanismo delle eccezioni dovunque sia opportuno.
5. Utilizzare il meccanismo delle eccezioni per provare che gli argomenti di un metodo vengono valutati da sinistra verso destra.

Capitolo 13

Thread

Obiettivi didattici

- Esecuzione di parti di programma in forma indipendente
- Attributi e stati dei thread
- Istanziare e attivare i thread
- Sincronizzazione
- Modificatore volatile
- Comunicazioni tra thread

In tutti gli esempi visti fino a ora, era presente un flusso di esecuzione che iniziava dopo la definizione del metodo `main` e che, dopo essere passato attraverso un certo numero istruzioni e metodi, si arrestava alla fine del metodo di partenza. Possiamo pensare al flusso dell'esecuzione di un programma come a un filo che lega tutte le istruzioni che vengono eseguite in sequenza dall'attivazione del metodo `main` fino alla sua terminazione.

Si intuisce come in certe situazioni possa risultare comodo poter attivare dei metodi in modo asincrono rispetto al flusso di esecuzione. Supponiamo per esempio di voler visualizzare l'ora in tempo reale durante l'esecuzione di un programma. La realizzazione diventa abbastanza semplice avendo a disposizione un sistema per attivare un metodo il cui codice sia eseguito in modo indipendente dal resto del programma e che quindi visualizza l'ora mentre l'elaborazione segue il suo corso. Questa modalità di attivazione corrisponde alla creazione di un nuovo filo che fa la sua comparsa all'inizio del metodo e unisce tutte le istruzioni fino al termine del metodo stesso, lasciando

che il filo principale segua il suo corso indisturbato. In inglese la parola 'filo' si traduce con 'thread' e un programma che esegue alcune sue parti in modo indipendente si dice *multithread*. Possiamo dunque dire che tutti gli esempi visti fino a ora erano a thread *singolo*.

Il multithreading consente di ampliare il paradigma concettuale della programmazione, di affrontare in modo migliore l'interazione utente-computer e di poter sfruttare al massimo le architetture hardware più avanzate con CPU multiple. Tutti i moderni sistemi operativi permettono l'esecuzione multithread, ma spesso la realizzazione dei programmi è complicata dall'inadeguatezza dei linguaggi comunemente usati, nati prima del concetto di multithread.

Java è un linguaggio in cui il multithread è stato previsto a livello di progettazione, dunque include costrutti e istruzioni tali da semplificare quanto possibile la gestione di programmi che ne facciano uso. Dividere il codice in moduli che girano in modo indipendente, infatti, offre possibilità irraggiungibili da programmi a singolo thread, ma complica anche la programmazione e la manutenzione del software.

Quando la Java Virtual Machine ha più thread attivi contemporaneamente, essa termina tutte le proprie attività nel momento in cui tutti i thread non *demoni* (*daemon*) terminano, oppure quando viene invocato il metodo `exit()` della classe `Runtime` o della classe `System` che vedremo in seguito. Un thread demone si distingue da quelli non demoni proprio in relazione al comportamento appena descritto della Virtual Machine e permette di avere thread che eseguono un codice ciclico all'infinito, come per esempio un'animazione, e la cui terminazione è indotta dalla terminazione degli altri thread.

In Java un thread ha i seguenti attributi:

- Il nome, che è semplicemente una stringa a scopo informativo; due o più thread possono avere lo stesso nome, senza che ciò causi il minimo problema;
- un *thread group* di appartenenza. Un thread group è un insieme di thread e thread group. Ogni thread appartiene a un solo thread group e tutti i thread group meno uno, detto "main thread group", appartengono a qualche altro thread group. In pratica quindi i thread group possono formare una struttura ad albero di cui il system thread group rappresenta la radice. I thread group forniscono un sistema per imporre dei livelli di sicurezza sui programmi in esecuzione e sono usati solo a livello di sistema;
- una *priorità* di esecuzione che in Java è semplicemente un numero compreso tra 1 e 10. Una priorità più alta significa che il thread deve essere eseguito preferibilmente a un thread con una priorità più bassa. Questo permette di privilegiare certi thread che debbono interagire con l'utente o con altre risorse in modo immediato rispetto a thread che effettuano compiti per i quali non è richiesta un'interattività immediata, come per esempio la stampa di un documento. Questa preferenza è però solo indicativa e l'utilizzo di una certa priorità può dar luogo a comportamenti diversi su sistemi diversi;
- se è un thread demone o meno.

Oltre a questi attributi, un thread ha anche uno *stato* che può essere uno dei seguenti:

- attivo (in inglese *alive*, cioè 'vivo') o inattivo: un thread è attivo se ne è stata lanciata l'esecuzione e la sua elaborazione non è stata ancora terminata, inattivo altrimenti;
- sospeso o non sospeso: un thread è sospeso se per mezzo di un apposito metodo ha bloccato temporaneamente l'esecuzione. Il thread può essere riattivato tramite un metodo opportuno. Questo stato ha senso solo per i thread attivi.

Ora che abbiamo visto le caratteristiche dei thread in Java, vediamo quali sono gli strumenti che abbiamo a disposizione per gestirli.

13.1 I thread in Java

In Java un thread è rappresentato da un oggetto della classe `Thread`. È possibile ottenere l'istanza del thread corrente per mezzo del metodo statico `Thread.currentThread()`. Per attivare un nuovo thread esistono due possibilità.

La prima consiste nel creare un'istanza di un oggetto di tipo `Thread` passandogli come argomento un'istanza di una classe che implementi l'interfaccia `Runnable`. Questa interfaccia definisce semplicemente il metodo `public void run()` che quindi deve essere definito nella classe argomento del costruttore di `Thread`. Appena si invoca il metodo `start()` sull'oggetto della classe `Thread`, viene attivato in modo asincrono il suddetto metodo `run`. Vediamo un esempio

```
public class ProvaThread implements Runnable
{
   public static void main (String argv[]) {
      System.out.println ("Thread corrente: " +
                          Thread.currentThread());
      ProvaThread pt = new ProvaThread();
      Thread t = new Thread(pt);
      t.start ();
      try {
         Thread.sleep (2000);
      } catch (InterruptedException e) {
      }
      System.out.println ("Fine main");
   }
   public void run () {
      System.out.println ("Thread run: " +
                          Thread.currentThread());
      for (int i = 0; i < 5; i++) {
         System.out.println (i);
         try {
            Thread.currentThread().sleep (1000);
```

```
            } catch (InterruptedException e) {
            }
        }
        System.out.println ("Fine run");
    }
}
```

Abbiamo dovuto dichiarare la classe `ProvaThread` che implementa la classe `Runnable` e definire di conseguenza il metodo `run`. Nel metodo `main` per prima cosa viene visualizzato il thread corrente, dopodiché viene creata un'istanza della classe `ProvaThread` usata come argomento per la creazione di un'istanza di `Thread`. A questo punto parte il nuovo thread tramite l'invocazione del metodo `start`. Per rendere visibile la contemporaneità dell'esecuzione, abbiamo usato il metodo statico `sleep` di `Thread` che sospende l'esecuzione del thread corrente per il numero di millisecondi specificato come argomento. Questo metodo va incluso in una `try` perché esso può gettare l'eccezione `InterruptedException` nel caso che il thread corrente venga interrotto da un altro thread. Dopo un'attesa di due secondi quindi viene visualizzato il messaggio "Fine main" e il metodo `main` termina.

Anche nel metodo `run` è presente la visualizzazione del thread corrente, seguita da un ciclo che visualizza i numeri da 0 a 4 con un intervallo di un secondo e dalla visualizzazione del messaggio "Fine run". Lanciando questo esempio si ottiene il seguente risultato

```
Thread corrente: Thread[main,5,main]
Thread run: Thread[Thread-0,5,main]
0
1
Fine main
2
3
4
Fine run
```

Gli argomenti tra parentesi quadre che si ottengono dalla visualizzazione di un oggetto della classe `Thread` sono rispettivamente il nome, la priorità e il nome del thread group.

Il secondo modo per attivare un thread consiste nel derivare una propria classe da `Thread` e ridefinire il metodo `run` con il codice che si intende far eseguire in modo asincrono. Il metodo verrà eseguito non appena si invoca il metodo `start` su un'istanza della classe derivata. Vediamo l'esempio precedente implementato utilizzando questa modalità

```
class MioThread extends Thread
{
    public void run () {
        System.out.println ("Thread run: " +
                            Thread.currentThread());
```

```
        for (int i = 0; i < 5; i++) {
            System.out.println (i);
            try {
                Thread.currentThread().sleep (1000);
            } catch (InterruptedException e) {
            }
        }
        System.out.println ("Fine run");
    }
}

public class ProvaThread
{
    public static void main (String argv[]) {
        System.out.println ("Thread corrente: " +
                            Thread.currentThread());
        MioThread t = new MioThread();
        t.start ();
        try { Thread.sleep (2000);
        } catch (InterruptedException e) {
        }
        System.out.println ("Fine main");
    }
}
```

Questi due modi possono essere usati indifferentemente a seconda delle necessità, anche se il primo è da preferirsi in quanto non interferisce con la gerarchia delle classi. Vediamo ora altri metodi comunemente usati della classe `Thread`.

- `public final String getName()`: restituisce il nome del thread.
- `public final void setName(String name)`: permette di assegnare un nome al thread.
- `public final int getPriority()`: restituisce un numero compreso tra 1 (`Thread.MIN_PRIORITY`) e 10 (`Thread.MAX_PRIORITY`) che indica la priorità del thread. 5 (`Thread.NORM_PRIORITY`) è il valore di default.
- `public final void setPriority(int newPriority)`: permette di assegnare una nuova priorità al thread compresa tra `Thread.MIN_PRIORITY` e `Thread.MAX_PRIORITY`.
- `public final boolean isDaemon()`: restituisce `true` se il thread è demone, `false` altrimenti.
- `public final void setDaemon(boolean on)`: se `on` vale `true`, il thread viene impostato demone, altrimenti se `on` vale `false`, il thread viene impostato non demone.

- `public final boolean isAlive()`: restituisce `true` se il thread è attivo, `false` altrimenti.
- `public void interrupt()`: invia una richiesta di interruzione, ma il thread non è detto che reagisca immediatamente. Se il thread è in stato di attesa, viene risvegliato e viene gettata un'eccezione `InterruptedException`.
- `public boolean isInterrupted()`: restituisce `true` se al thread è stata inviata una richiesta d'interruzione, `false` altrimenti.
- `public static boolean interrupted()`: restituisce `true` se al thread corrente è stata inviata una richiesta d'interruzione, `false` altrimenti.

13.2 La sincronizzazione

La potenza del multithreading risiede nel fatto che parti di uno stesso programma girino in modo indipendente uno dall'altro, ma a volte è necessario che certe operazioni vengano eseguite serialmente, vale a dire da un solo thread per volta. Se, per esempio, due o più thread accedono contemporaneamente a un set di variabili correlate oppure a una stessa risorsa del sistema, come un file, una stampante o una connessione di rete, i risultati possono essere imprevedibili.

Dobbiamo avere quindi a disposizione degli strumenti che permettano di eseguire certe sezioni di codice, sotto determinate condizioni, a non più di un thread alla volta. In altre parole a volte è necessario *sincronizzare* i thread.

Questo non è un problema nuovo, né tanto meno è caratteristico di Java, ma riguarda tutti i linguaggi che possono avere più thread attivi contemporaneamente. Per affrontare il problema della sincronizzazione dei thread si può ricorrere al concetto di *mutex*. Un mutex (contrazione di *mu*tual *ex*clusion) è una risorsa del sistema che può essere 'posseduta' da un solo thread alla volta. Per fare un paragone pratico, un mutex è un po' come la chiave di una toilette che in alcuni luoghi pubblici viene tenuta alla cassa. Quando una persona necessita di usare la toilette, si rivolge alla cassa e richiede la chiave, che sicuramente gli viene concessa se è disponibile. Se una seconda persona richiede la chiave mentre la toilette è ancora occupata, deve attendere che la chiave sia restituita alla cassa prima di poterla ottenere.

In Java tale meccanismo è stato automatizzato quanto più possibile; ogni istanza di qualsiasi oggetto ha associato un mutex. Quando un thread esegue un metodo che è stato dichiarato *sincronizzato* mediante il modificatore `synchronized`, entra in possesso del mutex associato all'istanza e nessun altro metodo sincronizzato può essere eseguito su quell'istanza fintanto che il thread non ha terminato l'esecuzione del metodo. Nell'esempio che segue

```
public class ProvaThread2 implements Runnable
{
   public static void main (String argv[]) {
      ProvaThread2 pt = new ProvaThread2();
      Thread t = new Thread (pt);
```

```
        t.start ();
        pt.m2();
    }
    public void run() {
        m1();
    }
    synchronized void m1 () {
        for (char c = 'A'; c < 'F'; c++) {
            System.out.println (c);
            try {
                Thread.sleep (1000);
            } catch (InterruptedException e) { }
        }
    }
    void m2 () {
        for (char c = '1'; c < '6'; c++) {
            System.out.println (c);
            try {
                Thread.sleep (1000);
            } catch (InterruptedException e) { }
        }
    }
}
```

due metodi, `m1` e `m2`, vengono invocati contemporaneamente da due thread su una stessa istanza `pt`. Uno dei due metodi, `m1`, è stato dichiarato `synchronized`, mentre l'altro, `m2`, no. Dunque il mutex associato a `pt` viene acquisito all'ingresso del metodo `m1`, ma non blocca l'esecuzione di `m2` in quanto esso non tenta di acquisire il mutex. Il risultato prodotto è quindi il seguente:

```
1
A
2
B
3
C
4
D
5
E
```

Se invece si dichiara `synchronized` anche il metodo `m2`, si hanno due thread che tentano di acquisire lo stesso mutex, per cui i due metodi vengono eseguiti in sequenza, producendo il seguente risultato

```
1
2
```

```
3
4
5
A
B
C
D
E
```

Abbiamo detto che ciascuna istanza ha sempre associato un mutex. Se dunque due thread invocano lo stesso metodo sincronizzato su due istanze diverse, essi verranno eseguiti contemporaneamente. L'esempio seguente

```
public class ProvaThread3 implements Runnable
{
   public static void main (String argv[]) {
      ProvaThread3 pt = new ProvaThread3();
      ProvaThread3 pt2 = new ProvaThread3();
      Thread t = new Thread (pt);
      t.start ();
      pt2.m1();
   }
   public void run() {
      m1();
   }
   synchronized void m1 () {
      for (char c = 'A'; c < 'F'; c++) {
         System.out.println (c);
         try {
            Thread.sleep (1000);
         } catch (InterruptedException e) { }
      }
   }
}
```

produce quindi il risultato

```
A
A
B
B
C
C
D
D
E
E
```

Anche i metodi statici possono essere dichiarati sincronizzati. Poiché essi non sono legati ad alcuna istanza, viene acquisito il mutex associato all'istanza della classe `Class` che descrive la classe. Invocando due metodi statici sincronizzati di una stessa classe da due thread diversi, essi verranno quindi eseguiti in sequenza mentre invocando un metodo statico e un metodo d'istanza, entrambi sincronizzati, di una stessa classe, essi verranno eseguiti contemporaneamente, come dimostra l'esempio

```
public class ProvaThread4 implements Runnable
{
   public static void main (String argv[]) {
      ProvaThread4 pt = new ProvaThread4();
      Thread t = new Thread (pt);
      t.start ();
      m2();
   }
   public void run() {
      m1();
   }
   synchronized void m1 () {
      for (char c = 'A'; c < 'F'; c++) {
         System.out.println (c);
         try {
            Thread.sleep (1000);
         } catch (InterruptedException e) { }
      }
   }
   static synchronized void m2 () {
      for (char c = '1'; c < '6'; c++) {
         System.out.println (c);
         try {
            Thread.sleep (1000);
         } catch (InterruptedException e) { }
      }
   }
}
```

che produce come risultato

```
1
A
2
B
C
3
D
4
E
5
```

Per ottenere l'esecuzione in sequenza dei due metodi è sufficiente dichiarare `static` anche il metodo `m1`; in tal caso il risultato diverrà

```
1
2
3
4
5
A
B
C
D
E
```

È opportuno tenere conto sempre della possibilità che un metodo sia eseguito da più di un thread contemporaneamente, anche se a una prima analisi non si prevede una tale eventualità. Le classi standard di Java sono tutte previste per il funzionamento con più thread.

Nel caso comunque che sia noto il fatto che una classe non ha metodi sincronizzati ma si desidera evitare l'accesso contemporaneo a uno o più metodi, è possibile acquisire il mutex di una determinata istanza racchiudendo il o i metodi da sincronizzare in un blocco `synchronized` la cui forma generale è la seguente

```
synchronized (istanza) {
   istruzione1;
   ...
   istruzioneN;
}
```

L'esempio:

```
public class ProvaThread5 implements Runnable
{
   public static void main (String argv[]) {
      ProvaThread5 pt = new ProvaThread5();
      Thread t = new Thread (pt);
      t.start ();
      synchronized (pt) {
         pt.m2();
      }
   }
   public void run() {
      m1();
   }
   synchronized void m1 () {
      for (char c = 'A'; c < 'F'; c++) {
         System.out.println (c);
```

```
            try {
               Thread.sleep (1000);
            } catch (InterruptedException e) { }
         }
      }
      void m2 () {
         for (char c = '1'; c < '6'; c++) {
            System.out.println (c);
            try {
               Thread.sleep (1000);
            } catch (InterruptedException e) { }
         }
      }
}
```

esegue i metodi `m1` e `m2` non contemporaneamente, anche se `m2` non è dichiarato `synchronized`. Questo modo di gestire la sincronizzazione con dei mutex impliciti semplifica notevolmente la gestione di programmi multithread poiché il programmatore non ha la preoccupazione di rilasciare il mutex ogni volta che un metodo termina normalmente o a causa di una eccezione, in quanto questa operazione viene eseguita automaticamente.

Un thread può acquisire più volte uno stesso mutex che viene considerato nuovamente disponibile quando il thread lo ha rilasciato tante volte quante lo ha acquisito. Questo serve per evitare condizioni di *deadlock* che potrebbero bloccare la Virtual Machine. Nell'esempio che segue

```
class ProvaThread6 {
   public static void main(String[] args) {
      ProvaThread6 pt = new ProvaThread6();
      synchronized(pt) {
         synchronized(pt) {
            System.out.println("fatto!");
         }
      }
   }
}
```

viene acquisito due volte il mutex su `pt`. Se non fosse permesso acquisire più volte uno stesso mutex da uno stesso thread, il secondo blocco `synchronized` aspetterebbe in eterno il rilascio del mutex, per cui in pratica la Virtual Machine si bloccherebbe. Viceversa il programma mostrato funziona senza problemi.

✔ NOTA

Nella documentazione di Java e in generale nei manuali forniti con gli ambienti di sviluppo, spesso si tende a indicare con monitor *ciò che abbiamo definito con il termine mutex. Nella letteratura sulla* programmazione concorrente, *monitor indica*

un tipo astratto, cioè un oggetto, comune a più processi o thread, che ha il compito di gestire una risorsa condivisa. Con il termine monitor, dunque, possiamo far riferimento al mutex corredato dai metodi per la comunicazione fra thread che vedremo tra poco. Secondo il punto di vista degli autori è più corretto affermare che un oggetto di Java, grazie al suo mutex e ai suoi metodi di comunicazione fra thread, può diventare un monitor. Nel paragrafo finale del capitolo viene presentato un esempio di un oggetto di questo tipo.

13.3 Variabili volatili

Java permette ai thread che accedono a variabili condivise di mantenerne una copia privata di lavoro per migliorare le prestazioni; queste copie temporanee vengono riallineate sicuramente con le copie principali solo in determinati momenti e cioè quando il thread acquisisce un mutex.

Java prevede un altro modo per assicurare che due thread si scambino informazioni in modo affidabile tramite variabili condivise, e consiste nell'usare il modificatore `volatile` nella dichiarazione di queste ultime. Quando una variabile viene dichiarata volatile, l'interprete esegue un riallineamento tra la copia di lavoro e la copia principale a ogni accesso, limitando le possibilità di ottimizzazione del codice, ma garantendo la congruenza dei valori fra thread diversi. Dichiarare una variabile contemporaneamente `volatile` e `final` causa un errore in compilazione.

13.4 Comunicazione fra thread

Evitare l'esecuzione contemporanea di brani di codice particolarmente delicati è una condizione necessaria per poter sviluppare programmi multithread che non diano risultati imprevedibili. Per poter sfruttare meglio le potenzialità del multithreading, è però necessario disporre dei meccanismi di comunicazione che permettano il colloquio fra due o più thread, in modo tale che essi possano collaborare per uno scopo comune. Un esempio classico è quello di un thread che produce dei dati, per esempio leggendoli da una base dati, e un altro li deve utilizzare in qualche modo come, per esempio, inviarli a una stampante. Quando due thread cooperano in questo modo, si dice che fra essi esiste una relazione *produttore-consumatore*. Il problema che si pone in questo tipo di relazione, consiste nel fatto che il thread consumatore deve attendere che i dati da utilizzare diventino disponibili, mentre il thread produttore deve essere sicuro che il thread consumatore sia pronto a ricevere per non rischiare una perdita di dati.

Java mette a disposizione alcuni metodi che fanno parte della classe `Object`, per cui sono disponibili da istanze di qualsiasi classe. Questi metodi debbono essere invocati solo in metodi sincronizzati e sono

- `public final void wait()`: il thread che invoca questo metodo rilascia il mutex associato all'istanza e viene sospeso fintanto che non viene risvegliato da un altro thread che acquisisce lo stesso mutex e invoca il metodo `notify` o `notifyAll`, oppure viene interrotto con il metodo `interrupt` della classe `Thread`.

- `public final void wait(long millis)`: questo metodo si comporta in modo analogo al precedente, ma se dopo un'attesa corrispondente al numero di millisecondi specificato in `millis` non è stato risvegliato, esso si risveglia in tutti i casi.
- `public final void wait(long millis, int nanos)`: questo metodo si comporta in modo analogo al precedente, ma permette di specificare l'attesa con una risoluzione temporale a livello di nanosecondi su sistemi operativi che lo consentono.
- `public final void notify()`: risveglia il primo thread che ha invocato `wait` sull'istanza. Poiché il metodo che invoca `notify` deve aver acquisito il mutex, il thread risvegliato deve attenderne il rilascio, dopodiché compete per la sua acquisizione come un qualsiasi altro thread.
- `public final void notifyAll()`: risveglia tutti i thread che hanno invocato `wait` sull'istanza. I thread risvegliati competono per l'acquisizione del mutex e se ne esiste uno con priorità più alta, esso viene subito eseguito.

Vediamo ora un esempio completo e semplice, per quanto possibile, in cui supponiamo di voler lanciare un thread che si occupi esclusivamente di visualizzare dei dati che gli vengono forniti da un altro thread.

```
class Visualizza implements Runnable {
   Monitor monitor;
   Visualizza (Monitor m) {
      monitor = m;
      Thread t = new Thread (this);
      t.setDaemon(true);
      t.start ();
   }
   public void run () {
      for ( ; ; )
         System.out.println (monitor.ricevi());
   }
}

class Monitor {
   private boolean pieno = false;
   private String buffer;
   synchronized void spedisci (String msg) {
      if (pieno)
         try {
            wait();
         } catch (InterruptedException e) { }
      pieno = true;
      notify ();
      buffer = msg;
```

```
    }
    synchronized String ricevi () {
        if (!pieno)
            try {
                wait();
            } catch (InterruptedException e) { }
        pieno = false;
        notify ();
        return buffer;
    }
}

public class ProvaThread7
{
    public static void main (String argv[]) {
        Monitor monitor = new Monitor();
        String messaggi[] = { "Esempio",
                              "di",
                              "comunicazione",
                              "fra",
                              "thead" };
        Visualizza v = new Visualizza (monitor);
        for (int i = 0; i < messaggi.length; i++)
            monitor.spedisci(messaggi[i]);
    }
}
```

La classe `Visualizza` serve per far partire il thread che si occupa della visualizzazione e infatti notiamo che il metodo `run` fa un ciclo infinito in cui esegue il metodo `println`. Il costruttore di questa classe riceve e memorizza in una variabile d'istanza un oggetto della classe `Monitor` che, come vedremo tra poco, si occupa di sincronizzare le operazioni tra produttore e consumatore. Il costruttore crea poi un nuovo thread e, prima di farlo partire, lo fa diventare demone. Senza quest'ultima operazione, il thread appena creato girerebbe all'infinito non consentendo all'interprete di concludere l'elaborazione e obbligando l'utente a concluderla in modo forzato con la pressione di un tasto d'interruzione. In questo modo invece, l'interprete conclude l'elaborazione, e quindi anche il nuovo thread, non appena termina l'altro thread attivo.

La classe `Monitor` ha il compito di effettuare la comunicazione coordinata fra due thread. Un'istanza di questa classe riceve una stringa dal thread produttore tramite il metodo `spedisci`, la memorizza nella sua variabile d'istanza `buffer` fintanto che un thread consumatore ne richiede il valore tramite il metodo `ricevi`. La variabile d'istanza booleana, `pieno`, indica se `buffer` contiene un valore che deve essere ricevuto dal thread consumatore o meno.

Quando il thread produttore invoca il metodo `spedisci`, quest'ultimo verifica per prima cosa il valore di `pieno` e, se corrisponde a `false`, memorizza il dato ricevuto nella variabile `buffer`, aggiorna di conseguenza la variabile `pieno` ed ese-

gue il metodo `notify` per avvertire il thread consumatore che c'è un nuovo dato disponibile. Se `pieno` vale `true`, il thread si mette in attesa fintanto che il thread consumatore non segnala che l'area di comunicazione è disponibile.

Il thread consumatore invoca invece il metodo `ricevi` il quale verifica il valore della variabile `pieno`. Se essa vale `true` significa che c'è un dato disponibile, per cui, prima di restituire il valore contenuto in `buffer`, modifica il valore di `pieno` ed esegue il metodo `notify` per avvertire il thread produttore che l'area di comunicazione è disponibile per ospitare un nuovo dato. Se `pieno` vale `false`, il thread si mette in attesa fintanto che il thread produttore non segnala la disponibilità di un nuovo dato.

La classe `ProvaThread7` serve per verificare il funzionamento di tutto il meccanismo e il thread di partenza funge da produttore fornendo dei dati memorizzati in un array. Come si può notare, gli unici metodi sincronizzati e tutti i metodi per la comunicazione sono contenuti esclusivamente nella classe `Monitor`.

Quando si usano più thread, bisogna fare attenzione a evitare condizioni di *deadlock*. Una condizione di deadlock si verifica quando due o più thread si bloccano reciprocamente, non permettendo al programma di proseguire e lasciando l'interprete in uno stato di attesa infinita. Per esempio se un thread che ha acquisito il mutex dell'oggetto A tenta di acquisire il mutex dell'oggetto B mentre un altro thread che ha acquisito il mutex dell'oggetto B tenta di acquisire il mutex dell'oggetto A, entrambi i thread sono destinati ad aspettare indefinitamente.

Java non prevede nessun tipo di riconoscimento o risoluzione di una condizione di deadlock, per cui è il programmatore che deve usare tecniche opportune per evitare il verificarsi di tale condizione. Le condizioni di deadlock per fortuna non sono frequenti, ma possono verificarsi anche dopo molto tempo di funzionamento corretto e sono di difficile individuazione. Come regola generale, un deadlock può verificarsi solo quando due o più thread tentano di acquisire il mutex di due o più oggetti. Demandando quindi la sincronizzazione di più thread a un unico oggetto, come nell'esempio precedente, siamo certi di evitare questo insidioso problema.

Domande di verifica

1. Qual è il significato di *multitherad*?
2. Due `thread` possono avere lo stesso nome?
3. Cosa è il *thread group*? E la priorità di esecuzione?
4. Cosa si intende per *thread non demone*? E per `thread` attivo e sospeso?
5. Quali sono la classe e il metodo che consentono di attivare un nuovo `thread`?
6. A cosa è utile l'interfaccia `Runnable`?
7. Quali sono i metodi per attivare un `thread` e in cosa si distinguono?
8. A cosa serve il modificatore `synchronized` e come si usa? E il modificatore `volatile`?

9. Cosa si intende per thread produttore-consumatore? Come funziona? Quali metodi sono necessari allo scopo?

Esercizi

1. Scrivere un programma che abbia due thread, ciascuno dei quali esegue un ciclo identico. Verificare quante volte viene eseguito ciascun ciclo in un tempo determinato variando la priorità dei thread.
2. Scrivere un programma che visualizzi un numero progressivo ogni secondo, fino a che non si preme Invio.
3. Provare a scrivere un programma in cui due thread siano in relazione produttore-consumatore senza l'ausilio dei metodi di comunicazione fra thread `wait` e `notify`. Un modo possibile è quello di controllare a intervalli temporali la variabile utilizzata per la comunicazione fra i thread (polling).
4. Modificare l'ultimo esempio del capitolo in modo che utilizzi un array di stringhe, anziché una sola, per la comunicazione fra i thread, in modo da migliorare le prestazioni.
5. L'ultimo esempio del capitolo, produttore-consumatore, non funziona correttamente se due o più thread usano l'istanza della classe `Monitor` per spedire dati all'istanza della classe `Visualizza`. Verificare questo fatto e trovarne una spiegazione.
6. Modificare l'ultimo esempio del capitolo in modo che due o più thread possano utilizzare l'istanza della classe `Monitor` per spedire dati all'istanza della classe `Visualizza`.
7. Scrivere un programma che vada in deadlock.

Capitolo 14

Ambiente di esecuzione

Obiettivi didattici

- Gestione di insiemi di oggetti
- Interfaccia `Map` e i suoi metodi
- Gestione della sicurezza
- Classi `Runtime` e `Process`
- Classe `System` e oggetto `Properties` per la definizione dell'ambiente
- Altri metodi utili per l'ambiente di esecuzione

In questo capitolo vengono illustrate alcune importanti caratteristiche relative all'esecuzione dei programmi e alla loro interazione con il sistema. A tale scopo viene premessa la trattazione di alcune classi di utilizzo generale le cui istanze sono utilizzate per scambiare informazioni tra programmi e sistema.

14.1 Associazioni di oggetti

Abbiamo già parlato di collezioni di oggetti, abbiamo visto che esse permettono di raggruppare un numero qualsiasi di oggetti e successivamente di ricuperarli, o tramite una scansione sequenziale oppure tramite un indice in modo analogo a quanto si fa con gli array. In molti casi però torna utile poter associare a ciascun oggetto della collezione un nome da usare per la sua individuazione e ricupero. Pensiamo, per esempio, all'insieme dei partecipanti a una gita, dei libri di una biblioteca o dei programmi attivi in un computer. In questi casi sarebbe più semplice poter ricuperare un elemento in base, per esempio, a una stringa invece che a un semplice numero.

L'interfaccia `Map<K, V>`, appartenente al package `java.util`, mette a disposizione dei metodi per la gestione di un insieme di coppie di oggetti. Ogni volta che si vuole memorizzare un oggetto *valore* (`value`) in un'istanza di una classe che implementa questa interfaccia, è necessario fornire anche un altro oggetto *chiave* (`key`) in base al quale sarà possibile poi ricuperare l'oggetto valore. Gli oggetti chiave devono essere tutti diversi l'uno dall'altro. Come regola le classi che implementano `Map` dovrebbero stabilire l'uguaglianza fra due chiavi per mezzo del metodo `equals`. Questa interfaccia, pur non rappresentando strettamente una collezione (non deriva da `Collection`), fa parte del "collection framework" di cui abbiamo già parlato.

I metodi definiti nell'interfaccia sono:

- `public int size()`: restituisce il numero di elementi contenuti nell'istanza.
- `public boolean isEmpty()`: restituisce `true` se l'istanza non contiene elementi, `false` altrimenti.
- `public boolean containsKey (Object key)`: restituisce `true` se la chiave specificata come argomento individua un elemento presente nell'insieme, `false` altrimenti. Questo metodo può gettare un'eccezione di tipo `ClassCastException`, se `key` è di tipo non appropriato, e `NullPointerException` se `key` è uguale a `null` e l'implementazione dell'interfaccia non consente chiavi `null`.
- `public boolean containsValue (Object value)`: restituisce `true` se l'oggetto specificato come argomento è presente tra i valori dell'insieme.
- `public V get(Object key)`: se `key` corrisponde a una chiave memorizzata nell'istanza, questo metodo restituisce l'oggetto valore associato, altrimenti restituisce `null`. Questo metodo può gettare eccezioni in modo analogo a `containsKey`. Notare che la restituzione di `null` non garantisce che la chiave sia assente in quanto potrebbe accadere che la chiave specificata sia presente e che a essa corrisponda il valore `null`.
- `public V put(K key, V value)`: inserisce nell'istanza la coppia `key/value`. Se l'istanza contiene già la chiave specificata, il valore associato a essa viene sostituito e poi restituito, altrimenti viene restituito `null`. Questo metodo può non essere supportato dalla classe e quindi gettare l'eccezione `UnsupportedOperationException`. Altre eccezioni che questo metodo può gettare sono: `ClassCastException` o `IllegalArgumentException` se le classi di `key` e/o di `value` non sono di tipo appropriato, `NullPointerException` se `key` e/o `value` sono `null` e questa classe non supporta tale valore.
- `public V remove(Object key)`: se `key` corrisponde a una chiave memorizzata nell'istanza, la coppia chiave/valore viene rimossa dall'istanza e viene restituito l'oggetto valore associato, altrimenti viene restituito `null`. Anche questo metodo può non essere supportato e quindi gettare eccezione `UnsupportedOperationException`.
- `public void putAll(Map<? extends K, ? extends V> m)`: copia tutte le coppie `key/value` di `m` nell'istanza corrente, eliminando tutte le coppie eventualmente presenti. Questo metodo può gettare le stesse eccezioni di `put`.

- `public void clear()`: svuota completamente l'istanza di tutti gli elementi contenuti. Questo metodo può non essere supportato e quindi gettare eccezione `UnsupportedOperationException`.
- `public Set<K> keySet()`: restituisce un oggetto, che implementa l'interfaccia `Set`, contenente tutte le chiavi dell'istanza. Dell'interfaccia `Set` abbiamo accennato brevemente dicendo che rappresenta un insieme di oggetti tutti diversi l'uno dall'altro.
- `public Collection<V> values()`: restituisce un oggetto, che implementa l'interfaccia `Collection`, contenente tutti i valori dell'istanza.
- `public Set<Map.Entry<K,V>> entrySet()`: restituisce un oggetto, che implementa l'interfaccia `Set`, contenente un insieme di oggetti di tipo `Map.Entry`. Questa è un'interfaccia membro di `Map` che rappresenta una coppia chiave/valore. Ciascun elemento del `Set` restituito corrisponde a un elemento dell'istanza corrente.
- `public boolean equals(Object obj)`: restituisce `true` se `obj` ha lo stesso contenuto dell'istanza corrente.
- `public int hashCode()`: restituisce un codice hash per l'istanza corrente: tale codice è definito come la somma dei codici hash di tutti gli oggetti contenuti nel `Set` restituito da `entrySet()`.

L'interfaccia `Enumeration<E>`, anch'essa appartenente al package `java.util`, permette di scorrere un insieme di oggetti. Dalla versione 2 di Java è stata rimpiazzata dall'interfaccia `Iterator`, ma nelle classi dove era già usata è rimasta per compatibilità. Definisce due metodi, e precisamente:

- `public E nextElement()`: restituisce il prossimo elemento se esiste, altrimenti getta l'eccezione `NoSuchElementException`.
- `public boolean hasMoreElements()`: restituisce `true` se ci sono ancora elementi da scorrere, `false` altrimenti.

La classe `Hashtable` implementa `Map` e possiede anche i seguenti metodi:

- `public Enumeration<K> keys()`: restituisce un'istanza di `Enumeration` che permette di scorrere tutte le chiavi contenute nell'istanza corrente.
- `public Enumeration<V> elements()`: restituisce un'istanza di `Enumeration` che permette di scorrere tutte i valori contenuti nell'istanza corrente.

Vediamo di fare un esempio semplice per riassumere quanto detto: supponiamo di voler memorizzare l'età di un certo numero di persone e di voler poi ricuperare questo dato in base al nome di battesimo.

```
import java.util.Enumeration;
import java.util.Hashtable;
public class Eta
{
   public static void main (String argv[]) {
```

```
        Hashtable<String,Integer> ht = new
                                    Hashtable<String,Integer>();
        ht.put ("Marco", new Integer (39));
        ht.put ("Luigi", new Integer (33));
        ht.put ("Claudio", new Integer (37));
        System.out.println("Età di Luigi=" + ht.get("Luigi"));
        Enumeration<String> enk = ht.keys();
        while (enk.hasMoreElements())
           System.out.println(enk.nextElement());
        Enumeration<Integer> env = ht.elements();
        while (env.hasMoreElements())
           System.out.println(env.nextElement());
    }
}
```

Abbiamo dovuto utilizzare degli oggetti della classe `Integer` per memorizzare l'età, in quanto semplici variabili di tipo `int` non sarebbero state accettate perché non sono oggetti. Nell'esempio sono stati inseriti anche due cicli che visualizzano rispettivamente l'elenco completo delle chiavi e dei valori, in modo da mostrare l'utilizzo di oggetti di tipo `Enumeration`. Il risultato dell'elaborazione è la seguente visualizzazione

```
Età di Luigi=33
Luigi
Claudio
Marco
33
37
39
```

La classe `java.util.Properties` deriva da `Hashtable` e ha due estensioni funzionali con la limitazione che sia le chiavi che i valori debbono essere di tipo `String`. La prima estensione riguarda il fatto che il contenuto di un'istanza di questa classe può essere scritto e letto direttamente da un file (o meglio uno `Stream`, come vedremo nel capitolo seguente). L'altra estensione riguarda la possibilità di associare a ogni istanza un altro oggetto della stessa classe dove poter effettuare la ricerca di una chiave nel caso essa non risulti presente. Questo meccanismo in pratica permette di assegnare dei valori di default per certe chiavi. Il metodo `getProperty` si comporta in modo simile al metodo `get` di `Map` ma, a differenza di questo, quando non trova una chiave ed esiste un oggetto `Properties` associato all'istanza, restituisce il valore restituito dal metodo `getProperty` di quest'ultimo. Per vedere in che tipo di situazione è possibile avvantaggiarsi delle caratteristiche di questa classe, supponiamo di voler memorizzare dei dati personali di un insieme di persone. Supponiamo inoltre di sapere che, al momento attuale, i dati richiesti per ogni persona siano nome, età e cittadinanza, anche se non sempre tali informazioni sono tutte disponibili, ma che probabilmente in un futuro molto prossimo saranno richiesti altri tipi di informazione.

Nell'esempio

```
import java.util.Enumeration;
import java.util.Properties;

class Persona {
   static private Properties datiDefault = new Properties();
   public Properties dati = new Properties(datiDefault);

   public Persona (String nome) {
      dati.put ("Nome", nome);
   }

   static public Enumeration tipiDati() {
      return datiDefault.keys();
   }
   static {
      datiDefault.put ("Nome", "sconosciuto");
      datiDefault.put ("Nazionalità", "sconosciuta");
      datiDefault.put ("Età", "sconosciuta");
   }
}

public class Gente {
   public static void main (String argv[]) {
      Persona gente[] = new Persona[3];
      gente[0] = new Persona("Marco");
      gente[0].dati.put ("Nazionalità", "Italiana");

      gente[1] = new Persona("Claudio");
      gente[1].dati.put ("Età", "37");

      gente[2] = new Persona("Luigi");
      gente[2].dati.put ("Età", "33");
      gente[2].dati.put ("Nazionalità", "Italiana");

      for (int i = 0; i < gente.length; i++)
          mostra (gente[i]);
   }

   static public void mostra (Persona p) {
      String chiave;
      Enumeration en = Persona.tipiDati();

      while (en.hasMoreElements())
         System.out.println ( (chiave =
                     (String) en.nextElement()) + " = "
                           + p.dati.getProperty (chiave));
```

```
        System.out.println ("-------");
    }
}
```

La classe `Persona` usa una variabile d'istanza di tipo `Properties` per memorizzare i dati di una persona e si avvale di una variabile statica `datiDefault` dello stesso tipo per contenere dei valori di default. Il metodo statico `mostra` usa le chiavi di `datiDefault` per visualizzare i dati importanti. Se in seguito saranno richiesti altri dati, è sufficiente aggiungere le chiavi e i valori default relativi in `datiDefault` e il codice per aggiungere questi nuovi dati senza necessità di modificare nessun'altra parte del programma. L'esempio precedente produce la seguente visualizzazione

```
Età = sconosciuta
Nome = Marco
Nazionalità = Italiana
-------
Età = 37
Nome = Claudio
Nazionalità = sconosciuta
-------
Età=33
Nome = Luigi
Nazionalità = Italiana
-------
```

Nel caso si voglia avere la certezza di non vedersi restituire un valore `null` anche quando una chiave non può essere reperita neanche tra i valori di default, è possibile invocare il metodo `getProperty(String chiave, String default)` che in casi del genere restituisce il valore specificato nella stringa `default`.

14.2 Classe `Date`

Questa classe, appartenente al package `java.util`, viene utilizzata per rappresentare uno specifico istante con una precisione fino al millesimo di secondo. Il suo costruttore senza argomenti restituisce un'istanza inizializzata con l'istante di sistema in cui è stata allocata.

✔ NOTA

Nelle versioni di Java precedenti la 1.1, questa classe aveva anche lo scopo di interpretare le date in termini di anno, mese, giorno, ore, minuti, secondi e aveva dei metodi per interagire con le stringhe contenenti una rappresentazione di data. Successivamente gli sviluppatori del linguaggio si sono resi conto che la classe, così com'era, non rispondeva ai bisogni dettati dall'internazionalizzazione, per cui hanno deciso di creare due nuove classi, `Calendar` *e* `DateFormat`*, per adempiere agli scopi suddetti.*

La classe dispone dei seguenti costruttori e metodi:

- `public Date()`: crea un'istanza inizializzata con la data e l'ora corrente con una precisione fino al millesimo di secondo, sistema operativo permettendo.
- `public Date(long millisecondi)`: crea un'istanza inizializzata con la data e l'ora ottenuta considerando `millisecondi` come il numero di millisecondi trascorsi dalle 00:00:00 del 1 gennaio 1970 (questa è la data di riferimento usata nei sistemi UNIX ed è detta "the epoch").
- `public long getTime()`: restituisce il numero di millisecondi a partire dalle 00:00:00 del 1 gennaio 1970.
- `public void setTime(long millisecondi)`: imposta l'istanza considerando `millisecondi` come il numero di millisecondi trascorsi dalle 00:00:00 del 1 gennaio 1970.
- `public boolean equals(Object altraData)`: restituisce `true` se l'oggetto passato come argomento è di tipo `Date` e se l'istante rappresentato nell'istanza è uguale a quello rappresentato in `altraData`, `false` altrimenti.
- `public boolean before(Date altraData)`: restituisce `true` se l'istante rappresentato nell'istanza è antecedente a quello rappresentato in `altraData`, `false` altrimenti.
- `public boolean after(Date altraData)`: restituisce `true` se l'istante rappresentato nell'istanza è seguente a quello rappresentato in `altraData`, `false` altrimenti.
- `public int compareTo (Date altraData)`: restituisce zero se l'istante rappresentato nell'istanza è uguale a quello rappresentato in `altraData`, un valore minore di zero se `altraData` rappresenta un istante antecedente e un valore maggiore di zero se `altraData` rappresenta un istante successivo.

Esistono altri costruttori e metodi, il cui uso però è attualmente deprecato.

14.3 Il metodo `main`

Fin dal primo capitolo abbiamo visto come qualsiasi programma Java inizi da una classe contenente un metodo statico `main`. In effetti non esiste una vera e propria entità 'programma' in quanto qualsiasi elaborazione viene eseguita da un insieme di classi che vengono lette dall'interprete solo in caso di necessità. Uno stesso insieme di classi può dare risultati diversi modificando la classe di partenza. Osserviamo le seguenti classi, residenti rispettivamente nei file `Main1.java` e `Main2.java`.

File `Main1.java`:

```
public class Main1
{
   public static void main (String argv[]) {
      System.out.println("Main1");
```

```
        Main2 m2 = new Main2();
    }
    static {
        System.out.println("Main1 static");
    }
}
```

File `Main2.java`:

```
public class Main2 {
    public static void main (String argv[]) {
        System.out.println("Main2");
    }
    static {
        System.out.println("Main2 static");
    }
}
```

Le classi dell'esempio hanno entrambe un metodo `main` e differiscono solamente per il fatto che nella prima viene creata un'istanza della seconda mentre in quest'ultima non c'è alcun riferimento alla prima. Compilando le classi e lanciando l'interprete con il comando

```
java Main1
```

si ottiene sul video il risultato

```
Main1 static
Main1
Main2 static
```

che mostra chiaramente come il metodo `static` di `Main1` venga eseguito prima del metodo `main` mentre il metodo `static` di `Main2` viene eseguito solo prima che venga invocato un metodo definito nella classe. Lanciando invece il comando

```
java Main2
```

si ottiene sul video il seguente risultato

```
Main2 static
Main2
```

che mostra che la classe `Main1` non è stata caricata dall'interprete, come del resto era ovvio attendersi.

È possibile fornire dei parametri a `main`, scrivendoli sulla linea di comando, che vengono memorizzati nell'array di stringhe che compare come parametro del metodo. Compilando l'esempio

```
public class Main3 {
    public static void main (String argv[]) {
        for (int i = 0; i < argv.length; i++)
            System.out.println("N." + i +"["+ argv[i] + "]");
    }
}
```

e lanciando il comando

```
java Main3 "è una frase" è una frase
```

si ottiene

```
N.0[è una frase]
N.1[è]
N.2[una]
N.3[frase]
```

14.4 Classe SecurityManager

Un programma Java in esecuzione può avere un *gestore della sicurezza*, che è un'istanza di una sottoclasse di `SecurityManager`. Questa classe astratta contiene un gran numero di metodi il cui nome inizia con "check". Essi vengono richiamati da vari metodi delle librerie Java prima dell'esecuzione di certe operazioni delicate. Il gestore della sicurezza ha l'opportunità di prevenire il completamento di determinate operazioni gettando un'eccezione di tipo `SecurityException`.

Lo scopo del gestore della sicurezza è di poter creare degli ambienti controllati in cui certe operazioni non sono permesse, come nel caso delle applicazioni scaricate via Internet che non possono né leggere né scrivere sui dischi del computer locale.

14.5 Classi Runtime e Process

La classe `Runtime` deriva da `Object` e appartiene al package `java.lang`. Una sua istanza rappresenta l'esecuzione dell'interprete Java. Non è possibile creare un oggetto di questa classe in quanto ha un solo costruttore che è privato. È però possibile ottenere una referenza all'oggetto correntemente in esecuzione mediante il metodo statico `Runtime.getRuntime()`. L'istanza ottenuta può essere usata per causare la terminazione dell'elaborazione per mezzo del metodo d'istanza `public void exit(int stato)`. Nella variabile `stato` è possibile specificare un codice da restituire al sistema operativo e che può essere letto da chi ha lanciato l'esecuzione; per convenzione il codice 0 significa che il programma ha eseguito i propri compiti senza problemi mentre un numero diverso da 0 indica convenzionalmente che qualcosa non è andata per il verso giusto. Quando un programma Java termina per l'uscita dal metodo `main`, il codice restituito è 0, mentre se la terminazione è causata da un'eccezione non catturata, il codice restituito è 1. Nel paragrafo successivo vedremo il metodo statico `System.exit(int stato)` che è il modo convenzionale di terminare l'elaborazione e che invoca questo metodo.

Il metodo d'istanza `exec` di `Runtime` fornisce il modo per attivare altri programmi; di questo metodo esistono alcune varianti che differiscono per i parametri formali, ma la forma più semplice è la seguente

```
public Process exec (String cmd)
```

che consente di specificare nella stringa `cmd` un comando in modo analogo a come faremmo al prompt dei comandi. Per attivare un programma che gira in un ambiente

a finestre, come per esempio l'editor di testo "notepad" di Windows, è sufficiente un'istruzione come la seguente

```
try {
   Runtime.getRuntime().exec("notepad");
} catch (Exception e) {
   System.out.println ("Errore exec:" + e);
}
```

Il blocco `try` è necessario in quanto `exec` può gettare le eccezioni `IOException` e `SecurityException`. Volendo fare in modo che l'editor apra direttamente il testo in "Main3.java", è sufficiente modificare l'istruzione precedente nel modo seguente

```
try {
   Runtime.getRuntime().exec("notepad Main3.java");
} catch (Exception e) {
   System.out.println ("Errore exec:" + e);
}
```

Questo metodo può essere utilizzato anche per lanciare altri programmi Java, ma provando a lanciare l'esecuzione dell'ultimo esempio del precedente paragrafo con l'istruzione

```
try {
   Runtime.getRuntime().exec("java Main3 \"è una frase\" è
                              una frase");
} catch (Exception e) {
   System.out.println ("Errore exec:" + e);
}
```

il risultato ottenuto è alquanto deludente poiché su video non appare niente. La mancata visualizzazione non dipende dal fatto di lanciare un programma Java, ma dal fatto di lanciare un programma che funziona in modalità non grafica, cioè a caratteri. Per capirne la causa, bisogna premettere alcune nozioni del sistema operativo dove è nato Java, cioè UNIX, ereditate anche da altri sistemi operativi.

Quando si lancia l'esecuzione di un'applicazione UNIX, il sistema crea un *processo* e gli assegna tre file aperti, detti rispettivamente file *standard input*, *standard output* e *standard error*. Il primo di questi permette all'utente di fornire dati al programma, il secondo permette al programma di visualizzare dati, mentre il terzo viene utilizzato per la visualizzazione degli errori. Con l'avvento delle interfacce grafiche si è modificato il meccanismo d'interazione tra programmi e utente, ma, nonostante ciò, i tre file menzionati sono ancora utilizzati, insieme a una meccanismo di UNIX, detto *piping*, per effettuare una comunicazione tra processi. Attraverso tale meccanismo un programma *figlio* può colloquiare col processo *padre* che ne ha lanciato l'esecuzione, come se quest'ultimo fosse un utente e quindi utilizzando lo standard input per leggere i dati dal padre e lo standard output per fornirglieli. Poiché Java è destinato prevalentemente ad ambienti a grafici, è stato semplificato l'utilizzo dei tre file come meccanismo di comunicazione tra processi, per cui quando in Java si lancia l'esecuzione di un programma con il metodo `exec`, l'interprete si "appropria" dei tre file assegnati al nuovo processo dal sistema. Il processo figlio quindi, se non ha interfaccia

grafica, non può più comunicare direttamente con l'utente ma solo con il processo padre. Quest'ultimo può interagire con il figlio per mezzo di un'istanza della classe `OutputStream` e due istanze della classe `InputStream`: queste due classi astratte definiscono i metodi per lavorare con i file e saranno illustrate nel prossimo capitolo.

Per interagire col processo figlio, il processo padre deve utilizzare l'istanza della classe `Process` che rappresenta il processo lanciato e che viene restituito dal metodo `exec`; questa classe ha i seguenti metodi:

- `public abstract OutputStream getOutputStream()`: restituisce un'istanza di `OutputStream` corrispondente allo standard input del processo.
- `public abstract InputStream getInputStream()`: restituisce un'istanza di `InputStream` corrispondente allo standard output del processo.
- `public abstract InputStream getErrorStream()`: restituisce un'istanza di `InputStream` corrispondente allo standard error del processo.
- `public abstract int waitFor()`: pone in attesa il thread corrente fino al termine del processo e, quando esso termina, ne restituisce il codice di uscita. Se il thread viene interrotto, viene gettata un'eccezione `InterruptedException`.
- `public abstract int exitValue()`: se il processo è già terminato, restituisce il codice di uscita, altrimenti getta un'eccezione `IllegalThreadStateException`.
- `public abstract void destroy()`: termina il processo corrispondente all'istanza.

La classe `Process` è astratta, ma `exec` restituisce una sua sottoclasse concreta dipendente dal sistema operativo.

La visualizzazione dell'output di un programma a caratteri lanciato dall'interno di un programma Java può essere ottenuta creando un nuovo thread dedicato a questo scopo come nell'esempio

```
import java.io.*;

class OutFiglio implements Runnable {
  InputStream in;
  OutFiglio (InputStream in) {
    this.in = in;
    Thread t = new Thread(this);
    t.setDaemon(true);
    t.start();
  }
  public void run() {
    String s;
    byte b[] = new byte[1024];
    int len;
    try {
      while ((len = in.read(b)) >= 0) {
        s = new String(b, 0, len);
        System.out.print (s);
```

```
      }
    } catch (Exception e) {
        return;
    }
  }
}

public class Exec1
{
  public static void main (String argv[]) {
    try {
      Process p =
        Runtime.getRuntime().exec("java Main3 è una frase");
      new OutFiglio (p.getInputStream());
      int exitCode = p.waitFor();
      System.out.println ("Codice = " + exitCode);
    } catch (Exception e) {
      System.out.println ("Errore exec:" + e);
    }
  }
}
```

che produce la seguente visualizzazione

```
N.0[è]
N.1[una]
N.2[frase]
Codice = 0
```

Non entriamo nei dettagli sull'utilizzo dei file in quanto vengono illustrati nel prossimo capitolo, ma comunque dovrebbe essere sufficientemente chiaro il principio di funzionamento. Notare che abbiamo utilizzato il metodo `waitFor` per fare in modo che il thread di partenza attenda la fine del processo lanciato. Modificando la classe `Main3`

```
class Main3
{
   public static void main (String argv[]) {
      for (int i = 0; i < argv.length; i++)
         System.out.println("N." + i +"["+ argv[i] + "]");
      Runtime.getRuntime().exit (99);
   }
}
```

e rilanciando la classe `Exec1` si ottiene invece

```
N.0[è]
N.1[una]
N.2[frase]
Codice = 99
```

14.6 Classe `System`

La classe `System` contiene una serie di variabili e metodi statici che forniscono diverse funzionalità utili per gestire alcune risorse del sistema, ma che oltre a ciò non hanno molto in comune.

Non è permesso istanziare oggetti della classe `System` in quanto possiede un unico costruttore dichiarato privato.

Abbiamo già visto due dei tre oggetti statici appartenenti a questa classe, `System.out` e `System.in`, appartenenti rispettivamente alle classi `PrintStream` e `InputStream` che vedremo in dettaglio successivamente, che permettono di scrivere e leggere su terminale a caratteri, o come si dice, su *console*. Il terzo oggetto statico di `System` è `System.err` appartenente alla classe `PrintStream`, che permette di scrivere su standard error.

✔ **NOTA**

La classe `PrintStream` *è una sottoclasse concreta di* `OutputStream`. *Essendo* `InputStream` *una classe astratta, in realtà* `System.in` *contiene un'istanza di una sua sottoclasse concreta che normalmente è* `DataInputStream`. *Il fatto di specificare* `System.in` *di tipo* `InputStream` *lascia agli implementatori del linguaggio la possibilità di usare classi concrete diverse a seconda delle versioni, sistemi operativi ecc.*

14.6.1 Proprietà del sistema

Un oggetto statico di tipo `Properties` è associato alla classe `System` e contiene alcune informazioni riguardanti il sistema, dette appunto *proprietà del sistema*. Riportiamo nella Tabella 14.1 il set minimo di informazioni che debbono essere disponibili in qualsiasi versione di Java, tenendo presente però che l'elenco può differire tra versione e versione e tra fornitore e fornitore.

Per accedere alle proprietà sono disponibili i seguenti metodi statici.

- `public static Properties getProperties()`: restituisce un oggetto di tipo `Properties` contenente le proprietà del sistema.
- `public static void setProperties(Properties props)`: permette di impostare un proprio oggetto di tipo `Properties` con le proprietà del sistema.
- `public static String getProperty(String chiave)`: restituisce la proprietà legata alla chiave specificata come argomento se tale chiave esiste, `null` altrimenti.
- `public static String getProperty(String chiave, String default)`: restituisce la proprietà legata alla chiave specificata come argomento se tale chiave esiste, la stringa `default` altrimenti.

L'invocazione di questi metodi può causare un'eccezione di tipo `SecurityException`.

Tabella 14.1 Livelli d'interazione

Chiave	Valore
`java.home`	Directory d'installazione di Java
`java.version`	Versione dell'interprete
`java.vendor`	Nome del fornitore
`java.vendor.url`	Indirizzo Internet del fornitore
`java.vm.specification.version`	Versione delle specifiche della Java Virtual Machine
`java.vm.specification.vendor`	Fornitore delle specifiche della Java Virtual Machine
`java.vm.specification.name`	Nome delle specifiche della Java Virtual Machine
`java.vm.version`	Versione dell'implementazione della Java Virtual Machine
`java.vm.vendor`	Fornitore dell'implementazione della Java Virtual Machine
`java.vm.name`	Nome dell'implementazione della Java Virtual Machine
`java.specification.version`	Versione delle specifiche del Java Runtime Environment
`java.specification.vendor`	Fornitore delle specifiche del Java Runtime Environment
`java.specification.name`	Nome delle specifiche del Java Runtime Environment
`java.class.path`	Contenuto della variabile CLASSPATH
`java.class.version`	Versione delle classi
`java.library.path`	Lista di percorsi di ricerca per il caricamento delle librerie
`java.io.tmpdir`	Directory per i file temporanei
`java.compiler`	Nome del compilatore JIT (Just In Time)
`java.ext.dirs`	Directory delle estensioni
`os.name`	Nome del sistema operativo
`os.version`	Versione del sistema operativo
`os.arch`	Architettura del sistema operativo
`file.separator`	Carattere separatore di file ('/' su UNIX, '\' su Windows)
`line.separator`	Terminatore di riga
`path.separator;`	Carattere separatore di path (':' su UNIX, ';' su Windows)
`user.name`	Nome dell'utente corrente
`user.dir`	Direttori di lavoro corrente
`user.home`	Directory HOME dell'utente

14.6.2 Gestore della sicurezza

Due metodi statici permettono di ricuperare o di impostare un gestore della sicurezza. Un programma che inizia l'esecuzione tramite il metodo `main`, come in tutti gli esempi visti fino a ora, non ha nessun gestore della sicurezza impostato, per cui è possibile impostarne uno proprio. Le *applet*, cioè le applicazioni che vengono scaricate da Internet ed eseguite all'interno di un browser HTML ne hanno invece uno già impostato ed è permesso ricuperarlo, ma non sostituirlo. I metodi per manipolare i gestori della sicurezza sono:

- `public static SecurityManager getSecurityManager()`: restituisce l'istanza di `SecurityManager` che gestisce la sicurezza del sistema se ne è già stata impostata una, `null` altrimenti.
- `public static void setSecurityManager(SecurityManager s)`: se non è stato ancora impostato un gestore della sicurezza, imposta quello specificato da `s`. Se esiste già un oggetto `SecurityManager` impostato, a esso viene richiesto il permesso di sostituirlo e, in caso negativo, viene gettata un'eccezione di tipo `SecurityException`.

14.6.3 Altri metodi utili

- `public static void exit(int stato)`: permette di interrompere l'esecuzione dell'elaborazione restituendo al sistema operativo il codice specificato in `stato`. L'invocazione di questo metodo è del tutto equivalente all'invocazione `Runtime.getRuntime().exit(stato)`.
- `public static void gc()`: l'invocazione di questo metodo richiede alla Virtual Machine di liberare la memoria da oggetti inutilizzati; `gc` sta per *garbage collector* (raccoglitore di spazzatura). L'invocazione di questo metodo è del tutto equivalente all'invocazione `Runtime.getRuntime().gc()`.
- `public static void runFinalization()()`: l'invocazione di questo metodo richiede alla Virtual Machine di eseguire il metodo `finalize` per tutti gli oggetti scaricati per i quali non è stato ancora eseguito. L'invocazione di questo metodo è del tutto equivalente all'invocazione `Runtime.getRuntime().runFinalization()`.
- `public static long currentTimeMillis()`: restituisce il numero di millisecondi trascorsi dalle 00:00:00 del 1 gennaio 1970.
- `public void loadLibrary(String nomeLib)`: permette il caricamento di librerie di codice eseguibile da utilizzare nei metodi di tipo `native`. Le modalità di utilizzo delle librerie variano per ogni sistema operativo. L'invocazione di questo metodo è del tutto equivalente all'invocazione `Runtime.getRuntime().loadLibrary(String nomeLib)`.
- `public void load(String nomeFile)`: come il precedente ma accetta nomi di file qualsiasi, anziché nomi di librerie. L'invocazione di questo metodo è del tutto equivalente all'invocazione `Runtime.getRuntime().load (String nomeFile)`.

Domande di verifica

1. A cosa è utile e come si utilizza l'interfaccia `Map<K,V>`?
2. Quali sono i costruttori e i metodi di `Date`?
3. Quando si utilizza il metodo `main`?
4. A cosa risulta utile la classe `SecurityManager`?
5. A cosa serve il metodo `exec` di `Runtime` e come si usa?
6. Quali sono le principali proprietà del sistema che ci si aspetta definite in `Proprieties` di `System`?
7. Che effetto hanno i metodi `getSecurityManager` e `setSecurityManager`?
8. Descrivere i metodi `exit`, `gc`, `runFinalization`, `currentTimeMillis`, `loadLibrary` e `load`.

Esercizi

1. Scrivere una classe che visualizzi le proprietà del proprio sistema Java.
2. Scrivere una classe che visualizzi la data corrente.
3. Scrivere una classe che visualizzi il giorno della settimana di una data impostata sulla riga di comando.
4. Scrivere una classe che richieda una data e visualizzi i giorni di differenza con la data odierna.
5. Verificare il codice di uscita di un programma Java che si interrompe per un'eccezione non catturata.
6. Scrivere una classe che lanci un programma con interfaccia grafica disponibile sul proprio sistema e ne causi la terminazione dopo 5 secondi.

Capitolo 15

Gestione dell'I/O

Obiettivi didattici

- Il package `java.io` per il controllo dei flussi
- Gestione di file e directory
- Flussi di input
- Flussi di output
- Accesso casuale
- Oggetti persistenti

La maggior parte dei programmi elabora dati provenienti da una qualche fonte esterna allo scopo di produrre dei risultati. È possibile naturalmente scrivere un programma che elabori esclusivamente dati già presenti all'interno del codice e magari non produca nessun risultato, ma sarebbe di poca utilità pratica. In generale possiamo quindi pensare a un programma come a una macchina che riceve un flusso di dati in ingresso, dati di *input*, e fornisce un flusso di dati in uscita, dati di *output*. In Java l'input e l'output, o, come si dice più brevemente, l'*I/O* è organizzato attorno al concetto di *flusso* (*stream*). Un flusso è una sequenza di elementi, tipicamente di 8 o 16 bit, che fluiscono da una *fonte* al programma e/o dal programma a una *destinazione*. Il package `java.io` fornisce le classi che permettono di gestire questi flussi e si appoggia principalmente su quattro classi:

- la classe astratta `InputStream`, per l'input di dati in byte;
- la classe astratta `Reader`, per l'input di caratteri Unicode di 16 bit;
- la classe astratta `OutputStream` per l'output di dati in byte;
- la classe astratta `Writer`, per l'output di caratteri Unicode di 16 bit.

Le classi `InputStream` e `Reader`, così come le classi `OutputStream` e `Writer`, offrono le stesse funzionalità e si differenziano solo per il fatto di lavorare con byte o con caratteri Unicode. Le classi concrete derivate da `Reader` e `Writer` hanno bisogno rispettivamente di un'istanza di una classe concreta derivata da `InputStream` e `OutputStream` per poter effettuare la connessione con un file del sistema operativo, per cui si può affermare che tutti i dati in ingresso e in uscita da un programma Java sono gestiti per mezzo di classi derivate da `InputStream` e `OutputStream` con solo le seguenti eccezioni:

- dati di sistema operativo, come per esempio la data;
- dati acquisiti tramite interfaccia grafica;
- dati acquisiti e/o scritti tramite la classe `RandomAccessFile`, che fa sempre parte del package `java.io` e permette di accedere a file su disco con accesso casuale consentendo di effettuare sia letture che scritture.

Abbiamo già visto come le classi `InputStream` e `OutputStream` possano essere impiegate per leggere dalla tastiera e scrivere su un terminale a caratteri, ma il loro uso può essere esteso a file su disco, database, porte seriali o parallele, dispositivi di backup, connessioni di rete, comunicazioni tra processi, array di byte ecc.

Il package `java.io` contiene attualmente 50 classi e 12 interfacce; in questo capitolo ci limiteremo a trattare quelle più interessanti e di utilizzo più generale.

15.1 Classe `File`

La classe `File` originariamente descriveva le proprietà di un file su disco; dalla versione 1.4 può descrivere anche le proprietà di una risorsa di rete. Java è nato su UNIX dove il file system è strutturato in modo gerarchico, ma, data la sua vocazione a essere un linguaggio indipendente dalla piattaforma, è stato fatto uno sforzo per rendere la classe `File` più generale possibile. Un file per Java è individuato da un *percorso* (path) composto da due parti, la directory e il nome, che sono divisi da un carattere particolare detto *carattere separatore*. Il carattere separatore varia da sistema operativo a sistema operativo, la classe riconosce in ogni caso il separatore usato su UNIX che è la barra `'/'`.

Una directory in UNIX è un file particolare che al suo interno contiene un elenco di nomi di file e/o altre directory. La classe `File` adotta questo approccio indipendentemente dalla struttura realmente utilizzata sul sistema operativo ospite. Per questo motivo una sua istanza può descrivere sia un file che una directory che una risorsa di rete. Secondo questa logica, sono disponibili i seguenti costruttori:

- `public File(String dir, String n)`: costruisce un'istanza che rappresenta un elemento file o directory individuato dal sistema operativo col nome `n` e posto nella directory `dir`. Se il parametro `dir` contiene `null`, il parametro `n` viene considerato come un path completo, altrimenti viene fatta la concatenazione tra `dir` e `n` utilizzando il separatore opportuno. Se `n` contiene `null` viene gettata un'eccezione di tipo `NullPointerException`.

- `public File(String n)`: questo metodo è equivalente a `File(null, n)`.
- `public File(File dir, String nome)`: costruisce un'istanza che rappresenta un elemento file o directory individuato dal sistema operativo col nome `n` e posto nella directory rappresentata dall'oggetto `dir`. Se `nome` contiene `null` viene gettata un'eccezione `NullPointerException`.`public File(String n)`: questo metodo è equivalente a `File(null, n)`.
- `public File(URI uri)`: costruisce un'istanza che rappresenta una risorsa di rete il cui *Uniform Resource Identifier* (Identificatore Uniforme di Risorsa) è passato come argomento. Se `nome` contiene `uri` viene gettata un'eccezione `NullPointerException`.

Un'istanza di questa classe fornisce informazioni sul file e può venire utilizzata per identificare la fonte e/o la destinazione di un flusso di dati. Vediamo alcuni metodi che permettono di ottenere informazioni sul file, tenendo conto del fatto che la loro rappresentazione è quanto più possibile indipendente dal sistema operativo.

- `public boolean exists()`: restituisce `true` se il file esiste, `false` altrimenti.
- `public boolean isFile()`: restituisce `true` se il file esiste ed è un file normale, `false` altrimenti.
- `public boolean isDirectory()`: restituisce `true` se il file esiste ed è una directory, `false` altrimenti.
- `public boolean canRead()`: restituisce `true` se il file esiste ed è permesso leggerlo, `false` altrimenti.
- `public boolean canWrite()`: restituisce `true` se il file esiste ed è permesso scriverci, `false` altrimenti.
- `public long lastModified()`: restituisce un numero che rappresenta il momento dell'ultima modifica. Il valore restituito rappresenta il numero di millisecondi trascorsi dalle 00:00:00 del 1 gennaio 1970.
- `public long length()`: restituisce la lunghezza del file.

Tutti i metodi precedenti possono gettare un'eccezione di tipo `SecurityException` nel caso sia presente un security manager che vieta il reperimento delle informazioni. Un'eccezione di questo tipo può essere gettata anche dalle seguenti operazioni, che sono le uniche che possono modificare il file.

- `public boolean renameTo(File dest)`: prova a rinominare il file o la directory in base al contenuto di `dest` e restituisce `true` se l'operazione ha avuto successo, `false` altrimenti.
- `public boolean delete()`: prova a eliminare il file o la directory e restituisce `true` se l'operazione ha avuto successo, `false` altrimenti.
- `public void deleteOnExit()`: richiede che il file o la directory siano cancellati al momento del termine dell'esecuzione della Java Virtual Machine. La cancellazione può avvenire solo se l'esecuzione termina normalmente. Una volta

richiesta la cancellazione di un file, essa non è annullabile, quindi questo metodo va usato con cautela.

- `boolean setExecutable(boolean eseguibile, boolean proprietario)`: rende il file eseguibile o meno a seconda del parametro `eseguibile`. Il parametro `proprietario` indica se la proprietà deve essere applicata solo al proprietario del file o a tutti. Restituisce `true` se l'operazione ha successo.
- `boolean setExecutable(boolean eseguibile)`: equivalente a `setExecutable (eseguibie, true)`.
- `setReadable(boolean leggibile, boolean proprietario)`: rende il file leggibile o meno a seconda del parametro `leggibile`. Il parametro `proprietario` indica se la proprietà deve essere applicata solo al proprietario del file o a tutti. Restituisce `true` se l'operazione ha successo.
- `setReadable(boolean leggibile)`: equivalente a `setReadable (leggibile, true)`.
- `setWritable(boolean scrivibile, boolean proprietario)`: rende il file scrivibile o meno a seconda del parametro `scrivibile`. Il parametro `proprietario` indica se la proprietà deve essere applicata solo al proprietario del file o a tutti. Restituisce `true` se l'operazione ha successo.
- `setWritable(boolean scrivibile)`: equivalente a `setWritable (scrivibile, true)`.
- `boolean setLastModified(long tempo)`: imposta la data e l'ora dell'ultima modifica fatta sul file. Il parametro `tempo` indica il numero di millisecondi passati dal primo gennaio 1970. Restituisce `true` se l'operazione ha successo.
- `boolean setReadOnly()`: rende il file accessibile solo in lettura. Restituisce `true` se l'operazione ha successo.

Informazioni sul nome del file e sulla sua locazione all'interno del file system possono essere ottenute con i successivi metodi, che non gettano eccezioni.

- `public String getName()`: restituisce il nome del file senza la directory.
- `public String getPath()`: restituisce il path completo del file.
- `public String getAbsolutePath()`: restituisce il path assoluto del file.
- `public String getParent()`: restituisce la directory del file.

Nel caso che il file corrisponda a una directory, si possono usare i seguenti metodi per il reperimento del contenuto.

- `public String[] list()`: restituisce un array di stringhe in cui ciascun elemento contiene il nome di un elemento (file o directory) contenuto nella directory.
- `public String[] list(FilenameFilter filtro)`: come il precedente, permette inoltre di selezionare solo i nomi con determinate caratteristiche definite da `filtro`. `FilenameFilter` è un'interfaccia che ha definito come unico me-

todo `boolean accept (File dir, String nome)` il quale deve restituire `true` se il nome corrisponde ai criteri desiderati. Il metodo `list` invoca `accept` per ogni elemento e include nell'array da restituire solo quelli che corrispondono ai criteri.

- `public boolean mkdir()`: crea una directory corrispondente alle informazioni contenute nell'istanza.

Nell'esempio vengono utilizzate delle istanze della classe `File` per scrivere una semplice classe che, data una directory di partenza, visualizza tutti i file e tutte le sottodirectory ricorsivamente, fornendo per ciascun elemento l'indicazione dello spazio occupato complessivamente.

```
import java.io.*;

public class UsoDelDisco
{
  static public void main (String argv[]) {
    UsoDelDisco ist = new UsoDelDisco(argv);
  }

  UsoDelDisco (String argv[]) {
    if (argv.length != 1) {
      System.err.println ("Uso: java " +
                          this.getClass().getName() +
                          " <directory>");
      return;
    }
    leggiDir (argv[0]);
  }
  private int leggiDir (String file) {
    int Return = 0;
    File f = new File(file);
    if (f.exists()) {
      Return += f.length();
      if (f.isDirectory()) {
        String list[] = f.list();
        for (int i = 0; i < list.length; i++)
          Return += leggiDir (file + File.separator + list[i]);
      }
      System.out.println (file + '=' + Return);
    }
    return Return;
  }
}
```

Notare che come separatore di file viene usata la variabile statica `File.separator` che è di tipo `String` e contiene lo stesso valore della proprietà di sistema

`file.separator`, cioè appunto il carattere separatore riconosciuto dal sistema operativo. È disponibile anche la variabile `File.separatorChar` di tipo `char` che contiene lo stesso valore.

15.2 Flussi di input

La classe astratta `InputStream` comprende i metodi per la lettura di un flusso. Tutti i metodi descritti nel seguito gettano un'eccezione di tipo `IOException` in caso di condizione d'errore.

- `public abstract int read()`: legge un byte dallo stream e lo restituisce nel valore di ritorno come un intero compreso tra 0 e 255. Nel caso non sia disponibile nessun byte, come quando, per esempio, ci si trova alla fine del file, viene restituito il valore -1.
- `public int read(byte[] b, int off, int len)`: legge un numero di byte non superiore a `len` e li memorizza nell'array `b` a partire dall'indice `off`. Restituisce il numero di byte letti o -1 in caso non vi siano byte disponibili da leggere.
- `public int read(byte[] b)`: corrisponde all'invocazione `read(b, 0, b.length)`.
- `public long skip(long n)`: cerca di saltare i successivi `n` byte. Nei casi in cui è possibile saltare solo un numero di byte inferiore a quelli specificati in `n`, come per esempio se si raggiunge la fine del file, viene saltato questo numero inferiore di byte. Il metodo restituisce il numero di byte saltati.
- `public int available()`: restituisce un numero di byte sicuramente disponibili a una successiva lettura. Questo metodo ha il suo migliore utilizzo nelle sottoclassi che ricevono dati da una fonte che li invia irregolarmente come, per esempio, una connessione di rete o una porta seriale.
- `public void close()`: chiude lo stream, che non può più essere riaperto tramite questa istanza.
- `public boolean markSupported()`: restituisce `true` se lo stream supporta i metodi `mark` e `reset` descritti di seguito, `false` altrimenti.
- `public void mark(int limite)`: permette di 'marcare' la posizione del flusso corrente in modo tale che sia possibile riposizionarsi in tale punto con l'invocazione del metodo `reset` per riletture successive. Se si leggono più di `limite` byte dal punto marcato, la marcatura viene persa.
- `public void reset()`: riposiziona il flusso nel punto marcato tramite il metodo `mark`.

Si può notare che esiste un metodo `close`, ma non esiste un corrispondente metodo `open`, questo perché nelle classi derivate la connessione del flusso viene aperta dal costruttore.

La sottoclasse di `InputStream` più comunemente usata è `FileInputStream`, che permette la lettura di file su file system. Tale classe ha i costruttori:

- `public FileInputStream(String path)`: apre il file individuato da `path`. Nel caso il file non esista oppure non possa essere aperto per qualche altro motivo, viene gettata un'eccezione di tipo `FileNotFoundException`.
- `public FileInputStream(File file)`: apre il file corrispondente all'oggetto `file`. Nel caso il file non esista oppure non possa essere aperto per qualche altro motivo, viene gettata un'eccezione di tipo `FileNotFoundException`.
- `public FileInputStream(FileDescriptor fdObj)`: crea una nuova istanza usando la connessione già esistente rappresentata da `fdObj`. `FileDescriptor`; infatti è una classe che rappresenta una connessione già instaurata. Un oggetto di questo tipo è ottenibile da qualsiasi oggetto di tipo `FileInputStream`, aperto correttamente, tramite il metodo `getFD()`.

La classe `Reader` ha lo stesso scopo e metodi analoghi alla classe `InputStream` con l'unica differenza che tratta con caratteri a 16 bit, anziché con byte. Da essa deriva indirettamente la classe `FileReader` che ha costruttori analoghi a `FileInputStream`.

Per esempio vediamo una classe che, dato un nome di file sulla linea di comando, ne esegue la visualizzazione sul terminale.

```
import java.io.*;

public class VediFile
{
   static public void main (String argv[]) {
      VediFile ist = new VediFile(argv);
   }

   VediFile (String argv[]) {
      if (argv.length != 1) {
         System.err.println ("Uso: java " +
                             this.getClass().getName() +
                             " <file>");
         return;
      }
         FileInputStream in;
      try {
           in = new FileInputStream(argv[0]);
      } catch (Exception e) {
         System.err.println ("Errore apertura file :" +  e);
         return;
      }
      int c;
      try {
         while ((c = in.read()) > 0)
```

```
            System.out.print ((char) c);
         in.close();
      } catch (Exception e) {
         System.err.println ("Errore lettura file :" +  e);
      }
   }
}
```

15.3 Flussi di output

La classe astratta `OutputStream` comprende i metodi per la scrittura sui flussi e i suoi metodi sono speculari a quelli della classe `InputStream`.

- `public abstract void write(int b)`: scrive un byte nello stream corrispondente agli 8 bit meno significativi dell'intero `b`. In pratica `b` può valere un intero compreso tra 0 e 255. Nel caso che il byte non possa essere scritto, come quando, per esempio, lo stream è chiuso, viene gettata un'eccezione di tipo `IOException`.
- `public void write(byte[] b, int off, int len)`: scrive il numero di byte specificato da `len` prendendoli dall'array `b` a partire dall'indice `off`. Anche in questo caso se i byte non possono essere scritti, viene gettata un'eccezione `IOException`.
- `public void write(byte[] b)`: corrisponde all'invocazione `write(b, 0, b.length)`.
- `public void flush()`: se la classe implementa una bufferizzazione dei dati in uscita, questo metodo richiede l'invio dei dati eventualmente presenti nel buffer verso la destinazione, altrimenti non esegue nessuna operazione.
- `public void close()`: chiude lo stream, che non può più essere riaperto tramite questa istanza.

Anche in questo caso la sottoclasse di `OutputStream` più utilizzata è quella che permette la scrittura su file e cioè `FileOutputStream`, che oltre ad avere dei costruttori del tutto analoghi a quelli di `FileInputStream`, ha il seguente

- `FileOutputStream(String path, boolean append)`: apre il file individuato da `path`. Se il parametro `append` vale `true`, i byte sul file vengono 'appesi' al fondo del file aperto, in caso contrario invece il contenuto del file viene cancellato e le scritture vengono effettuate dall'inizio.

Vediamo una classe d'esempio che, dati due nomi di file, esegue una copia del primo sul secondo.

```
import java.io.*;

public class CopiaFile {
  static public void main (String argv[]) {
```

```
    CopiaFile ist = new CopiaFile(argv);
  }

  CopiaFile (String argv[]) {
    if (argv.length != 2) {
      System.err.println ("Uso: java " +
                          this.getClass().getName() +
                    " <file sorgente> <file destinazione>");
      return;
    }
      FileInputStream in;
      FileOutputStream out;
    try {
        in = new FileInputStream(argv[0]);
        out = new FileOutputStream(argv[1]);
    } catch (Exception e) {
      System.err.println ("Errore apertura file :" +  e);
      return;
    }
    int c;
    try {
      while ((c = in.read()) > 0)
         out.write (c);
      in.close();
      out.close();
    } catch (Exception e) {
      System.err.println ("Errore copia file :" +  e);
      return;
    }
  }
}
```

Va sottolineato il fatto che se il file destinazione non esiste, viene creato automaticamente, mentre se esiste, esso viene completamente ricoperto.

La classe `Writer` è del tutto analoga a `OutputStream`, con la differenza che tratta caratteri a 16 bit Unicode, anziché byte. Essa inoltre ha i seguenti due metodi per la scrittura diretta di stringhe.

- `public void write(String s, int off, int len)`: scrive il numero di caratteri a 16 bit specificato da `len` prendendoli dalla stringa `s` a partire dall'elemento in posizione `off`.
- `public void write(String s)`: corrisponde all'invocazione `write(s, 0, s.length())`.

Da essa deriva la classe `FileWriter` che ha costruttori analoghi a `FileOutputStream`.

La classe `PrintStream`, che è la classe d'appartenenza degli oggetti statici `System.out` e `System.err`, deriva indirettamente dalla classe `OutputStream` e aggiunge a questa una serie di metodi `print` e `println` che consentono la visualizzazione, per esempio su terminale o su stampante, di qualsiasi tipo primitivo e oggetto. Per la visualizzazione degli oggetti, questi metodi sfruttano il metodo `toString`. A differenza dei metodi di `OutputStream`, quelli della classe `PrintStream` non gettano eccezioni, ma in caso di errore modificano un flag il cui valore può essere controllato tramite il metodo `public boolean checkError()`, che restituisce `true` in caso si siano verificati dei problemi. Il costruttore di questa classe richiede come parametro un oggetto di tipo `OutputStream` connesso con un dispositivo di output.

Analogamente la classe `PrintWriter` ha le stesse funzionalità di `PrintStream`, ma deriva da `Writer` ed è quindi da preferire a `PrintStream` per la visualizzazione di dati di tipo stringa in quanto tratta caratteri Unicode, anziché byte. Il costruttore di questa classe richiede come parametro un oggetto di tipo `Writer` connesso con un dispositivo di output.

15.4 Accesso casuale

Le classi viste fino ad ora servono per trattare flussi di dati in entrata e in uscita dal programma e consentono di scambiare dati, oltre che con file, anche con dispositivi di input/output, connessioni in rete ecc. Oltre a questa funzione di scambio di dati, i file su disco possono essere utilizzati come *base di dati* permanente che un set di programmi scrive, aggiorna e rilegge in base alle richieste dell'utente. Per un tale utilizzo, il paradigma dei flussi si rivela troppo limitato, in quanto si ha la necessità di poter scrivere, leggere e riscrivere porzioni di un file a discrezione del programmatore.

La classe `RandomAccessFile` viene incontro a queste necessità; per mezzo di essa un file viene trattato come un array di byte memorizzato sul file system. Possiamo considerare che per ogni file aperto, esista un indice implicito di questo array, detto *file pointer*, che viene incrementato a ogni lettura di un numero corrispondente a quello dei byte letti, ma che può essere impostato a un qualsiasi valore tramite un metodo opportuno. Un file può essere aperto o solo in lettura o in lettura e scrittura. La classe `RandomAccessFile` non è astratta e anzi non ha sottoclassi nel package `java.io` e molti dei suoi metodi sono dichiarati `final` in quanto essa ha già tutte le funzionalità considerate necessarie. Due costruttori sono disponibili per la creazione di un oggetto della classe:

- `public RandomAccessFile(String path, String mode)`: apre il file individuato da `path`. Nel caso il file non possa essere aperto per qualche motivo, viene gettata un'eccezione di tipo `FileNotFoundException`. La stringa `mode` può avere solo due valori, o `"r"` o `"rw"`, altrimenti viene gettata un'eccezione di tipo `IllegalArgumentException`. Nel primo caso il file viene aperto solo in lettura, mentre nel secondo viene aperto in lettura e scrittura.

- `public RandomAccessFile(File file, String mode)`: apre il file corrispondente all'oggetto `file`. Nel caso il file non esista oppure non possa essere aperto per qualche altro motivo, viene gettata un'eccezione di tipo `FileNotFoundException`. La stringa `mode` ha lo stesso significato descritto nel metodo precedente.

Si può creare un file inesistente invocando uno dei costruttori in modalità di lettura e scrittura.

La classe `RandomAccessFile` implementa le interfacce `DataOutput` e `DataInput`. La prima di queste definisce una serie di metodi che hanno lo scopo di trasformare qualsiasi dato primitivo in una sequenza di byte, mentre la seconda definisce i metodi per la ricostruzione dei tipi primitivi a partire da sequenze di byte.

I metodi definiti di `DataOutput` gettano un'eccezione di tipo `IOException` in caso di errore e sono:

- `public void write(int b)`: scrive un byte, corrispondenti agli 8 bit meno significativi dell'intero `b`, nello stream. In pratica `b` può valere un intero compreso tra 0 e 255.
- `public void write(byte[] b, int off, int len)`: scrive il numero di byte specificato da `len` prendendoli dall'array `b` a partire dall'indice `off`.
- `public void write(byte[] b)`: corrisponde all'invocazione `write(b, 0, b.length)`.
- `public void writeBoolean(boolean v)`: nel caso in cui l'argomento sia `true` scrive un byte che vale 1, altrimenti scrive un byte che vale 0.
- `public void writeByte(int v)`: scrive un byte corrispondente a `(byte)v`.
- `public void writeShort(int v)`: scrive due byte corrispondenti a `(short)v`.
- `public void writeChar(int v)`: scrive due byte corrispondenti a `(char)v`.
- `public void writeInt(int v)`: scrive quattro byte corrispondenti all'argomento `v`.
- `public void writeLong(long v)`: scrive otto byte corrispondenti all'argomento `v`.
- `public void writeFloat(float v)`: scrive quattro byte corrispondenti all'argomento `v`.
- `public void writeDouble(double v)`: scrive otto byte corrispondenti all'argomento `v`.
- `public void writeBytes(String s)`: ciascun byte della stringa `s` viene convertito in un byte e viene scritto.
- `public void writeChars(String s)`: la stringa `s` viene convertita in un array di `char` e viene scritta.
- `public void writeUTF(String s)`: la stringa `s` viene convertita in un array di byte secondo il formato UTF-8 modificato per Java, e viene scritto.

✔ **NOTA**

Il formato UTF (Universal Transfer Format) permette la rappresentazione di stringhe di caratteri Unicode in un formato in cui ciascun carattere ha lunghezza variabile. Questo formato prevede la memorizzazione anche della lunghezza della stringa, il che rende semplice la rilettura per mezzo del metodo `readUTF` *illustrato nel seguito.*

L'interfaccia `DataInput` definisce dei metodi speculari a quelli appena descritti, i quali, pur essendo analoghi per funzionalità a quelli visti in `InputStream`, in generale non restituiscono il numero di byte letti e in caso di lettura oltre la fine del file gettano un'eccezione di tipo `EOFException`. Se incontrano altri problemi invece si comportano come la maggioranza dei metodi visti in questo capitolo gettando un'eccezione di tipo `IOException`.

- `public void readFully(byte[] b, int off, int len)`: legge un numero di byte non superiore a `len` e li memorizza nell'array `b` a partire dall'indice `off`.
- `public void readFully(byte[] b)`: corrisponde all'invocazione `readFully(b, 0, b.length)`.
- `public int skipBytes(int n)`: cerca di saltare i successivi `n` byte. Nei casi in cui è possibile saltare solo un numero di byte inferiore a quelli specificati in `n`, come per esempio se si raggiunge la fine del file, viene saltato questo numero inferiore di byte. Il metodo restituisce il numero di byte saltati e non getta mai un'eccezione di tipo `EOFException`.
- `public boolean readBoolean()`: legge un byte e restituisce `false` se vale 0 e `true` altrimenti.
- `public byte readByte()`: legge un byte e lo restituisce nel valore di ritorno come un byte compreso tra -128 e 127.
- `public int readUnsignedByte()`: legge un byte e lo restituisce nel valore di ritorno come un intero compreso tra 0 e 255.
- `public short readShort()`: legge due byte, li converte in un elemento di tipo `short` e restituisce il valore ottenuto.
- `public int readUnsignedShort()`: legge due byte, li converte in un intero che può avere valori compresi tra 0 e 65535 e restituisce il valore ottenuto.
- `public char readChar()`: legge due byte, li converte in un elemento di tipo `char` e restituisce il valore ottenuto.
- `public int readInt()`: legge quattro byte, li converte in un elemento di tipo `int` e restituisce il valore ottenuto.
- `public long readLong()`: legge otto byte, li converte in un elemento di tipo `long` e restituisce il valore ottenuto.
- `public float readFloat()`: legge quattro byte, li converte in un elemento di tipo `float` e restituisce il valore ottenuto.

- `public double readDouble()`: legge otto byte, li converte in un elemento di tipo `double` e restituisce il valore ottenuto.
- `public String readLine()`: legge i successivi byte fino a che non viene incontrato il carattere di fine riga `'\n'`. Ogni byte viene convertito in un carattere e dall'unione di questi viene generata una stringa che viene restituita. Il carattere di fine riga non fa parte della stringa restituita. Il carattere di ritorno a capo `'\r'` viene sempre ignorato.
- `public String readUTF()`: legge una stringa in formato UTF-8 modificato per Java, la converte in un oggetto della classe `String` e restituisce la referenza dell'oggetto ottenuto.

La classe `RandomAccessFile`, oltre a implementare i metodi delle interfacce `DataOutput` e `DataInput` mette a disposizione anche i metodi:

- `public abstract int read()`: analogo a quello definito nella classe `InputStream`.
- `public int read(byte[] b, int off, int len)`: analogo a quello definito nella classe `InputStream`.
- `public int read(byte[] b)`: analogo a quello definito nella classe `InputStream`.
- `public native void seek(long pos)`: imposta il file pointer nella posizione specificata in byte da `pos`. La posizione 0 indica l'inizio del file.
- `public native long getFilePointer()`: restituisce la posizione attuale del file pointer, misurata in byte.
- `public native long length()`: restituisce la lunghezza del file, misurata in byte.
- `public final FileDescriptor getFD()`: restituisce l'oggetto che rappresenta la connessione con il file.
- `public native void close()`: chiude la connessione col file, che non può più essere riaperta tramite questa istanza.

Tutti i metodi precedenti possono gettare eccezioni di tipo `IOException`.

Vediamo per esempio una classe che, dati sulla linea di comando il nome di un file e due caratteri, sostituisce nel file tutte le occorrenze del primo carattere con il secondo. Supponendo di lavorare con caratteri di un byte, la classe potrebbe essere la seguente.

```
import java.io.*;

public class CambiaByte {
  static public void main (String argv[]) {
     CambiaByte ist = new CambiaByte(argv);
  }
```

```
  CambiaByte (String argv[]) {
    if (argv.length != 3 ||
        argv[1].length() != 1 ||
        argv[2].length() != 1) {
        System.err.println ("Uso: java " +
                            this.getClass().getName() +
                            " <file> <byte> <byte>");
        return;
    }
        RandomAccessFile inOut;
    try {
        inOut = new RandomAccessFile(argv[0], "rw");
    } catch (Exception e) {
        System.err.println ("Errore apertura file :" +  e);
        return;
    }
    byte vecchio = (byte)argv[1].charAt(0);
    byte nuovo = (byte)argv[2].charAt(0);
    byte c;
    try {
        for ( ; ; ) {
          c = inOut.readByte();
          if (c == vecchio) {
            inOut.seek (inOut.getFilePointer() - 1);
            inOut.writeByte (nuovo);
          }
        }
    } catch (EOFException e) {
    } catch (Exception e) {
       System.err.println ("Errore sostituzione in file :" + e);
    }
    try {
       inOut.close();
    } catch (Exception e) {
       System.err.println ("Errore chiusura file :" + e);
    }
  }
}
```

Va sottolineato che se il file indicato sulla linea di comando non esiste, viene creato automaticamente con lunghezza 0.

15.5 Gli oggetti persistenti

Abbiamo esaminato nei precedenti paragrafi alcune delle numerose classi del package `java.io` che permettono di leggere e scrivere flussi di byte, anche sotto forma di tipi primitivi e stringhe. Utilizzando tali classi, è possibile in teoria memorizzare anche un oggetto completo, riconducendolo dapprima a un insieme di tipi primitivi e quindi trasferendoli sul file in sequenza. Questo tipo di operazione è detta *serializzazione* e la sua implementazione si presenta in realtà assai complessa. Infatti, non è sufficiente conoscere semplicemente il formato della classe cui appartiene l'oggetto che si vuole memorizzare, ma è necessario conoscere il formato delle sue superclassi e di tutte le classi e relative superclassi di tutti gli oggetti contenuti. Sia le superclassi che gli oggetti contenuti, infatti, debbono essere a loro volta serializzati e così via fintanto che si arriva a oggetti che contengono solo tipi primitivi.

Mentre si effettua questa scansione ricorsiva bisogna inoltre prestare attenzione a riconoscere le *dipendenze cicliche* che porterebbero il programma a un loop infinito. Un esempio di dipendenza ciclica si ha quando un oggetto `A` contiene una referenza a un oggetto `B`, che a sua volta contiene una referenza a un oggetto `C`, che contiene infine una referenza all'oggetto `A`.

Fortunatamente il package `java.io` include due classi, `ObjectOutputStream` e `ObjectInputStream`, che implementano in modo automatico rispettivamente la serializzazione e l'operazione inversa, cioè la *deserializzazione* di oggetti.

La classe `ObjectOutputStream` deriva da `OutputStream` e implementa l'interfaccia `ObjectOutput`, che a sua volta deriva da `DataOutput`. L'unico costruttore pubblico di questa classe ha come parametro un oggetto della classe `OutputStream` che definisce la destinazione degli oggetti serializzati. Il metodo che caratterizza questa classe è `public void writeObject(Object obj)` che esegue la serializzazione e la scrittura nel flusso di un oggetto. La serializzazione viene effettuata automaticamente, ma l'oggetto da serializzare deve implementare l'interfaccia `Serializable`, altrimenti viene gettata un'eccezione di tipo `NotSerializableException`. L'interfaccia `Serializable` non ha metodi e serve solo per dividere l'universo delle classi in due categorie, quelle che possono essere serializzate e quelle che non possono. Le classi fornite con Java sono in gran parte serializzabili. Nel caso si desideri rendere serializzabile una classe che deriva da o contiene una classe non serializzabile, questo è possibile solo se le classi non serializzabili hanno definito un costruttore senza argomenti che sia accessibile; gli oggetti appartenenti alle classi non serializzabili non vengono serializzati, per cui se si vuole memorizzarne i valori si deve procedere manualmente.

Non vengono serializzate né le variabili statiche né le variabili d'istanza dichiarate `transient`.

Uno o più oggetti scritti per mezzo della classe `ObjectOutputStream` possono essere riletti tramite la classe `ObjectInputStream` che deriva da `InputStream` e implementa l'interfaccia `ObjectInput` che a sua volta deriva da `DataInput`. Anche questa classe ha un unico costruttore che ha come parametro un oggetto della classe `InputStream` che definisce la fonte da cui ricuperare gli oggetti. Il metodo che effettua la deserializzazione è `public final Object readObject()`. Quando

un oggetto viene letto per mezzo di questo metodo, per prima cosa viene caricata in memoria la classe corrispondente se non è ancora disponibile, dopodiché viene creata un'istanza e inizializzata coi valori di default. A questo punto per gli oggetti appartenenti alle classi non serializzabili viene invocato il costruttore senza argomenti, mentre agli altri viene assegnato il valore letto dal flusso iniziando dagli oggetti appartenenti alle classi più vicine in ordine gerarchico alla classe `Object` e terminando con le classi più specifiche.

Osserviamo il seguente esempio.

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

class Base implements Serializable {
  char chr[];
  int num = 10;

  public Base (char [] chr, int num) {
    this.chr = chr;
    this.num = num;
  }
}

public class ScriviOggetto extends Base
{
  String nome;
  ScriviOggetto (String nome, char chr[], int num) {
    super (chr, num);
    this.nome = nome;
  }

  public String toString () {
    return  nome + "," + chr + "," + num;
  }

  public static void main (String argv[]) {
    char chr[]= { 'A','r','r','a','y' };
    ScriviOggetto so = new ScriviOggetto ("Stringa", chr,
                                          123456789);
    Date d = new Date();
    System.out.println (so + " " + d);
    try {
        FileOutputStream f = new FileOutputStream("prova");
        ObjectOutput  s  =  new  ObjectOutputStream(f);
        s.writeObject(so);
        s.writeObject(d);
        s.close();
```

```
    } catch (Exception e) {
        System.out.println ("Errore in scrittura:" + e);
    }
    leggi();
  }

  public static void leggi () {
    try {
        FileInputStream in = new FileInputStream("prova");
        ObjectInputStream s = new ObjectInputStream(in);
        Object o = s.readObject();
        Object d = s.readObject();
        System.out.println (o + " " + d);
    } catch (Exception e) {
        System.out.println ("Errore in lettura:" + e);
    }
  }
}
```

Come si vede, viene definita una classe `Base` che implementa l'interfaccia `Serializable` e da questa viene derivata una classe `ScriviOggetto`. Nel metodo `main` viene creata un'istanza di quest'ultima classe e un'istanza della classe `Date` contenente la data odierna. Entrambe le istanze vengono serializzate e trasferite nel file identificato sul file system dal nome `prova`. Viene infine invocato il metodo `leggi` che ha il compito di leggere il file `prova` e visualizzarne il contenuto. Il risultato dell'elaborazione è il seguente

```
Stringa,[C@1a46e30,123456789 Fri Aug 11 10:53:22 CEST 2006
Stringa,[C@8813f2,123456789 Fri Aug 11 10:53:22 CEST 2006
```

Notare come non sia stato necessario dichiarare che la classe `ScriviOggetto` implementa l'interfaccia `Serializable`, in quanto eredita questa proprietà dalla classe genitrice.

Modificando la dichiarazione della variabile num della classe `Base` utilizzando il modificatore `transient` nel modo seguente

```
transient int num = 10;
```

il risultato dell'elaborazione diverrebbe

```
Stringa,[C@1a46e30,123456789 Fri Aug 11 10:55:27 CEST 2006
Stringa,[C@8813f2,0 Fri Aug 11 10:55:27 CEST 2006
```

in quanto le variabili dichiarate in tal modo non vengono serializzate.

Se facciamo in modo che la classe `Base` non implementi più l'interfaccia `Serializable`, mentre modifichiamo la classe `ScriviOggetto` in modo tale che la implementi, ricompiliamo e lanciamo l'interprete, otteniamo un errore del seguente tipo:

```
Errore in lettura:java.io.InvalidClassException:
ScriviOggetto; ScriviOggetto; no valid constructor
```

dovuto al fatto che in fase di deserializzazione dell'oggetto è stato cercato un costruttore senza argomenti della classe non più serializzabile `Base`, e, non trovandolo, è stata gettata un'eccezione di tipo `InvalidClassException`. Aggiungendo il seguente costruttore alla classe suddetta

```
public Base () {
   this.chr = new char[0];
}
```

il risultato diviene il seguente

```
Stringa,[C@1a46e30,123456789 Fri Aug 11 10:59:15 CEST 2006
Stringa,[C@530daa,10 Fri Aug 11 10:59:15 CEST 2006
```

Creiamo e compiliamo ora la classe seguente nella stessa directory dove è stato sviluppato l'esempio precedente.

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

public class LeggiOggetto
{
   public static void main (String argv[]) {
     try {
         FileInputStream in = new FileInputStream("prova");
         ObjectInputStream s = new ObjectInputStream(in);
         Object o = s.readObject();
         Object d = s.readObject();
         System.out.println (o + " " + d);
     } catch (Exception e) {
         System.out.println ("Errore in lettura:" + e);
     }
   }
}
```

Lanciando l'esecuzione di `LeggiOggetto`, essa produce la corretta visualizzazione della classe `ScriviOggetto`, anche se nel codice non compare nessun riferimento a detta classe. Quest'ultima infatti viene caricata durante la fase di esecuzione in base alle informazioni ricuperate dal file. Provando infatti a eliminare `ScriviOggetto.class` dalla directory, l'esecuzione provoca un errore come il seguente

```
Errore in lettura:java.lang.ClassNotFoundException:
ScriviOggetto
```

Domande di verifica

1. In cosa si differenziano le classi `InputStream/OutputStream` dalle classi `Reader/Writer`?
2. Di quale package fa parte e quale funzionalità offre la classe `RandomAccessFile`?
3. Quali costruttori sono disponibili nella classe `File`?
4. Quali metodi sono disponibili per ottenere tutte le possibili informazioni su file e directory? Per modificarli? E per reperire il contenuto di una directory?
5. Come agisce ognuno dei metodi disponibili in `InputStream`?
6. Perché si usa la classe `FileInputStream` e quali costruttori possiede? E la classe `FileReader`?
7. Come agisce ognuno dei metodi disponibili in `OutStream`?
8. Perché si usa la classe `FileOutputStream` e quali costruttori possiede? E la classe `FileWriter`?
9. Descrivere la classe `RandomAccessFile`, i suoi costruttori, le interfacce `DataOutput` e `DataInput` e i loro metodi.
10. Qual è lo scopo delle classi `ObjectOutputStream` e `ObjectInputStream` e come possono essere utilizzate?

Esercizi

1. Scrivere una classe che accetti dalla linea di comando una directory e un valore e visualizzi tutti i file contenuti nella directory e nelle eventuali sottodirectory di dimensione superiore al valore impostato.
2. Modificare le classi riportate come esempi `VediFile` e `CopiaFile` in modo che, invece di leggere un byte per volta, ne leggano un certo numero in modo da migliorare le prestazioni.
3. Modificare la classe `CopiaFile` in modo che dia un messaggio di avvertimento nel caso si tenti di riscrivere su di un file esistente.
4. Scrivere una classe che confronti due file e determini se sono uguali o meno.
5. Scrivere una classe che permetta di concatenare due o più file.
6. Scrivere una classe che permetta di copiare tutti i file di una directory in un'altra.
7. Scrivere una classe che permetta di modificare il nome di un file.
8. Scrivere una classe che permetta di sostituire una parola all'interno di un file con un'altra di uguale lunghezza.

9. Scrivere un semplice programma per la gestione di un'agenda telefonica che usi la classe `RandomAccessFile` per la memorizzazione dei dati.

10. Scrivere un semplice programma per la gestione di un'agenda telefonica che sfrutti il meccanismo degli oggetti persistenti per la memorizzazione dei dati.

Capitolo 16

Interfaccia grafica

Obiettivi didattici

- Il package `java.awt` per gli ambienti grafici
- Eventi e oggetti *listener*
- Visualizzare oggetti grafici
- Lettura da tastiera
- Eventi dal cursore del mouse
- Gestione di colori, font e posizione del testo

Le interfacce grafiche, o GUI *Graphical User Interface*, hanno permesso la creazione di programmi più flessibili, più facili da usare e che non richiedono conoscenze sulla struttura dei computer. Le GUI hanno portato a un vero e proprio rovesciamento del modello di programmazione precedente; prima del loro avvento infatti, un programma seguiva il proprio flusso di elaborazione e interagiva con l'utente solo quando il codice stesso lo richiedeva. Un programma con interfaccia grafica invece tipicamente si mette semplicemente in attesa che l'utente interagisca con il computer e reagisce a ogni azione in modo opportuno. Una programmazione che supporta questo modello viene detta *guidata dagli eventi* (*event driven*).

Il supporto a questa grande libertà di azione dell'utente in tutte le situazioni possibili ha richiesto delle architetture piuttosto complesse, per cui la programmazione in ambiente grafico non è affatto semplice. In Java esistono due package che permettono la creazione di interfacce grafiche: `java.awt` e `javax.swing`. Cominceremo col vedere il primo dei due, sia perché è quello nato prima, sia perché, pur essendo di minore dimensione, permette di illustrare tutti i principi fondamentali. Swing del resto è costruito sopra AWT e contiene un numero molto elevato di classi, per cui ne parleremo piuttosto sommariamente in seguito.

L'acronimo AWT sta per *Abstract Window Toolkit*. Questo package contiene a sua volta altri dieci sottopackage e precisamente `java.awt.color`, `java.awt.datatransfer`, `java.awt.dnd`, `java.awt.event`, `java.awt.font`, `java.awt.geom`, `java.awt.im`, `java.awt.image`, `java.awt.image.renderable` e `java.awt.print`.

Esamineremo i concetti di base e introdurremo alcune delle classi principali al solo scopo di mettere in grado il lettore di poter sviluppare alcuni programmi interessanti oltreché di potersi muovere agevolmente tra manuali di riferimento e approfondire le proprie conoscenze.

16.1 L'AWT

Uno dei problemi maggiori che è stato affrontato nello sviluppo del package per l'interfaccia grafica di Java è dovuto al fatto di dover funzionare in ambienti molto differenti l'uno dall'altro. Quando è stato progettato il package `java.awt`, è stato necessario costruire un modello astratto che inglobasse i concetti generali comuni a tutti gli ambienti grafici in modo tale da renderlo capace di calarsi su qualsiasi architettura operativa. Il risultato è un insieme di classi che rappresentano il minimo comun denominatore degli ambienti grafici esistenti, ma sufficientemente flessibile per la realizzazione di programmi di qualsiasi tipo.

✔ NOTA

Dalla release 1.0 alla release 1.1 del JDK sono state effettuate modifiche importanti nell'AWT che riguardano in certa parte anche il paradigma concettuale, pur mantenendo la compatibilità all'indietro. Noi vedremo solo il nuovo modello, che è quello destinato a essere supportato anche nei rilasci successivi. Per provare gli esempi riportati in questo capitolo non è possibile quindi utilizzare un sistema di sviluppo Java che non sia allineato almeno alla release 1.1.

La classe principale del package è `Component` che rappresenta un oggetto grafico che ha una posizione e una dimensione sullo schermo, su cui è possibile scrivere o disegnare e che può ricevere *eventi* dall'utente. Per ora consideriamo evento un'azione dell'utente, come la pressione di un tasto o il click del mouse. Da questa classe astratta deriva la maggior parte delle classi del package, fra cui la classe astratta `Container` che rappresenta un oggetto grafico che può contenere al proprio interno oggetti grafici, compresi altri oggetti `Container`.

In generale un'istanza di `Component` può essere visualizzata solo se contenuta o collegata a un oggetto visibile; fanno eccezione gli oggetti della classe `Frame` che quindi si collocano come radici della gerarchia di contenimento. La parola *frame* in inglese significa 'cornice' e, infatti, questa classe, che deriva indirettamente da `Container`, rappresenta una finestra con un titolo e un bordo. Vediamo un primo esempio che consente la semplice visualizzazione di una finestra.

```
import java.awt.*;

public class FinestraVuota extends Frame
{
   public static void main (String argv[]) {
      FinestraVuota ist = new FinestraVuota();
     System.out.println ("Esco da main");
   }
   FinestraVuota () {
      setBounds (30, 10, 300, 200);
      setTitle (getClass().getName());
      setVisible(true);
   }
}
```

Il metodo `setBound`, definito nella classe `Component`, permette di impostare la posizione e la dimensione della finestra; i quattro argomenti infatti impostano rispettivamente le coordinate orizzontale e verticale dell'angolo in alto a sinistra e la larghezza e l'altezza della finestra. Le coordinate prendono come origine l'angolo in alto a sinistra dello schermo e sono espresse in *pixel* che è la dimensione del punto più piccolo rappresentabile sullo schermo. Le misure sono anch'esse espresse in pixel. Il metodo `setTitle` permette di visualizzare la stringa passata per argomento come titolo della finestra. Il metodo `setVisible` con argomento `true` infine rende visibile la finestra.

Compilando ed eseguendo questa classe, appare una normale finestra vuota (vedi Figura 16.1). Anche se lavorano in modalità grafica, i programmi Java hanno sempre associato un terminale a caratteri, detto *console*, che tipicamente è quello da cui si lancia l'applicazione. Grazie quindi alla `println` posta al termine del metodo `main`, si può notare come la finestra rimanga attiva anche se l'esecuzione di `main` termina quasi immediatamente; questo perché quando viene attivata la modalità grafica, vengono creati dei thread indipendenti.

Figura 16.1 Visualizzazione di una finestra

16.2 Eventi

Provando a chiudere la finestra tramite il menu di sistema o utilizzando i normali bottoni dedicati a questo scopo, ci si rende conto che non si ottiene nessun risultato e che l'unico modo per terminare l'esecuzione del programma è di premere il tasto di interruzione (tipicamente Ctrl-C o DEL) da console. Questo perché il comando di chiusura della finestra non fa altro che generare un evento e la nostra classe non è stata abilitata a trattarne. Secondo il modello implementato nell'AWT, se un oggetto grafico deve trattare degli eventi, deve *delegare* questo compito a un oggetto apposito, che può essere anche se stesso.

Un oggetto di questo tipo è detto *ascoltatore* (*listener*) e deve implementare un'interfaccia opportuna dipendente dal tipo di eventi da trattare. Le interfacce per gli oggetti listener derivano da `java.util.EventListener`, fanno parte del package `java.awt.event` e sono:

- `ActionListener`
- `AdjustmentListener`
- `AWTEventListener`
- `ComponentListener`
- `ContainerListener`
- `HierarchyBoundsListener`
- `HierarchyListener`
- `FocusListener`
- `InputMethodListener`
- `ItemListener`
- `KeyListener`
- `MouseListener`
- `MouseMotionListener`
- `MouseWheelListener`
- `TextListener`
- `WindowFocusListener`
- `WindowListener`
- `WindowStateListener`

Gli eventi che riguardano la modifica dello stato di una finestra, come la richiesta di chiusura, vengono trattati tramite l'interfaccia `WindowListener` che definisce i seguenti metodi:

- `public void windowClosed(WindowEvent evento)`: invocato quando la finestra è stata chiusa.
- `public void windowClosing(WindowEvent evento)`: invocato in caso di richiesta di chiusura.

- `public void windowIconified(WindowEvent evento)`: invocato quando una finestra viene ridotta a icona.
- `public void windowDeiconified(WindowEvent evento)`: invocato quando una finestra ridotta a icona viene ingrandita.
- `public void windowOpened(WindowEvent evento)`: invocato quando la finestra è stata aperta.
- `public void windowActivated(WindowEvent evento)`: invocato quando la finestra diventa attiva. In un ambiente a finestre esiste un'unica finestra attiva per volta. L'attivazione di una finestra viene eseguita tipicamente con il mouse o con la pressione di determinati tasti di sistema.
- `public void windowDeactivated(WindowEvent evento)`: invocato quando la finestra viene disattivata. La disattivazione di una finestra può essere causata dalla chiusura o dall'attivazione di un'altra finestra.

La classe `WindowEvent`, che fa parte anch'essa del package `java.awt.event`, rappresenta un evento che può essere ricevuto da una finestra. Da un'istanza di questa classe è possibile ricavare il tipo di evento e la finestra che lo ha ricevuto. Esiste una gerarchia di classi che descrive gli eventi che possono essere ricevuti da un oggetto grafico e che ha la sua radice in `AWTEvent` del package `java.awt` anche se tutte le sue sottoclassi appartengono al package `java.awt.event`.

Vediamo dunque come possiamo fare in modo di chiudere la finestra precedente.

```
import java.awt.*;
import java.awt.event.*;

public class Eventi extends Frame
{
   Ascoltatore asc = new Ascoltatore();
   public static void main (String argv[]) {
      Eventi ist = new Eventi();
   }
   Eventi () {
      setBounds (30, 10, 300, 200);
      setTitle (getClass().getName());
      setVisible(true);
      addWindowListener(asc);
   }
}

class Ascoltatore implements WindowListener
{
   public void windowClosed (WindowEvent evt) {
      System.out.println (evt);
      System.exit(0);
   }
```

```
    public void windowClosing(WindowEvent evt) {
        evt.getWindow().dispose();
        System.out.println (evt);
    }
    public void windowDeiconified(WindowEvent evt) {
        System.out.println (evt);
    }
    public void windowIconified(WindowEvent evt) {
        System.out.println (evt);
    }
    public void windowOpened(WindowEvent evt) {
        System.out.println (evt);
    }
    public void windowActivated(WindowEvent evt) {
        System.out.println (evt);
    }
    public void windowDeactivated(WindowEvent evt) {
        System.out.println (evt);
    }
}
```

Abbiamo creato una classe `Ascoltatore` che deriva semplicemente da `Object` e che implementa tutti i metodi dell'interfaccia `WindowListener`. Nella classe `Eventi` è stato dichiarato un oggetto `asc` di tipo `Ascoltatore` che è stato designato come listener della finestra per mezzo del metodo `public void addWindowListener(WindowListener e)` definito nella classe `Frame`. Tale classe definisce inoltre il metodo `public void removeWindowListener(WindowListener l)` che consente la rimozione di un listener.

Compilando ed eseguendo questa classe e provando a effettuare delle operazioni sulla finestra, come l'attivazione e disattivazione oppure la riduzione a icona, possiamo osservare sulla console la successione degli eventi. In particolare, provando a chiudere la finestra, sulla console appaiono i seguenti messaggi che ci permettono di seguire la sequenza di chiusura.

```
java.awt.event.WindowEvent[WINDOW_CLOSING,opposite=null,oldState=0,newState=0]
on frame0

java.awt.event.WindowEvent[WINDOW_CLOSED,opposite=null,oldState=0,newState=0]
on frame0
```

L'azione di chiudere la finestra ha provocato l'invocazione del metodo `windowClosing`; in questo metodo abbiamo ricuperato l'istanza della finestra dall'argomento `evt` per mezzo del metodo `getWindow` e su tale istanza abbiamo invocato il metodo `dispose` che causa la chiusura della finestra e quindi l'invocazione del metodo `windowClosed` nel quale abbiamo definitivamente interrotto l'elaborazione tramite l'invocazione di `System.Exit`.

Usando questo modello di delega possiamo fare in modo di avere oggetti che controllano il comportamento di più finestre, ma certo nell'esempio appena visto, siamo stati costretti a definire sette metodi, quando in realtà si desiderava trattarne solo due.

Per evitare questo fastidio, sono state definite delle classi, dette *adattatori*, che implementano le interfacce necessarie per gli oggetti listener, ma i cui metodi non eseguono alcuna operazione. Per esempio, la classe `WindowAdapter` implementa i metodi di `WindowListener` che però non eseguono nessun tipo di elaborazione. Ecco quindi che potremmo sfruttarla per ridurre il codice dell'esempio appena visto:

```
import java.awt.*;
import java.awt.event.*;

public class Eventi extends Frame
{
   Ascoltatore asc = new Ascoltatore();
   public static void main (String argv[]) {
      Eventi ist = new Eventi();
   }
   Eventi () {
      setBounds (30, 10, 300, 200);
      setTitle (getClass().getName());
      setVisible(true);
      addWindowListener(asc);
   }
}

class Ascoltatore extends WindowAdapter
{
   public void windowClosed (WindowEvent evt) {
      System.out.println (evt);
      System.exit(0);
   }
   public void windowClosing(WindowEvent evt) {
      evt.getWindow().dispose();
      System.out.println (evt);
   }
}
```

Gli oggetti grafici reagiscono agli eventi in modi notevolmente differenti, per cui tipicamente ciascuna classe grafica ha associati uno o più listener che non sono utilizzabili da altre classi. Per rendere evidente il fatto che una classe è utilizzabile solo da un'altra e contemporaneamente rendere più semplice lo scambio di dati tra istanze di classi che debbono funzionare in modo cooperativo, è conveniente usare le classi locali e/o anonime. L'esempio precedente può essere quindi trasformato come segue:

```
import java.awt.*;
import java.awt.event.*;

public class Eventi extends Frame
{
   WindowAdapter asc;
```

```
    public static void main (String argv[]) {
        Eventi ist = new Eventi();
    }
    Eventi () {
        setBounds (30, 10, 300, 200);
        setTitle (getClass().getName());
        setVisible(true);
        addWindowListener(new WindowAdapter() {
            public void windowClosed (WindowEvent evt) {
                System.out.println (evt);
                System.exit(0);
            }
            public void windowClosing(WindowEvent evt) {
                Eventi.this.dispose();
                System.out.println (evt);
            }
        });
    }
}
```

Questo modo di scrivere all'inizio può apparire criptico ma, una volta abituati, non crea problemi di comprensione e sicuramente diminuisce il lavoro di scrittura del programmatore.

16.3 Il metodo `paint`

Per comprendere i meccanismi necessari per aggiungere del testo o della grafica a un componente grafico in un ambiente a finestre, bisogna tenere conto del fatto che la visualizzazione di un oggetto di questo tipo è soggetta a essere modificata indipendentemente dalla volontà di chi ha scritto il programma. In generale, infatti, una finestra può essere nascosta da un'altra, anche appartenente a un altro programma, può essere ridotta a icona e può essere modificata nelle dimensioni. Esistono quindi diversi casi in cui il contenuto di una finestra deve essere ridisegnato. Questo compito non viene gestito automaticamente dal sistema ma è di responsabilità del programma per due motivi: il primo riguarda il fatto che se il sistema mantenesse le rappresentazioni grafiche, o *bitmap*, di tutte le finestre, verrebbero impegnate ingenti quantità di memoria, mentre il secondo deriva dal fatto che se una finestra viene modificata nelle dimensioni, è possibile che il programma debba visualizzarne il contenuto in modo diverso.

Ogni volta che un componente grafico deve essere disegnato o ridisegnato, viene inviato al programma un evento che provoca l'attivazione del metodo `public void paint(Graphics g)` dell'oggetto destinatario. Questo metodo è definito nella classe `Component` in quanto tutti gli oggetti grafici sono soggetti a essere ridisegnati. L'ar-

gomento di `paint` è un oggetto della classe astratta `Graphics` che rappresenta la connessione con il contesto grafico, che nel caso di `paint` corrisponde a un'area dello schermo ma che, in generale, può essere un qualsiasi dispositivo grafico di output o anche un'area di memoria. È grazie a oggetti di questo tipo che le applicazioni possono effettuare visualizzazioni grafiche senza preoccuparsi del tipo di dispositivo di output. Questa classe definisce un discreto numero di metodi per la visualizzazione di cui ne riportiamo solo alcuni con una breve descrizione.

- `public void drawLine(int x1, int y1, int x2, int y2)`: disegna una linea a partire dalle coordinate `x1, y1` fino alle coordinate `x2, y2`.
- `public void drawRect(int x, int y, int larghezza, int altezza)`: disegna un rettangolo che ha l'angolo in alto a sinistra sulle coordinate `x, y` e le dimensioni specificate da `larghezza` e `altezza`.
- `public void fillRect(int x, int y, int larghezza, int altezza)`: disegna un rettangolo pieno che ha l'angolo in alto a sinistra sulle coordinate `x, y` e le dimensioni specificate da `larghezza` e `altezza`.
- `public void drawOval(int x, int y, int larghezza, int altezza)`: disegna un ovale inscritto nel rettangolo che ha l'angolo in alto a sinistra sulle coordinate `x, y` e le dimensioni specificate da `larghezza` e `altezza`.
- `public void fillOval(int x, int y, int larghezza, int altezza)`: disegna un ovale pieno inscritto nel rettangolo che ha l'angolo in alto a sinistra sulle coordinate `x, y` e le dimensioni specificate da `larghezza` e `altezza`.
- `public void drawString(String str, int x, int y)`: scrive il contenuto della stringa `str` in modo tale che il rettangolo ideale che la racchiude abbia il suo angolo in basso a sinistra posizionato alle coordinate `x, y`.

Possiamo aggiungere allora il metodo `paint` alla precedente classe `Eventi` in modo da effettuare una semplice visualizzazione (vedi Figura 16.2).

```
public void paint (Graphics g) {
   g.drawString ("Il primo programma grafico", 5, 40);
}
```

Bisogna prestare attenzione che l'origine delle coordinate è sempre l'angolo in alto a sinistra esterno, per cui l'area occupata dai bordi della finestra è indirizzabile come coordinate, ma non utilizzabile in quanto già occupata. La classe `Container` contiene il metodo `public Insets getInsets()` che, invocato dopo che la finestra è resa visibile, permette di conoscere le dimensioni in pixel dei bordi. La classe `Insets`, di cui fa parte l'oggetto restituito da `getInsets`, contiene infatti le variabili pubbliche `int top`, `int bottom`, `int left` e `int right` che corrispondono rispettivamente al bordo in alto, al bordo in basso, al bordo di sinistra e al bordo di destra.

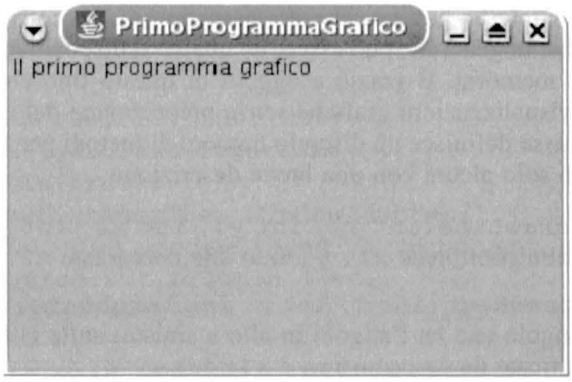


Figura 16.2 Visualizzazione di una stringa

16.4 La tastiera

Una finestra riceve costantemente gli eventi che la riguardano indipendentemente dal fatto che il programma debba trattarli o meno. La definizione di un listener definisce semplicemente un oggetto destinato al trattamento degli eventi e la sua assenza presuppone che gli eventi ricevuti siano trattati secondo modalità di default. Premendo quindi un qualsiasi tasto, viene generato un evento opportuno che viene inviato alla finestra attiva.

Un componente grafico che vuole leggere i tasti digitati dall'utente deve quindi aggiungere un listener che implementi l'interfaccia `KeyListener` i cui metodi sono i seguenti:

- `public void keyPressed(KeyEvent e)`: invocato quando viene premuto un tasto.
- `public void keyReleased(KeyEvent e)`: invocato quando viene rilasciato un tasto.
- `public void keyTyped(KeyEvent e)`: invocato dopo una pressione di un tasto cui corrisponde un carattere. La pressione dei tasti Shift o Ctrl, per esempio, non causa l'invocazione di questo metodo.

L'argomento di tali metodi è un oggetto di tipo `KeyEvent` che contiene una serie di costanti statiche e metodi che permettono il riconoscimento del tasto premuto, sia esso un carattere, un tasto funzionale o un tasto modificatore come Shift o Ctrl. Vediamo solo alcuni metodi della classe:

- `public char getKeyChar()`: restituisce il carattere digitato.
- `public int getKeyCode()`: restituisce un codice intero corrispondente al tasto premuto; una serie di costanti statiche permette di stabilire la corrispondenza tra codice e tasto premuto in modo indipendente, per quanto possibile, dalla piattaforma.

- `public boolean isActionKey()`: restituisce `true` se il tasto corrispondente all'evento corrisponde a un'azione, come per esempio un tasto funzionale o un tasto cursore, `false` altrimenti.

Nella classe `Component` è definito il metodo `public void addKeyListener (KeyListener l)` che permette di aggiungere un listener e il metodo `public void removeKeyListener (KeyListener l)` per rimuoverlo. Scriviamo dunque una semplice classe che visualizza i caratteri premuti sulla tastiera (Figura 16.3).

```
import java.awt.*;
import java.awt.event.*;

public class Tastiera extends Frame
                        implements KeyListener
{
   String testo = "";

   public static void main (String argv[]) {
      Tastiera ist = new Tastiera();
   }
   Tastiera () {
      setBounds (30, 10, 300, 200);
      setTitle (getClass().getName());
      addKeyListener (this);
      setVisible(true);
   }
   public void paint(Graphics g) {
      g.drawString (testo, 5, 40);
   }
   public void keyPressed(KeyEvent e) {
   }
   public void keyReleased(KeyEvent e) {
   }
   public void  keyTyped(KeyEvent e) {
      testo += e.getKeyChar();
      repaint();
   }
}
```

In questo esempio abbiamo implementato l'interfaccia `KeyListener` direttamente nell'oggetto derivato da `Frame` che è quindi diventato il listener di se stesso.

Nel metodo `keyTyped` è stato invocato il metodo `public void repaint()`, che fa parte della classe `Component`. Questa invocazione è necessaria per informare il sistema che il contenuto della finestra è stato modificato, per cui necessita di essere ridisegnata. Il metodo `repaint` causa l'invocazione del metodo `public void update(Graphics g)`, definito anch'esso nella classe `Component` che, se non viene ridefinito, ripulisce il contenuto della finestra e invoca il metodo `paint`.

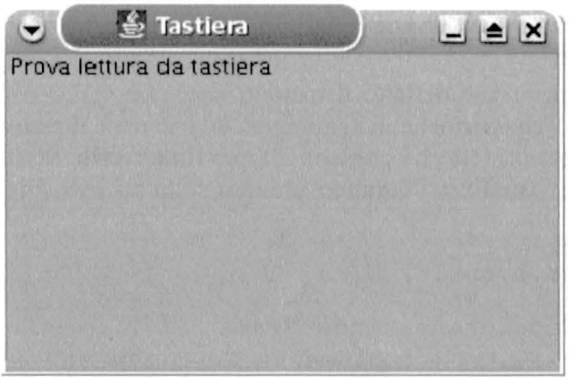


Figura 16.3 Visualizzazione dei caratteri premuti da tastiera

16.5 Il mouse

Quando il cursore del mouse si viene a trovare sull'area di una finestra, esso inizia a inviare una serie di eventi riguardanti la posizione, lo stato dei bottoni ecc. Per catturare tali eventi sono disponibili due interfacce che si differenziano per il tipo di eventi trattati. La prima è `MouseListener` e definisce i metodi:

- `public void mousePressed(MouseEvent e)`: invocato quando un bottone del mouse viene premuto.
- `public void mouseReleased(MouseEvent e)`: invocato quando un bottone del mouse viene rilasciato.
- `public void mouseClicked(MouseEvent e)`: invocato quando viene effettuato un click su un pulsante del mouse, cioè il pulsante viene premuto e rilasciato senza spostamenti del cursore.
- `public void mouseEntered(MouseEvent e)`: invocato quando il cursore del mouse entra nell'area del componente.
- `public void mouseExited(MouseEvent e)`: invocato quando il cursore del mouse esce dall'area del componente.

La seconda interfaccia è `MouseMotionListener`, serve per tracciare i movimenti del mouse e definisce i metodi:

- `public void mouseDragged(MouseEvent e)`: invocato ogni volta che viene rilevato uno spostamento del mouse con un pulsante premuto.
- `public void mouseMoved(MouseEvent e)`: invocato ogni volta che viene rilevato uno spostamento del mouse senza pulsanti premuti.

Entrambi i metodi di queste interfacce ricevono un parametro di tipo `MouseEvent` che fra gli altri mette a disposizione i metodi specifici:

- `public int getX()`: restituisce la coordinata orizzontale del cursore del mouse.
- `public int getY()`: restituisce la coordinata verticale del cursore del mouse.
- `public int getClickCount()`: restituisce il numero di click consecutivi.

Per aggiungere e rimuovere un listener per la gestione degli eventi del mouse a un componente, la classe `Component` mette a disposizione i seguenti metodi:

- `public synchronized void addMouseListener(MouseListener l)`: aggiunge il listener specificato come argomento al componente.
- `public synchronized void removeMouseListener(MouseListener l)`: rimuove il listener specificato come argomento dal componente.
- `public void addMouseMotionListener(MouseMotionListener l)`: aggiunge il listener specificato come argomento al componente.
- `public void removeMouseMotionListener(MouseMotionListener l)`: rimuove il listener specificato come argomento dal componente.

Osserviamo ora una semplice classe che permette di disegnare dei rettangoli premendo un bottone del mouse in un punto iniziale della finestra e spostando il cursore del mouse senza rilasciare il bottone finché non si giunge al punto finale desiderato (vedi Figura 16.4).

```
import java.awt.*;
import java.awt.event.*;

public class Mouse extends Frame {
   int x1 = -1, x2, y1, y2;
   public static void main (String argv[]) {
      Mouse ist = new Mouse();
   }
   Mouse () {
      setBounds (30, 10, 300, 200);
      setTitle (getClass().getName());
      addMouseListener (new MouseAdapter() {
         public void mousePressed(MouseEvent e) {
            x1 = e.getX();
            y1 = e.getY();
         }
         public void mouseReleased(MouseEvent e) {
            x1 = -1;
         }
      });
      addMouseMotionListener (new MouseMotionAdapter() {
         public void mouseDragged(MouseEvent e) {
```

```
                x2 = e.getX();
                y2 = e.getY();
                repaint();
            }
        });
        setVisible(true);
    }
    public void paint(Graphics g) {
        if (x1 > 0) {
            int x0, y0, larghezza, altezza;
            if (x1 < x2) {
                x0 = x1;
                larghezza = x2 - x1;
            } else {
                x0 = x2;
                larghezza = x1 - x2;
            }
            if (y1 < y2) {
                y0 = y1;
                altezza = y2 - y1;
            } else {
                y0 = y2;
                altezza = y1 - y2;
            }
            g.drawRect (x0, y0, larghezza, altezza);
        }
    }
}
```

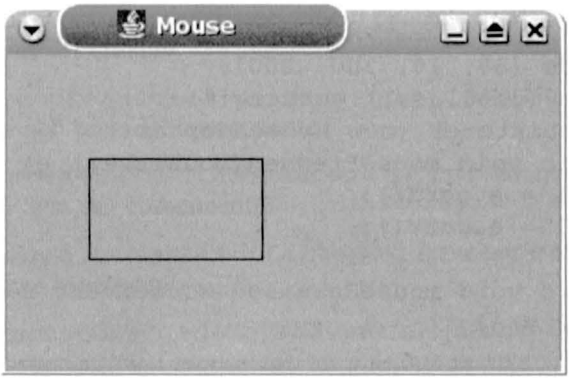


Figura 16.4 Disegno di un rettangolo con il mouse

In questo esempio, per implementare i listener, abbiamo usato due classi locali derivate dagli adattatori `MouseAdapter` e `MouseMotionAdapter`. Notare come dai metodi di tali classi sia possibile accedere a variabili e metodi della classe contenitrice, in questo caso `x1`, `y1`, `x2`, `y2` e `repaint()`, in modo assolutamente trasparente.

Notare che nel metodo `mouseDragged` è stato invocato il metodo `repaint` per causare il ridisegno della finestra. Abbiamo già detto che questo metodo fa in modo che venga richiamato il metodo `update` il quale per default ripulisce la finestra e invoca `paint`. Includendo quindi nella classe precedente il codice

```
public void update(Graphics g) {
   paint (g);
}
```

si evita che lo sfondo sia cancellato ogni volta e si possono ottenere immagini come quella in Figura 16.5.

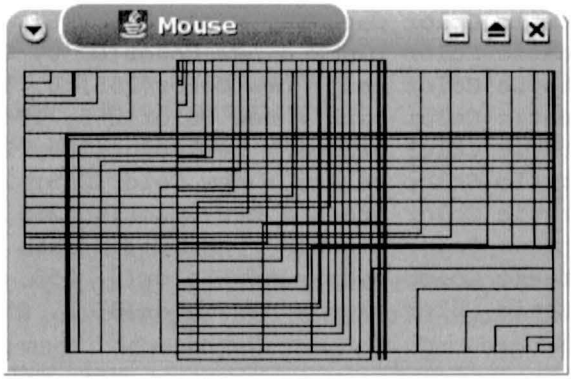


Figura 16.5 Disegno di più rettangoli con il mouse

16.6 I colori

I colori sono rappresentati nell'AWT tramite istanze della classe `Color` che deriva direttamente da `Object`. Secondo la rappresentazione data da questa classe, ogni colore è rappresentato da tre numeri compresi tra 0 e 255, ognuno dei quali indica la componente di rosso, verde e blu (RGB). Tra i costruttori disponibili per la classe troviamo i seguenti:

- `public Color(int rosso, int verde, int blu)`: crea un colore con le specificate componenti di `rosso`, `verde` e `blu`, che debbono essere numeri compresi tra 0 e 255.

- `public Color(int)`: crea un colore in cui il valore della componente di rosso è rappresentato dai bit 16-23, il valore della componente di verde dai bit 8-15 e il valore della componente di blu dai bit 0-7.
- `public Color(float rosso, float verde, float blu)`: crea un colore con le specificate componenti di `rosso`, `verde` e `blu`, che debbono essere numeri compresi tra 0.0 e 1.0 e corrispondono agli interi del primo costruttore divisi per 255.

In questo modo si può creare qualsiasi tonalità di colore anche se poi la resa grafica dipende dal tipo di terminale e scheda video utilizzata. L'AWT cercherà in ogni caso di approssimare quanto più possibile i colori definiti con quelli realmente disponibili.

Per semplificare il compito dei programmatori, nella classe sono definite alcune variabili statiche corrispondenti ai colori più comuni e precisamente:

```
public final static Color white = new Color(255, 255, 255);
public final static Color lightGray = new Color(192, 192, 192);
public final static Color gray = new Color(128, 128, 128);
public final static Color darkGray = new Color(64, 64, 64);
public final static Color black = new Color(0, 0, 0);
public final static Color red = new Color(255, 0, 0);
public final static Color pink = new Color(255, 175, 175);
public final static Color orange = new Color(255, 200, 0);
public final static Color yellow = new Color(255, 255, 0);
public final static Color green = new Color(0, 255, 0);
public final static Color magenta = new Color(255, 0, 255);
public final static Color cyan = new Color(0, 255, 255);
public final static Color blue = new Color(0, 0, 255);
```

✔ NOTA

Esistono nella classe `Color` *metodi che consentono di definire i colori per mezzo delle componenti tonalità, saturazione e luminosità (HSB) ed effettuare conversioni tra i due tipi di rappresentazione.*

Ogni istanza visibile della classe `Component` ha due colori associati, uno per lo sfondo (*background*) e uno per il primo piano (*foreground*). Questi colori possono essere ricuperati grazie rispettivamente ai metodi `public Color getBackground()` e `public Color getForeground()`, mentre per impostarne dei nuovi sono disponibili i metodi `public void setBackground(Color c)` e `public void setForeground(Color c)`. I colori di default sono tipicamente il bianco o il grigio chiaro per lo sfondo e il nero per il primo piano.

Un oggetto della classe `Graphics` ha sempre associato un *colore corrente* che viene utilizzato per tutte le visualizzazioni. Per default il colore corrente è uguale al colore di primo piano dell'oggetto di tipo `Component` associato. Per ricuperare il colore corrente la classe `Graphics` mette a disposizione il metodo `Color getColor()` mentre per impostarne uno nuovo il metodo da usare è `public void setColor(Color c)`.

Volendo quindi modificare l'esempio precedente in modo tale che lo sfondo della finestra sia di colore giallo e compaiano due scritte, di cui una rossa e una verde, possiamo usare la classe seguente (vedi Figura 16.6).

```
import java.awt.*;

public class Colori extends Frame
{
   public static void main (String argv[]) {
      Colori ist = new Colori();
   }
   Colori () {
  setBackground (Color.yellow);
  setForeground (Color.red);
      setBounds (30, 10, 300, 200);
      setTitle (getClass().getName());
      setVisible(true);
   }
   public void paint (Graphics g) {
      g.drawString ("Stringa rossa", 5, 40);
      g.setColor (Color.green);
      g.drawString ("Stringa verde", 5, 80);
   }
}
```

Va rilevato che, se sostituissimo l'invocazione `g.setColor (Color.green)` con `setForeground (Color.green)`, quest'ultima non avrebbe nessun effetto immediato, in quanto il colore corrente viene assegnato all'oggetto `Graphics` in fase di creazione. Solo causando il ridisegno delle stringhe si otterrebbe che entrambe vengano visualizzate col colore verde.

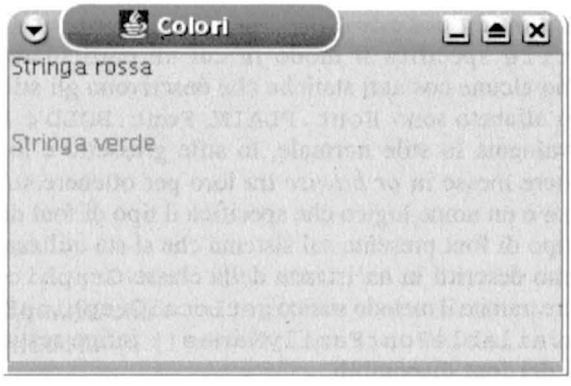


Figura 16.6 Scritte di vario colore su sfondo giallo

16.7 I font

Tutti gli ambienti a finestre ormai permettono l'utilizzo di *font* differenti per personalizzare i propri programmi. I font possono essere divisi in due grandi famiglie: a passo fisso e a passo variabile. I primi prevedono che ciascun carattere occupi sempre lo stesso spazio, come avveniva nelle macchine da scrivere manuali e sulle prime stampanti per computer, mentre i secondi prevedono che i caratteri occupino uno spazio orizzontale differente a seconda del carattere, come per esempio nel testo che state leggendo; si può notare dalla Tabella 16.1 infatti come una 'W' maiuscola occupi tre volte lo spazio orizzontale di una 'i' minuscola.

Tabella 16.1 Esempio di font a passo variabile e a passo fisso

Esempio di font a passo variabile	W. iii.	**Grassetto**	*corsivo*
`Esempio di font a passo fisso`	`WWW.` `iii.`	**`Grassetto`**	*`corsivo`*

Nell'AWT i font sono rappresentati da istanze della classe `Font` che deriva direttamente da `Object`. Uno dei due costruttori è il seguente del quale andiamo a esaminare in dettaglio i parametri partendo dall'ultimo:

```
public Font(String nome, int stile, int dim)
```

Il parametro `dim` specifica la dimensione verticale del font ed è espressa in "punti". Questa dimensione non è legata al tipo di dispositivo di output, ma è una misura che rappresenta in qualche modo la dimensione reale dei caratteri una volta visualizzati sullo schermo o su stampante. Un "punto" corrisponde circa a $^1/_{72}$ di pollice e, per fare un esempio, il font con cui è scritto questo testo ha dimensione 10,5 punti. La dimensione verticale influenza anche la dimensione orizzontale dei caratteri e in ogni caso essi saranno sempre proporzionati, vale a dire che il rapporto tra altezza e larghezza non cambia.

Il parametro `stile` specifica il modo in cui un particolare font può essere visualizzato. Esistono alcune costanti statiche che descrivono gli stili possibili: quelle riguardanti il nostro alfabeto sono `Font.PLAIN`, `Font.BOLD` e `Font.ITALIC` e specificano rispettivamente lo stile normale, lo stile grassetto e lo stile corsivo. Le costanti possono essere messe in *or bitwise* tra loro per ottenere stili composti.

Il parametro `nome` è un nome logico che specifica il tipo di font da utilizzare e deve corrispondere a un tipo di font presente sul sistema che si sta utilizzando. I font disponibili sul sistema sono descritti in un'istanza della classe `GraphicsEnvironment` che è possibile ottenere tramite il metodo statico `getLocalGraphicsEnvironment()`.

Il metodo `getAvailableFontFamilyNames()` infine restituisce un array di stringhe con i nomi dei font disponibili.

La visualizzazione dell'elenco dei font disponibili sul proprio sistema si può ottenere tramite la seguente classe:

```
import java.awt.*;

public class ListaFont {
   public static void main (String argv[]) {
      String listaFont[] = GraphicsEnvironment
                           .getLocalGraphicsEnvironment()
                           .getAvailableFontFamilyNames();
      for (int i = 0; i < listaFont.length; i++)
         System.out.println (listaFont[i]);
   }
}
```

Un esempio di visualizzazione possibile eseguito dalla classe è

```
Andale Mono
Arial
Arial Black
Arioso
Bitstream Charter
Century Schoolbook L
Chevara
ChevaraOutline
Comic Sans MS
Conga
Courier
Courier 10 Pitch
Courier New
Cursor
Dialog
DialogInput
Dingbats
Georgia
Helmet
HelmetCondensed
Hershey
Impact
Kochi Gothic
Kochi Mincho
Lucida Bright
Lucida Console
Lucida Sans
Lucida Sans Typewriter
Lucida Sans Unicode
Luxi Mono
Luxi Sans
Luxi Serif
```

```
Microsoft Sans Serif
Monospaced
Nimbus Mono L
Nimbus Roman No9 L
Nimbus Sans L
SansSerif
Serif
Standard Symbols L
StarBats
StarMath
Symbol
Tahoma
Times New Roman
Timmons
Trebuchet MS
URW Bookman L
URW Chancery L
URW Gothic L
URW Palladio L
Utopia
Verdana
Webdings
Wingdings
```

I font disponibili possono differire tra computer diversi, ma si può affermare che ne esiste sempre uno a passo fisso, chiamato `Monospaced`, e uno a passo variabile, chiamato `Dialog`.

Ogni istanza visibile della classe `Component` ha associato un oggetto della classe `Font` che per default è normalmente di tipo `Dialog`. Questo font può essere ricuperato per mezzo del metodo `public Font getFont()`, mentre per impostarne uno è necessario usare il metodo `public void setFont(Font f)`.

Anche la classe `Graphics` ha sempre associato un font che viene utilizzato per la visualizzazione di testi e che, per default, è uguale a quello dell'oggetto di tipo `Component` associato. Due metodi del tutto analoghi a quelli appena visti `getFont` e `setFont` permettono di ricuperare il font corrente e di impostarne uno nuovo. La classe successiva esegue la visualizzazione di due stringhe utilizzando due font diversi (vedi Figura 16.7).

```
import java.awt.*;

public class FontTipografici extends Frame {
    public static void main (String argv[]) {
      FontTipografici ist = new FontTipografici();
   }
   FontTipografici () {
  setFont (new Font ("Monospaced", Font.PLAIN, 12));
```

```
        setBounds (30, 10, 300, 200);
        setTitle (getClass().getName());
        setVisible(true);
    }
    public void paint (Graphics g) {
        g.drawString ("Stringa Monospaced", 5, 40);
   g.setFont (new Font ("Dialog", Font.PLAIN, 12));
        g.drawString ("Stringa Dialog", 5, 80);
    }
}
```

Anche in questo caso, se sostituissimo l'invocazione `g.setFont(new Font ("Dialog", Font.PLAIN,12))` con `setFont(new Font ("Dialog", Font.PLAIN,12))`, quest'ultima non avrebbe nessun effetto immediato, in quanto il font viene assegnato all'oggetto `Graphics` in fase di creazione. Solo causando il ridisegno delle stringhe si otterrebbe che entrambe vengano visualizzate col font `Dialog`.

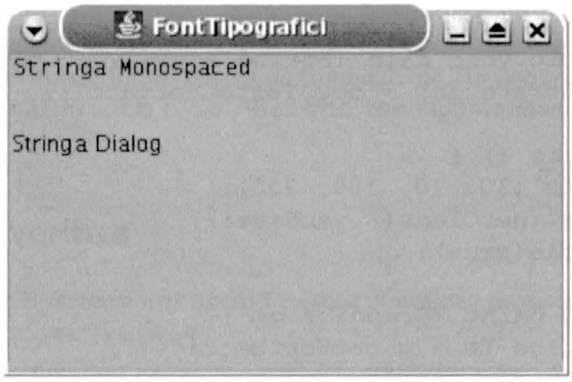


Figura 16.7 Font diversi

Risulta evidente che posizionare e allineare testi utilizzando i pixel come coordinate e trattando con caratteri di dimensioni variabili non è un compito molto agevole. Per aiutarci in questo compito, possiamo avvalerci degli oggetti della classe `FontMetrics`.

Questa classe astratta, che deriva direttamente da `Object`, rappresenta le dimensioni di un font e fornisce una serie di metodi che consentono di posizionare e allineare dei testi all'interno di un componente grafico. Una sua istanza concreta, dipendente dall'implementazione corrente dell'AWT, può essere ottenuta da un oggetto della classe `Graphics` per mezzo dei metodi `public FontMetrics getFontMetrics()`, che restituisce un'istanza corrispondente al font corrente, e `public FontMetrics getFontMetrics(Font f)`, che restituisce un'istanza corrispondente al font spe-

cificato come argomento. Anche la classe `Component` dispone di un metodo `public FontMetrics getFontMetrics(Font f)` che opera come quello appena visto. Di tale classe, che definisce diversi metodi, ne citeremo tre:

- `public int getHeight()`: restituisce l'altezza del font in pixel.
- `public int getMaxAdvance()`: restituisce la massima larghezza in pixel di un carattere.
- `public int stringWidth(String str)`: restituisce la larghezza totale in pixel della stringa passata come argomento.

A questo punto vediamo un esempio di una classe che, dato un array di stringhe, ne esegue una visualizzazione in modo tale che risulti sempre al centro della finestra (Figura 16.8).

```
import java.awt.*;

public class TestoCentrato extends Frame {
   String testo[] = { "Questo",
                      "testo",
                      "è",
                      "centrato" };
   public static void main (String argv[]) {
      TestoCentrato ist = new TestoCentrato();
   }
   TestoCentrato () {
      setBounds (30, 10, 300, 200);
      setTitle (getClass().getName());
      setVisible(true);
   }
   public void paint (Graphics g) {
      FontMetrics fm = g.getFontMetrics();
      Dimension d = getSize();
      Insets ins = getInsets();
      int larghezza = d.width - ins.left - ins.right;
      int altezza = d.height - ins.top - ins.bottom;
      int altFont = fm.getHeight();
      int totAlt = altFont * testo.length;
      int y0 = ins.top + altFont + (altezza - totAlt) / 2;
      for (int i = 0; i < testo.length; i++) {
         int x0 = ins.left + (larghezza -
                              fm.stringWidth(testo[i])) / 2;
         g.drawString (testo[i], x0, y0 + i * altFont);
      }
   }
}
```

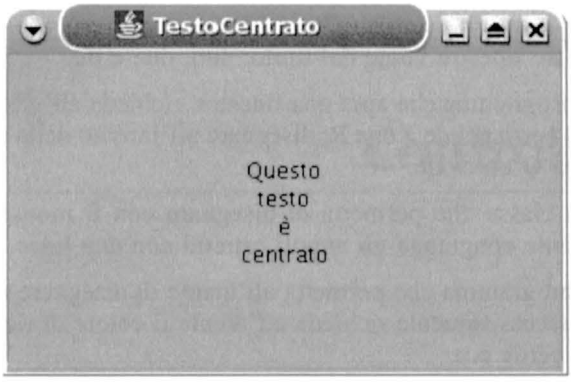


Figura 16.8 Visualizzazione del testo centrato nella finestra

Il testo da visualizzare è contenuto nell'array `testo`, che è una variabile d'istanza della classe. Il metodo `getSize()`, appartenente alla classe `Component` e di cui non è stata fornita ancora nessuna descrizione, restituisce un oggetto della classe `Dimension` contenente le dimensioni in pixel del componente grafico corrente su video. Quest'ultima classe infatti deriva direttamente da `Object` e contiene due variabili pubbliche, `int width` e `int height` che rappresentano una larghezza e un'altezza.

Domande di verifica

1. A cosa serve il metodo `setBound` e come si utilizza precisamente? E i metodi `setTitle` e `setVisible`?
2. Descrivere le funzioni della classe `WindowEvent` e dei suoi metodi.
3. Quali sono i metodi riportati nel testo della classe `Graphics`? E come si utilizzano?
4. Quali metodi sono disponibili per leggere i dati digitati dall'utente? Di quale interfaccia fanno parte? Quali metodi permettono rispettivamente di restituire il carattere digitato e il codice intero corrispondente al carattere premuto?
5. Come agisce ognuno dei metodi disponibili in `MouseListener` e `MouseMotionListener`?
6. Perché si usa la classe `Color` e quali costruttori possiede? Come si creano le tonalità dei colori?
7. Quale significato hanno i parametri del costruttore `Font` presentato nel testo?
8. Perché si usano i metodi `getFont` e `setFont`?
9. Descrivere la classe `FontMetrics` e i suoi metodi.

Esercizi

1. Visualizzare tre finestre vuote dal titolo: uno, due e tre.
2. Scrivere un programma che apra una finestra, richieda all'utente di premere un tasto, se esso corrisponde a una R, disegnare all'interno della finestra un rettangolo, se a una O un ovale.
3. Definire una classe che permetta di disegnare con il mouse un rettangolo e successivamente congiunga gli angoli estremi con due linee.
4. Scrivere un programma che permetta all'utente di disegnare un rettangolo con il mouse e successivamente richieda all'utente il colore di riempimento: R per rosso, V per verde ecc.
5. Realizzare un programma che consenta all'utente di scrivere un testo nella finestra, permettendogli di decidere a priori il font (Alt-M per Monospaced, Alt-T per Times ecc.) e il colore (Ctrl-R per rosso, Ctrl-V per verde ecc.).
6. Realizzare un programma che disegni all'interno di una finestra una tabella 3×3, formata da un totale di 9 caselle grigie, e permetta di giocare secondo le regole seguenti:

   - Un giocatore ha a disposizione pedine nere, l'altro bianche.
   - Ogni giocatore effettua una mossa alla volta posizionando una sua pedina su una casella a scelta tra quelle libere.
   - Vince chi posiziona in fila (verticale, orizzontale o obliqua) tre proprie pedine.

   Variazioni:

   - Far sì che uno dei due giocatori sia il computer.
   - Far sì che l'utente possa scegliere se giocare per primo o lasciare che lo faccia il computer.
   - Far scegliere all'utente il tipo di segnale grafico da utilizzare sulla tabella al posto delle pedine (esempio X contro O).

Capitolo 17

Controlli grafici

Obiettivi didattici

- Bottoni
- Inserimento e formattazione di testi
- Etichette
- *Checkbox* singole e mutuamente esclusive
- Liste a selezione singola e multipla
- Barre di scorrimento (*scrollbar*)
- Menu
- Impaginazione secondo diverse strategie
- Suddivisione della finestra principale in aree logiche

Alcune classi derivate dalla classe `Component` rappresentano degli oggetti grafici concepiti per essere contenuti negli oggetti delle classi derivate da `Container`, come per esempio `Frame`, e destinati a ricevere le azioni degli utenti. Tali oggetti, ai quali a volte ci si riferisce col nome di *controlli* (control), sono in pratica delle sottofinestre destinate a ricevere un determinato tipo di input e fornite dei metodi necessari per effettuare la propria rappresentazione grafica e per raccogliere gli eventi ed elaborarli. I controlli, per essere visualizzati, debbono essere *aggiunti* tramite il metodo `add` a un oggetto di una classe derivata da `Container` a sua volta visibile.

Mostriamo di seguito alcune classi derivate direttamente da `Component` che implementano le funzionalità dei controlli.

✔ **NOTA**

Nell'AWT il concetto di controllo non compare dalla gerarchia delle classi e il raggruppamento fatto in questo capitolo è dovuto ad analogie funzionali evidenziate in altri ambienti grafici. Di fatto le classi che qui definiamo controlli sono semplicemente le classi concrete derivate direttamente da `Component`.

17.1 Bottoni

Uno dei controlli più semplici è rappresentato dalla classe `Button` che disegna su video un *bottone da premere* (*push button*), cioè un'area rettangolare, tipicamente contenente una stringa descrittiva, che accetta un click del mouse, dando in tal caso l'impressione visiva di essere premuto. Due sono i costruttori disponibili:

- `public Button()`: costruisce un oggetto di tipo bottone senza descrizioni.
- `public Button(String etichetta)`: costruisce un oggetto di tipo bottone al cui centro viene visualizzata la stringa `etichetta`.

Abbiamo detto che un oggetto di questo tipo è in pratica una finestra, per cui, in effetti, riceve degli eventi del mouse in modo analogo a quanto visto precedentemente per un oggetto della classe `Frame`. In questo caso però non è importante conoscere esattamente il tipo e la modalità degli eventi giunti al componente, ma semplicemente se esso ha ricevuto una sequenza di eventi tali da poter essere interpretata logicamente come la pressione del bottone. Per questo motivo gli oggetti di questo tipo accettano listener che implementano l'interfaccia `ActionListener` la quale definisce il solo metodo `public void actionPerformed(ActionEvent e)` che viene invocato alla pressione del bottone. L'argomento del metodo è un oggetto di tipo `ActionEvent` che definisce tra gli altri il metodo `public String getActionCommand()` che restituisce la stringa di comando associata all'evento; per gli oggetti della classe `Button` tale stringa può essere impostata per mezzo del metodo `public void setActionCommand(String command)` e ricuperata tramite `public String getActionCommand()`, mentre per default ha il valore dell'etichetta impostata nel costruttore.

Quando un evento non corrisponde direttamente a un'azione fisica dell'utente ma a un'azione logica, derivata da uno o più eventi come in questo caso, si parla di *evento semantico*.

Scriviamo ora una classe che genera una finestra che visualizza la data e l'ora e che contiene due bottoni, premendo il primo dei quali si ottiene l'aggiornamento della data e ora, mentre premendo il secondo si causa la terminazione del programma (vedi Figura 17.1).

```
import java.util.Date;
import java.awt.*;
import java.awt.event.*;
```

```
public class Bottoni extends Frame {
   Date data = new Date();
   public static void main (String argv[]) {
      Bottoni ist = new Bottoni();
   }
   Bottoni () {
      setLayout(null);
      setBounds (30, 10, 300, 200);
      setTitle (getClass().getName());
      Button bData = new Button("Data");
      bData.setBounds (50, 150, 50, 30);
      bData.addActionListener (new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            data = new Date();
            repaint();
         }
      });
      Button bEsci = new Button("Esci");
      bEsci.setBounds (200, 150, 50, 30);
      bEsci.addActionListener (new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            System.exit(0);
         }
      });
      add (bData);
      add (bEsci);
      setVisible(true);
   }
   public void paint (Graphics g) {
      g.drawString (data.toString(), 5, 40);
   }
}
```

In questo esempio, nei metodi `actionPerformed` non è stato necessario controllare il valore passato come argomento in quanto è stato definito un listener diverso per ciascun bottone. Se avessimo definito un unico listener per entrambi i bottoni, nell'unico metodo `actionPerformed` definito avremmo dovuto usare `getActionCommand()` per riconoscere il bottone premuto. Mostreremo questo secondo approccio nell'esempio successivo.

Si può notare come un oggetto di tipo `Button` venga trattato con metodi analoghi a quelli utilizzati per l'oggetto di tipo `Frame`. Va sottolineato però che le coordinate specificate nel metodo `setBound` del bottone sono relative alla finestra contenitrice e non all'intero schermo.

Figura 17.1 Controlli push button

In questo esempio abbiamo scelto di posizionare ogni controllo nella finestra specificando esattamente la posizione e la dimensione di ciascun elemento in pixel. Questo tipo di approccio presenta due inconvenienti: il primo è che disegnare finestre in questo modo può risultare lungo e noioso, mentre il secondo è che se l'utente ridimensiona la finestra col mouse, i componenti rimangono ancorati alle loro posizioni e dimensioni, dando luogo a una *impaginazione* (layout) inadeguata (Figura 17.2).

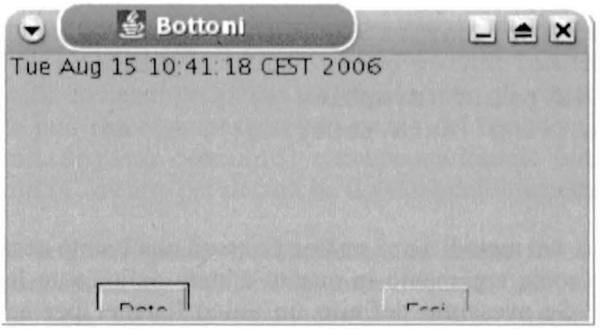


Figura 17.2 Inconvenienti derivanti dalla specifica della posizione e dimensione

Per risolvere questo tipo di problemi, la classe `Container` prevede la possibilità di avere associato un *impaginatore*, un oggetto cioè che implementa l'interfaccia `LayoutManager` e si occupa di disporre i controlli nella finestra in modo opportuno, tenendo in debito conto le dimensioni correnti di quest'ultima.

L'AWT mette a disposizione già alcune classi che implementano `LayoutManager` e che si differenziano tra loro per il tipo di strategia adottata per l'impaginazione; un oggetto di una di queste classi, `BorderLayout`, è associato per default a ogni nuovo oggetto appartenente a una classe che deriva da `Frame`. Questo comporta che quando si desidera effettuare un'impaginazione manuale, è necessario annullare l'impaginatore di default tramite l'invocazione del metodo `setLayout(null)`, come appare sulla prima riga del costruttore nel nostro esempio.

Anche negli esempi seguenti relativi a controlli, non abbiamo ritenuto opportuno usare impaginatori, rimandando l'illustrazione di queste classi alla fine del capitolo.

✔ **NOTA**

Negli ambienti operativi dove i controlli sono implementati a livello di sistema, l'AWT tende ad usarli direttamente sia per ottenere prestazioni migliori, sia per fare in modo che i programmi mantengano il look & feel del sistema ospite in modo da non risultare disomogenei rispetto alle applicazioni che l'utente è abituato a usare. Questo approccio d'altro canto comporta che alcuni utilizzi al limite della definizione diano risultati inaspettati e differenti da sistema operativo a sistema operativo. Nel caso si debbano sviluppare applicazioni multipiattaforma, raccomandiamo quindi di utilizzare solo funzionalità il cui comportamento è documentato in modo esplicito.

17.2 Input di testi

La classe `TextComponent` rappresenta un controllo che permette l'inserimento di testi e fornisce anche delle rudimentali funzioni di formattazione. Da essa derivano due classi, `TextField` e `TextArea`, che sono specializzate rispettivamente per l'introduzione di una linea di testo e l'introduzione di testi di più righe.

Su oggetti di questa classe tipicamente non è necessario aggiungere dei listener, ma è comunque possibile farlo. In particolare si può aggiungere un listener di tipo `ActionListener` che richiama il metodo `actionPerformed` ogni volta che si preme il tasto 'Invio'. Alcuni dei metodi più utilizzati sono riportati di seguito.

- `public void setText(String t)`: assegna il testo della stringa `t` al componente.
- `public String getText()`: restituisce una stringa col contenuto del componente.
- `public boolean isEditable()`: restituisce `true` se il componente è modificabile, `false` altrimenti.
- `public void setEditable(boolean b)`: rende il componente modificabile o meno a seconda che il parametro `b` valga rispettivamente `true` o `false`.

Dato che questa classe deriva da `Component`, il font e i colori possono essere alterati con i metodi già visti; per default vengono tipicamente utilizzati quelli definiti nella finestra contenitrice.

La classe seguente crea una finestra con due oggetti `TextField` e due bottoni e fa in modo che la pressione del primo bottone causi lo scambio del contenuto tra i due `TextField`, mentre la pressione del secondo causi l'uscita dal programma (vedi Figura 17.3).

```
import java.awt.*;
import java.awt.event.*;

public class LineaDiTesto extends Frame
                          implements ActionListener {
   TextField t1;
   TextField t2;
   public static void main (String argv[]) {
      LineaDiTesto ist = new LineaDiTesto();
   }
   LineaDiTesto () {
      setLayout(null);
      setBounds (30, 10, 300, 200);
      setTitle (getClass().getName());
      Button bData = new Button("Scambia");
      bData.setBounds (50, 150, 60, 30);
      bData.addActionListener (this);
      Button bEsci = new Button("Esci");
      bEsci.setBounds (200, 150, 60, 30);
      bEsci.addActionListener (this);
      t1 = new TextField();
      t1.setBounds (50, 50, 50, 20);
      t2 = new TextField();
      t2.setBounds (200, 50, 50, 20);
      add (t1);
      add (t2);
      add (bData);
      add (bEsci);
      setVisible(true);
   }
   public void actionPerformed(ActionEvent e) {
      if (e.getActionCommand().equals("Esci"))
         System.exit(0);
      if (e.getActionCommand().equals("Scambia")) {
         String scambio = t1.getText();
         t1.setText(t2.getText());
         t2.setText(scambio);
      }
   }
}
```

Figura 17.3 Oggetti `TextField` e push button

Si può notare come in questo esempio non sia più necessario definire il metodo `paint` in quanto i controlli sono autosufficienti per quanto riguarda la propria visualizzazione.

17.3 Etichette

Abbiamo visto come sia possibile scrivere dei testi costanti in una finestra tramite il metodo `paint`. In alcune situazioni però risulta più comodo poter visualizzare stringhe costanti utilizzando degli oggetti che, come `TextComponent` e classi derivate, gestiscano autonomamente la propria visualizzazione. Questo consente, da un lato di poter gestire in modo omogeneo e in un unico punto della classe tutto il codice riguardante la visualizzazione di una finestra, risparmiando di definire `paint` ed eventuali invocazioni di `repaint`, e dall'altro di poter utilizzare gli impaginatori automatici per ricercare il layout migliore in base agli oggetti su video e alle dimensioni della finestra, come vedremo in seguito.

La classe `Label` (etichetta) permette la visualizzazione di una singola linea di testo entro un'area specifica della finestra e definisce i costruttori:

- `public Label()`: crea un'etichetta vuota.
- `public Label(String testo)`: crea un'etichetta contenente la stringa `testo`.
- `public Label(String testo, int allineamento)`: crea un'etichetta contenente la stringa `testo` e l'allineamento specificato nel parametro `allineamento` che può contenere i valori `Label.LEFT`, `Label.RIGHT` e `Label.CENTER` rappresentanti rispettivamente l'allineamento a sinistra, che è quello di default, l'allineamento a destra e l'allineamento al centro.

Un oggetto di questa classe non deve trattare eventi e definisce, fra gli altri, i seguenti metodi:

- `public int getAlignment()`: permette di ricuperare il tipo di allineamento.

- `public void setAlignment(int allineamento)`: permette di impostare il tipo di allineamento.
- `public String getText()`: permette di ricuperare il testo associato all'etichetta; `public void setText(String testo)`: permette di impostare il testo associato all'etichetta.

Vediamo ora come sia possibile scrivere una classe che visualizza due stringhe di colore diverso e con font diversi senza usare il metodo `paint` (vedi Figura 17.4).

```
import java.awt.*;
public class Etichette extends Frame {
   public static void main (String argv[]) {
      Etichette ist = new Etichette();
   }
   Etichette () {
      setLayout (null);
  setBackground (Color.yellow);
  setForeground (Color.red);
      setFont (new Font("Monospaced", Font.PLAIN, 12));
      setBounds (30, 10, 300, 200);
      setTitle (getClass().getName());
      Label lRC = new Label ("Stringa rossa Monospaced");
      lRC.setBounds (5, 40, 200, 30);
      Label lVD = new Label ("Stringa verde Dialog");
      lVD.setBounds (5, 80, 200, 30);
      lVD.setFont (new Font("Dialog", Font.PLAIN, 12));
      lVD.setForeground (Color.green);
      add(lRC);
      add(lVD);
      setVisible(true);
   }
}
```

Figura 17.4 Colori e font diversi ottenuti senza il metodo `paint`

17.4 La classe Checkbox

La classe Checkbox, termine che in italiano si può tradurre con *casella di controllo*, serve per permettere agli utenti l'inserimento di valori booleani ed emula le caselle utilizzate nei questionari che debbono essere barrate per indicare la selezione di una determinata risposta. Un oggetto di questa classe infatti è composto da una casella, tipicamente quadrata, e da una etichetta associata. Utilizzando il mouse è possibile selezionare o deselezionare questi oggetti che rendono visibile il loro stato tramite la presenza o l'assenza di una crocetta o di un segno di spunta nella casella. Vediamo alcuni costruttori.

- `public Checkbox()`: costruisce un oggetto della classe con etichetta vuota inizialmente non selezionato.
- `public Checkbox(String etichetta)`: costruisce un oggetto della classe con etichetta corrispondente a quella specificata come argomento, inizialmente non selezionato.
- `public Checkbox(String etichetta, boolean stato)`: costruisce un oggetto della classe con etichetta corrispondente a quella specificata come argomento. Se `stato` vale `true`, l'oggetto è selezionato per default, altrimenti è non selezionato.

I metodi più utili della classe sono:

- `public String getLabel()`: restituisce l'etichetta associata all'oggetto.
- `public void setLabel(String etichetta)`: imposta l'etichetta associata all'oggetto.
- `public boolean getState()`: restituisce `true` se l'oggetto è selezionato, `false` altrimenti.
- `public void setState(boolean stato)`: seleziona l'oggetto se `stato` vale `true`, lo deseleziona altrimenti.
- `public void addItemListener(ItemListener l)`: aggiunge il listener specificato per la gestione degli eventi.
- `public void removeItemListener(ItemListener l)`: rimuove il listener specificato.

In generale non è necessario gestire gli eventi degli oggetti di questa classe in quanto nella maggioranza dei casi è sufficiente conoscerne semplicemente lo stato. Nel caso in cui sia necessario invece reagire immediatamente alla modifica di stato di uno di questi oggetti, è possibile specificare un listener che implementi l'interfaccia `ItemListener` che comprende il solo metodo `public void itemStateChanged(ItemEvent e)` invocato ogni volta che avviene una modifica di stato. La classe `ItemEvent` definisce tra gli altri il metodo `public Object getItem()` che restituisce un oggetto legato a quello che ha subito la modifica di stato, nel nostro caso una stringa contenente l'etichetta.

La classe seguente visualizza una finestra con due oggetti della classe `CheckBox` e due oggetti della classe `Label` che ne mostrano lo stato in tempo reale (Figura 17.5).

```
import java.awt.*;
import java.awt.event.*;

public class CasellaControllo extends Frame {
   Checkbox cTest1;
   Checkbox cTest2;
   Label lT1;
   Label lT2;
   public static void main (String argv[]) {
      CasellaControllo ist = new CasellaControllo();
   }
   CasellaControllo() {
      setLayout(null);
      setBounds (30, 10, 300, 200);
      setTitle (getClass().getName());
      cTest1 = new Checkbox("Test1");
      cTest1.setBounds (50, 50, 50, 20);
      cTest1.addItemListener (new ItemListener() {
         public void itemStateChanged(ItemEvent e) {
            vediStato (cTest1, lT1);
         }
      });
      cTest2 = new Checkbox("Test2");
      cTest2.setBounds (200, 50, 50, 20);
      cTest2.addItemListener (new ItemListener() {
         public void itemStateChanged(ItemEvent e) {
            vediStato (cTest2, lT2);
         }
      });
      lT1 = new Label ();
      lT1.setBounds (50, 100, 250, 20);
      lT2 = new Label ();
      lT2.setBounds (50, 120, 250, 20);
      vediStato (cTest1, lT1);
      vediStato (cTest2, lT2);
      add(cTest1);
      add(cTest2);
      add(lT1);
      add(lT2);
      setVisible(true);
   }
```

```
    void vediStato (Checkbox c, Label l) {
        l.setText("nome:" + c.getLabel() +
                  ", stato:" + (c.getState() ? "" : "NON ") +
                  "SELEZIONATO");
    }
}
```

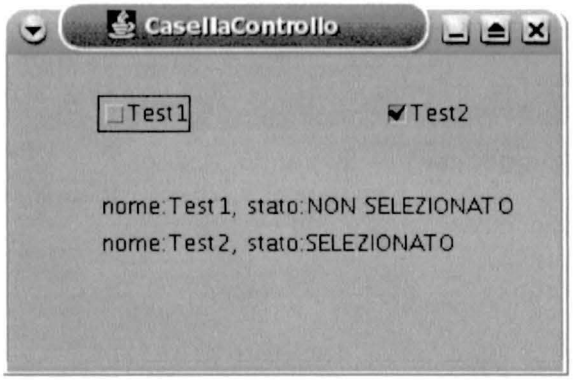


Figura 17.5 Oggetti `CheckBox` e `Label`

17.5 La classe `CheckboxGroup`

Ci sono occasioni in cui torna comodo avere delle caselle simili a quelle appena viste che permettano di scegliere una sola opzione fra un gruppo di possibilità mutuamente esclusive, come, per esempio, selezionare maschio/femmina, italiano/CEE/extra-CEE, ecc. La classe `CheckboxGroup` serve per raggruppare un numero qualsiasi di oggetti della classe `Checkbox` e fare in modo che di questi al massimo uno per volta risulti selezionato. Essa ha un solo costruttore senza argomenti e i seguenti metodi:

- `public Checkbox getSelectedCheckbox()`: restituisce la check box selezionata del gruppo. Se nessuna check box è selezionata, restituisce `null`.
- `public void setSelectedCheckbox(Checkbox box)`: imposta la check box passata per argomento come selezionata. Se la check box non appartiene al gruppo, questo metodo non fa niente.

Per fare in modo che un oggetto della classe `Checkbox` faccia parte di un gruppo, esso deve essere creato tramite il costruttore

- `public Checkbox(String s, boolean st, CheckboxGroup grp)`: crea un oggetto con etichetta `s`, il cui stato di selezione corrisponde a `st` e appartenente al gruppo `grp`.

Le caselle degli oggetti della classe `Checkbox` appartenenti a un gruppo assumono un aspetto visivo differente, tipicamente rotondo o a forma di rombo, rispetto a quelle di oggetti non appartenenti a gruppi, per informare l'utente del diverso tipo di comportamento.

La classe `CheckboxGroup` non gestisce eventi, ma è possibile usare quelli ricevuti dagli oggetti della classe `Checkbox` nei casi in cui sia necessario reagire immediatamente a una modifica di selezione.

Modifichiamo la classe vista precedentemente nel paragrafo sulle check box in modo tale che le selezioni diventino mutuamente esclusive e che un oggetto della classe `Label` visualizzi in tempo reale la casella selezionata

```
import java.awt.*;
import java.awt.event.*;

public class GruppoCaselle extends Frame
                        implements ItemListener {
   CheckboxGroup gTest;
   Checkbox cTest1;
   Checkbox cTest2;
   Label lT;
   public static void main (String argv[]) {
      GruppoCaselle ist = new GruppoCaselle();
   }
   GruppoCaselle() {
      setLayout(null);
      setBounds (30, 10, 300, 200);
      setTitle (getClass().getName());
      gTest = new CheckboxGroup ();
      cTest1 = new Checkbox("Test1", false, gTest);
      cTest1.setBounds (50, 50, 50, 20);
      cTest1.addItemListener (this);
      cTest2 = new Checkbox("Test2", false, gTest);
      cTest2.setBounds (200, 50, 50, 20);
      cTest2.addItemListener (this);
      lT = new Label ();
      lT.setBounds (50, 100, 250, 20);
      vediStato (gTest.getSelectedCheckbox());
      add(cTest1);
      add(cTest2);
      add(lT);
      setVisible(true);
   }
   void vediStato (Checkbox c) {
      if (c == null)
         lT.setText("Nessuna Selezione");
      else
```

```
            lT.setText("Selezione=" + c.getLabel());
        }
        public void itemStateChanged(ItemEvent e) {
            vediStato (gTest.getSelectedCheckbox());
        }
}
```

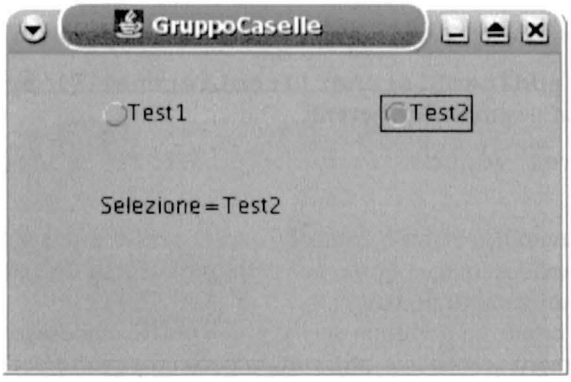


Figura 17.6 Selezioni mutuamente esclusive

17.6 La classe `Choice`

Nei casi in cui sia necessario che l'utente effettui una selezione tra un gruppo di possibilità mutuamente esclusive ma si desideri risparmiare spazio nella finestra oppure si desideri non modificare l'aspetto della finestra al variare del numero delle possibilità, si può utilizzare un oggetto della classe `Choice` che permette di avere associato un numero qualsiasi di stringhe, ciascuna delle quali è associata a un'opzione. Un oggetto di questa classe viene visualizzato in modo simile a un oggetto della classe `TextField` non modificabile con un piccolo bottone sulla destra, premendo il quale appare una finestra di tipo pop-up che consente di selezionare una qualsiasi delle stringhe contenute.

La classe definisce un unico costruttore senza argomenti e dispone tra gli altri dei metodi:

- `public void add(String str)`: aggiunge una stringa all'oggetto.
- `public int getItemCount()`: restituisce il numero di stringhe contenute nell'oggetto.
- `public String getItem(int pos)`: restituisce la stringa in posizione `pos`.
- `public void remove(String str)`: rimuove la prima stringa uguale `str`.

- `public void remove(int pos)`: rimuove la stringa in posizione `pos`.
- `public void removeAll()`: rimuove tutte le stringhe.
- `public String getSelectedItem()`: restituisce la stringa correntemente selezionata o `null` se nessuna stringa è selezionata.
- `public int getSelectedIndex()`: restituisce l'indice della stringa correntemente selezionata o `-1` se nessuna stringa è selezionata.
- `public void select(int pos)`: seleziona la stringa in posizione `pos`.
- `public void select(String str)`: seleziona la prima stringa uguale `str`.
- `public void addItemListener(ItemListener l)`: aggiunge il listener specificato per la gestione degli eventi.
- `public void removeItemListener(ItemListener l)`: rimuove il listener specificato.

Come si vede dai metodi, per un oggetto di questa classe è possibile specificare un listener di tipo `ItemListener`, come nel caso della classe `CheckBox`, per reagire immediatamente a un cambio di stato.

Osserviamo ora come sia possibile scrivere una classe che esegua lo stesso tipo di operazioni dell'esempio precedente utilizzando però un oggetto della classe `Choice` (vedi Figura 17.7).

```
import java.awt.*;
import java.awt.event.*;

public class Scelte extends Frame {
   Choice chTest;
   Label lT;
   public static void main (String argv[]) {
      Scelte ist = new Scelte();
   }
   Scelte() {
      setLayout(null);
      setBounds (30, 10, 300, 200);
      setTitle (getClass().getName());
      chTest = new Choice ();
      chTest.add ("Test1");
      chTest.add ("Test2");
      chTest.setBounds (50, 50, 100, 20);
      chTest.addItemListener (new ItemListener() {
         public void itemStateChanged(ItemEvent e) {
            vediStato ();
         }
      });
      lT = new Label ();
```

```
        lT.setBounds (50, 150, 250, 20);
        vediStato ();
        add(chTest);
        add(lT);
        setVisible(true);
    }
    void vediStato () {
        lT.setText("Selezione=" + chTest.getSelectedItem());
    }
}
```

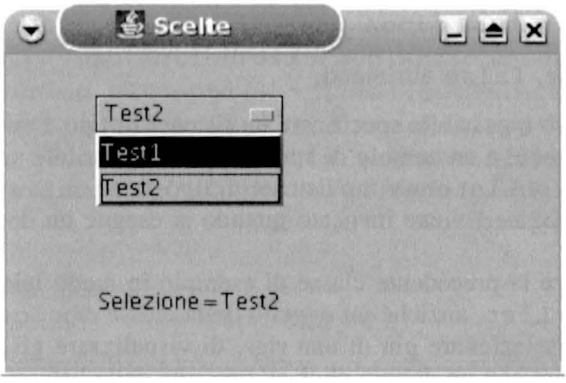


Figura 17.7 Pop-up utilizzando `Choice`

17.7 Liste

La classe `List` permette di ottenere funzionalità molto simili a quelle della classe `Choice` differendo da quest'ultima per il fatto di visualizzare costantemente un numero determinato di righe e per il fatto che, tramite un'opzione, consente di selezionare più di una riga. Se le righe associate a un oggetto di questo tipo non possono essere visualizzate contemporaneamente nell'area assegnata, compare automaticamente una scrollbar che permette lo scorrimento delle righe in alto e in basso.

Per la creazione di oggetti di questa classe sono disponibili i costruttori

- `public List(int nRighe, boolean multiSel)`: crea una lista con un numero di linee pari a `nRighe` e se questo parametro vale 0, viene preso un numero di righe di default. Questo parametro ha significato solo per alcuni impaginatori mentre è ignorato se si effettua l'impaginazione manuale. Se il parametro `multiSel` vale `true`, l'oggetto consente di selezionare più righe contemporaneamente, altrimenti la selezione di una riga comporta la deselezione automatica delle rimanenti.

- `public List(int nRighe)`: corrisponde a `List(nRighe, false)`.
- `public List()`: corrisponde a `List(0, false)`.

I metodi che abbiamo visto nella classe `Choice` sono definiti anche in questa classe e quelli seguenti servono per poter trattare con liste a selezione multipla.

- `public int[] getSelectedIndexes()`: restituisce un array con gli indici degli elementi selezionati.
- `public String[] getSelectedItems()`: restituisce un array con le stringhe corrispondenti agli elementi selezionati.
- `public void setMultipleMode(boolean b)`: imposta la possibilità di effettuare selezioni multiple o meno in base all'argomento `b`.
- `public boolean isMultipleMode()`: restituisce `true` se l'oggetto consente selezioni multiple, `false` altrimenti.

Anche in questo caso è possibile specificare un listener di tipo `ItemListener`, per reagire immediatamente a un cambio di stato. È inoltre possibile specificare, tramite il metodo `addActionListener`, un listener di tipo `ActionListener` il cui metodo `actionPerformed` viene invocato quando si esegue un doppio click su una riga della lista.

Modifichiamo ora la precedente classe di esempio in modo tale che impieghi un oggetto della classe `List`, anziché un oggetto della classe `Choice`, aggiungendo le possibilità di poter selezionare più di una riga, di visualizzare gli indici delle righe selezionate e di uscire con un doppio click su una riga della lista, visualizzando sulla console l'elenco delle righe selezionate (Figura 17.8).

```
import java.awt.*;
import java.awt.event.*;

public class Liste extends Frame {
   List lTest;
   Label lT1;
   Label lT2;
   public static void main (String argv[]) {
      Liste ist = new Liste();
   }
   Liste() {
      setLayout(null);
      setBounds (30, 10, 300, 200);
      setTitle (getClass().getName());
      lTest = new List (0, true);
```

```
        lTest.add ("Test zero");
        lTest.add ("Test uno");
        lTest.add ("Test due");
        lTest.add ("Test tre");
        lTest.add ("Test quattro");
        lTest.add ("Test cinque");
        lTest.setBounds (50, 50, 100, 80);
        lTest.addItemListener (new ItemListener() {
           public void itemStateChanged(ItemEvent e) {
              vediStato ();
           }
        });
        lTest.addActionListener (new ActionListener() {
           public void actionPerformed(ActionEvent e) {
              String elementi[] = lTest.getSelectedItems();
              for (int i = 0; i < elementi.length; i++)
                 System.out.println (elementi[i]);
              System.exit (0);
           }
        });
        lT1 = new Label ();
        lT1.setBounds (50, 140, 250, 20);
        lT2 = new Label ();
        lT2.setBounds (50, 160, 250, 20);
        vediStato ();
        add(lTest);
        add(lT1);
        add(lT2);
        setVisible(true);
    }
    void vediStato () {
        lT1.setText("Selezione=" + lTest.getSelectedItem());
        String multi = "Multiple=";
        int indici[] = lTest.getSelectedIndexes();
        for (int i = 0; i < indici.length; i++)
            multi += " " + indici[i];
        lT2.setText(multi);
    }
}
```

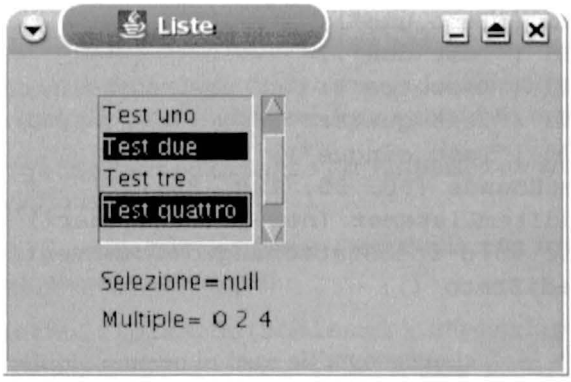


Figura 17.8 Scelte multiple da una lista

17.8 Barre di scorrimento

Gli oggetti della classe `List` (e in alcuni ambienti anche quelli della classe `Choice`) visualizzano una barra di scorrimento automaticamente ogni volta che la lista di stringhe non entra nell'area assegnata. Una barra di scorrimento potrebbe essere utilizzata anche per altri scopi come, per esempio, scorrere immagini. La classe `Scrollbar` rappresenta una barra di scorrimento che può essere impiegata per scopi di questo tipo.

Per capirne il funzionamento, è necessario introdurre alcune definizioni che descrivono le proprietà di un oggetto di tale tipo.

- *orientamento*: rappresenta il verso in cui il cursore di una scrollbar può scorrere. Due costanti intere rappresentano gli orientamenti possibili: `Scrollbar.VERTICAL`, che indica che il cursore può scorrere dall'alto verso il basso e viceversa, e `Scrollbar.HORIZONTAL`, che indica che il cursore può scorrere da sinistra a destra e viceversa.
- *valore corrente*: è un intero corrispondente alla posizione del cursore.
- *valore minimo*: è il valore minimo che una scrollbar può restituire, corrispondente alla posizione del cursore completamente a sinistra per una scrollbar orizzontale, e completamente in alto per una scrollbar verticale.
- *valore massimo*: è il valore teorico massimo che può restituire una scrollbar.
- *area visibile*: supponendo che la scrollbar sia associata a una pagina, di questa tipicamente sarà visibile solo una parte che possiamo rappresentare con una frazione. Se, per esempio, si ha una pagina con 100 righe di testo di cui se ne possono vedere solo 20 per volta, la frazione visibile sarà $^{20}/_{100}$. Definiamo allora *area visibile* il risultato intero dell'espressione (*<valore massimo>* - *<valore minimo>*) * *<frazione visibile>*.

- *incremento unitario*: intero da aggiungere o sottrarre al valore corrente ogni volta che viene effettuato un click su una delle frecce alle estremità della scrollbar.
- *incremento di blocco*: intero da aggiungere o sottrarre al valore corrente ogni volta che viene effettuato un click sull'area compresa tra il cursore e le frecce agli estremi.

Forti di queste definizioni, possiamo ora vedere i costruttori della classe.

- `public Scrollbar(int or, int val, int areaVis, int min, int max)`: costruisce un'istanza con orientamento uguale a `or`, valore corrente iniziale uguale a `val`, area visibile uguale a `areaVis`, valore minimo uguale a `min` e valore massimo uguale a `max`.
- `public Scrollbar(int or)`: corrisponde a `Scrollbar(or,0,10,0,100)`.
- `public Scrollbar()`: corrisponde a `Scrollbar(Scrollbar.VERTICAL)`.

Nella classe è disponibile un discreto numero di metodi, molti dei quali permettono di ricuperare e impostare le caratteristiche dell'oggetto; alcuni dei più utilizzati sono i seguenti:

- `public int getValue()`: restituisce il valore corrente.
- `public void setValue(int nuovoVal)`: permette di impostare un nuovo valore corrente.
- `public void setUnitIncrement(int v)`: permette di impostare l'incremento unitario, che per default è 1.
- `public void setBlockIncrement(int v)`: permette di impostare l'incremento di blocco, che per default è 10. Tipicamente questo valore dovrebbe essere lo stesso dell'area visibile.
- `public void addAdjustmentListener(AdjustmentListener l)`: permette di specificare un listener di tipo `AdjustmentListener`.

Come risulta dall'ultimo metodo, un oggetto di questa classe può avere un listener di tipo `AdjustmentListener`, il cui unico metodo, `public void adjustmentValueChanged (AdjustmentEvent e)` viene invocato ogni volta che l'utente agisce sulla scrollbar. L'argomento di tale metodo è di tipo `AdjustmentEvent` che definisce il metodo `getValue()`, che restituisce il valore corrispondente all'azione esercitata dall'utente.

La classe seguente visualizza una scrollbar e il valore corrispondente a ogni azione dell'utente su di essa (vedi Figura 17.9).

```
import java.awt.*;
import java.awt.event.*;

public class BarraScorrimento extends Frame {
  Scrollbar sTest;
  Label lT;
  public static void main (String argv[]) {
    BarraScorrimento ist = new BarraScorrimento();
  }
```

```
  BarraScorrimento() {
    setLayout(null);
    setBounds (30, 10, 300, 200);
    setTitle (getClass().getName());
    sTest = new Scrollbar (Scrollbar.HORIZONTAL, 400,
                                                  200, 0, 1000);
    sTest.setBlockIncrement (200);
    sTest.setBounds (50, 50, 150, 30);
    sTest.addAdjustmentListener (new AdjustmentListener() {
      public void adjustmentValueChanged(AdjustmentEvent e) {
         sTest.setValue(e.getValue());
         lT.setText ("Valore=" + e.getValue());
      }
    });
    lT = new Label ("Valore=" + sTest.getValue());
    lT.setBounds (50, 100, 250, 20);
    add(sTest);
    add(lT);
    setVisible(true);
  }
}
```

Va sottolineato che il valore corrente assunto dalla scrollbar non raggiunge mai il valore massimo, ma ne rimane al di sotto almeno di una quantità pari all'area visibile: infatti nella visualizzazione di una pagina, se il valore corrente raggiungesse il valore massimo, significherebbe che stiamo visualizzando la pagina dopo l'ultima.

Figura 17.9 Scrollbar

17.9 La classe `Canvas`

La classe `Canvas` di per se non è un controllo, ma definisce semplicemente un'area all'interno di una finestra su cui è possibile disegnare e che riceve tutti gli eventi. Il suo utilizzo principale è quello di permettere la costruzione di controlli personalizzati tramite derivazione. Si può quindi derivare da essa una classe che rappresenta un qualsiasi tipo di controllo non definito nel set standard dell'AWT, come un bottone che visualizza un'immagine anziché una stringa, una *toolbar* ecc. Se si utilizza un oggetto di questa classe direttamente, esso non fa altro che riempire la propria area con il colore di background.

17.10 Menu

I *menu* non sono controlli pur funzionando in modo simile e, infatti, le classi utilizzate per il loro trattamento hanno una gerarchia separata dagli altri oggetti grafici derivando tutte da `MenuComponent` che, a sua volta, deriva direttamente da `Object`.

A un oggetto della classe `Frame` può venire associata una barra di menu, rappresentata da un oggetto della classe `MenuBar`, tramite il metodo `public void setMenuBar(MenuBar mb)`. Riportiamo il costruttore e alcuni metodi della classe `MenuBar`.

- `public MenuBar()`: crea un'istanza della classe.
- `public Menu add(Menu m)`: aggiunge l'oggetto specificato alla barra dei menu.
- `public void remove(int indice)`: rimuove il menu in posizione `indice`.
- `public void remove(MenuComponent m)`: rimuove il menu specificato.
- `public int getMenuCount()`: restituisce il numero di menu contenuti.
- `public void setHelpMenu(Menu m)`: imposta il menu specificato come menu d'aiuto.

Una volta che un oggetto della classe `MenuBar` è stato creato, a esso possono essere aggiunti, tramite il metodo `add`, un certo numero di oggetti della classe `Menu`, ciascuno dei quali rappresenta per l'appunto un menu.

A un oggetto di quest'ultima classe può essere aggiunto un certo numero di oggetti di tipo `MenuItem`, ciascuno dei quali rappresenta una scelta del menu. Poiché la classe `Menu` deriva da `MenuItem`, è ammesso aggiungere un oggetto di tipo `Menu` a un altro, realizzando in tal modo menu a più livelli. Vediamo alcuni costruttori e metodi della classe `MenuItem`.

- `public MenuItem(String etichetta)`: crea un'istanza con l'etichetta specificata.
- `public void setEnabled(boolean b)`: abilita o disabilita l'elemento del menu a seconda se l'argomento vale `true` o `false`.
- `public void setActionCommand(String comando)`: assegna il parametro alla stringa di comando.

- `public void addActionListener(ActionListener l)`: permette di aggiungere un listener di tipo `ActionListener` all'istanza.

Si noterà che gli ultimi due metodi sono analoghi a quelli già visti parlando della gestione degli eventi della classe `Button` e infatti per rilevare una selezione da un menu si opera nello stesso modo in cui si rileva la pressione di un pulsante.

La classe `Menu`, derivando da `MenuItem`, ne eredita i metodi e in più definisce il costruttore e i metodi seguenti:

- `public Menu(String etichetta)`: crea un'istanza con l'etichetta specificata.
- `public MenuItem add(MenuItem mi)`: aggiunge l'elemento specificato e lo restituisce.
- `public void addSeparator()`: aggiunge un separatore.
- `public int getItemCount()`: restituisce il numero di elementi contenuti.
- `public void remove(int indice)`: rimuove l'elemento nella posizione specificata.
- `public void remove(MenuComponent el)`: rimuove l'elemento specificato.
- `public void removeAll()`: rimuove tutti gli elementi.

Scriviamo una classe che visualizza una finestra con due menu e un listener che visualizza le scelte effettuate (vedi Figura 17.10).

```
import java.awt.*;
import java.awt.event.*;

public class MenuTest  extends Frame
                        implements ActionListener {
   Label lT;
   public static void main (String argv[]) {
      MenuTest ist = new MenuTest();
   }
   MenuTest() {
      setLayout(null);
      setBounds (30, 10, 300, 200);
      setTitle (getClass().getName());
      MenuBar barraMenu = new MenuBar ();
      setMenuBar (barraMenu);
      Menu arch = new Menu("Archivio");
      arch.add (new
                  MenuItem("Nuovo")).addActionListener(this);
      arch.add (new
                  MenuItem("Apri")).addActionListener(this);
      arch.add (new
                  MenuItem("Aiuto")).addActionListener(this);
      arch.add (new
                  MenuItem("Esci")).addActionListener(this);
```

```
        barraMenu.add(arch);

        Menu sm = new Menu("Sotto menu");
        sm.add (new MenuItem("Sotto menu
                    1")).addActionListener(this);
        sm.add (new MenuItem("Sotto menu
                    2")).addActionListener(this);

        Menu modif = new Menu("Modifica");
        modif.add (new
                    MenuItem("Copia")).addActionListener(this);
        modif.add (new
                   MenuItem("Incolla")).addActionListener(this);
        modif.add (new
                  MenuItem("Cancella")).addActionListener(this);
        modif.addSeparator();
        modif.add (sm);
        barraMenu.add(modif);

        lT = new Label ();
        lT.setBounds (50, 150, 250, 20);
        add(lT);
        setVisible(true);
    }
    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("Esci"))
            System.exit(0);
        else
            lT.setText ("Menu:" + e.getActionCommand());
    }
}
```

Figura 17.10 Menu in cascata

Notare come sia stato possibile aggiungere un listener a ciascun elemento dei menu senza dover ricorrere a delle variabili grazie al fatto che il metodo `add(MenuItem mi)` restituisce l'oggetto fornito come argomento.

17.11 Impaginazione automatica

Abbiamo detto che è possibile associare un oggetto che implementa l'interfaccia `LayoutManager` a un'istanza appartenente a una classe derivata da `Container` in modo tale da ottenere una disposizione automatica. Nell'AWT sono definite alcune classi che effettuano l'impaginazione secondo strategie diverse.

Un oggetto della classe `BorderLayout` è associato per default a ogni istanza di una classe derivata da `Frame`. Per utilizzare correttamente questo impaginatore, è necessario aggiungere i componenti alla finestra utilizzando il metodo `public void add(Component comp, Object vincoli)` e specificare come secondo parametro una delle stringhe `"North"`, `"East"`, `"South"`, `"West"`, `"Center"`. L'impaginatore dispone i componenti ai bordi della finestra o nel centro a seconda del vincolo associato, in modo tale da riempire completamente tutto lo spazio. Due sono i costruttori disponibili e precisamente

- `public BorderLayout(int sepOriz, int sepVert)`: crea un'istanza in cui la distanza orizzontale tra i componenti è specificata in pixel da `sepOriz`, mentre la distanza verticale tra i componenti è specificata in pixel da `sepVert`.
- `public BorderLayout()`: corrisponde a `BorderLayout(0, 0)`.

La classe seguente usa questo impaginatore per disporre cinque bottoni (Figura 17.11).

```
import java.awt.*;

public class Bordo extends Frame {
   public static void main (String argv[]) {
      Bordo ist = new Bordo();
   }
   Bordo () {
      setLayout (new BorderLayout());
      setTitle (getClass().getName());
      add (new Button("Uno"), "North");
      add (new Button("Due"), "East");
      add (new Button("Tre"), "South");
      add (new Button("Quattro"), "West");
      add (new Button("Cinque"), "Center");
      setBounds (30, 10, 300, 200);
      setVisible(true);
   }
}
```

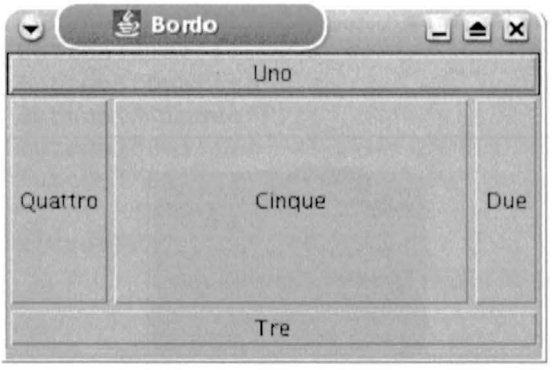


Figura 17.11 Disposizione dei bottoni tramite `BorderLayout`

L'utilizzo degli impaginatori è interessante sia perché le finestre create con il loro utilizzo si adattano automaticamente a ogni ridimensionamento della finestra da parte dell'operatore, sia perché permette la realizzazione di interfacce grafiche senza doversi preoccupare di calcolare la dimensione di ciascun componente in pixel. Da questo punto di vista l'esempio appena visto è ancora carente in quanto necessita del calcolo della dimensione della finestra principale. È invece possibile fare in modo che una finestra calcoli automaticamente la propria dimensione in modo opportuno a seconda degli oggetti che contiene e dell'impaginatore impostato. La classe `Component` implementa il metodo `getPreferredSize()` che restituisce un oggetto della classe `Dimension` contenente la dimensione preferita del componente. Questo metodo è opportunamente ridefinito nella sottoclasse `Container` in modo da poter calcolare la dimensione adeguata in base ai componenti contenuti.

Ecco allora come potrebbe essere modificato il programma precedente affinché calcoli autonomamente la dimensione opportuna.

```
import java.awt.*;

public class BordoAuto extends Frame {
   public static void main (String argv[]) {
      BordoAuto ist = new BordoAuto();
   }
   BordoAuto () {
      setLayout (new BorderLayout());
      setTitle (getClass().getName());
      add (new Button("Uno"), "North");
      add (new Button("Due"), "East");
      add (new Button("Tre"), "South");
      add (new Button("Quattro"), "West");
      add (new Button("Cinque"), "Center");
```

```
        pack();
        setVisible(true);
    }
}
```

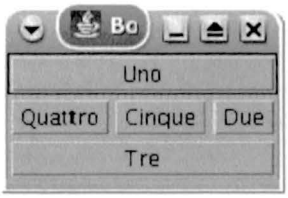


Figura 17.12 Calcolo automatico della dimensione della finestra

Notare che, una volta inseriti tutti i componenti della finestra, è stata sostituita l'invocazione del metodo `setBounds()` con l'invocazione del metodo `pack()`. Quest'ultimo forza il calcolo delle dimensioni della finestra e deve essere invocato dopo che tutti i componenti sono stati aggiunti.

La classe `FlowLayout` implementa un impaginatore usato tipicamente per disporre bottoni in una finestra. La strategia utilizzata è quella di disporre i componenti da sinistra a destra e dall'alto in basso. Questa classe dispone dei seguenti tre costruttori:

- `public FlowLayout(int allineamento, int sepOriz, int sepVert)`: crea un'istanza in cui la distanza orizzontale tra i componenti è specificata in pixel da `sepOriz`, mentre la distanza verticale tra i componenti è specificata in pixel da `sepVert`. Il parametro `allineamento` può assumere i valori `FlowLayout.LEFT`, `FlowLayout.CENTER` e `FlowLayout.RIGHT`, che impostano l'allineamento dei componenti rispettivamente a sinistra, centrale e a destra.
- `public FlowLayout(int allineamento)`: corrisponde a `FlowLayout(allineamento, 5, 5)`.
- `public FlowLayout()`: corrisponde a `FlowLayout(FlowLayout.CENTER)`.

La classe seguente usa questo tipo di impaginatore per visualizzare sette bottoni (vedi Figura 17.13).

```
import java.awt.*;

public class Flusso extends Frame {
    public static void main (String argv[]) {
        Flusso ist = new Flusso();
    }
    Flusso () {
        setLayout (new FlowLayout());
        setTitle (getClass().getName());
```

```
        add (new Button("Uno"));
        add (new Button("Due"));
        add (new Button("Tre"));
        add (new Button("Quattro"));
        add (new Button("Cinque"));
        add (new Button("Sei"));
        add (new Button("Sette"));
        pack();
        setVisible(true);
    }
}
```

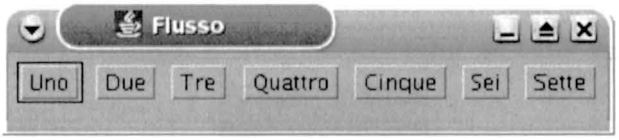


Figura 17.13 Disposizione dei bottoni tramite `FlowLayout`

La classe `GridLayout` implementa un impaginatore che crea una griglia nella finestra e pone ciascun componente in una delle sue celle. È possibile specificare il numero di righe e di colonne della griglia: nel caso che il numero di celle non sia sufficiente a contenere i componenti, l'impaginatore aumenta automaticamente il numero di colonne fino ad arrivare a un numero di celle sufficiente. Anche questa classe dispone di tre costruttori.

- `public GridLayout(int rig, int col, int sepOriz, int sepVert)`: crea un'istanza che definisce una griglia con `rig` righe e `col` colonne in cui la distanza orizzontale tra i componenti è specificata in pixel da `sepOriz`, mentre la distanza verticale tra i componenti è specificata in pixel da `sepVert`.
- `public GridLayout(int rig, int col)`: corrisponde a `GridLayout(rig, col, 0, 0)`.
- `public GridLayout()`: corrisponde a `GridLayout(1, 1, 0, 0)`.

La classe successiva usa un impaginatore di questo tipo con 4 righe e due colonne per visualizzare sette bottoni (vedi Figura 17.14).

```
import java.awt.*;

public class Griglia extends Frame {
   public static void main (String argv[]) {
      Griglia ist = new Griglia();
   }
```

```
    Griglia () {
        setLayout (new GridLayout(4,2));
        setTitle (getClass().getName());
        add (new Button("Uno"));
        add (new Button("Due"));
        add (new Button("Tre"));
        add (new Button("Quattro"));
        add (new Button("Cinque"));
        add (new Button("Sei"));
        add (new Button("Sette"));
        pack();
        setVisible(true);
    }
}
```

Figura 17.14 Disposizione dei bottoni tramite `GridLayout`

17.12 Composizione di impaginatori

L'uso degli impaginatori apre degli orizzonti interessanti, è evidente però che nessun impaginatore offre da solo una flessibilità tale da poter essere impiegato in applicazioni reali. Per poter creare delle interfacce grafiche più complesse è necessario suddividere la finestra principale in aree logiche distinte, ciascuna col proprio impaginatore, e quindi usare un impaginatore per sistemare ciascuna area.

La classe `Panel` è la più semplice classe contenitrice. Essa fornisce uno spazio in cui possono essere inseriti componenti grafici di qualsiasi tipo, compresi altri oggetti di tipo `Panel`. Dato che essa deriva da `Container`, ogni istanza ha associato un impaginatore che, per default, è di tipo `FlowLayout`. Essa ha i seguenti costruttori:

- `public Panel()`: crea un'istanza con l'impaginatore di default.
- `public Panel(LayoutManager)`: crea un'istanza con l'impaginatore specificato.

Vediamo ora come possono essere combinati gli impaginatori visti in modo da creare una finestra di dialogo accettabile. Riprendiamo la classe `LineaDiTesto` vista precedentemente e modifichiamola in modo che, oltre a dimensionarsi automaticamente, ponga un testo descrittivo accanto a ciascun oggetto di tipo `TextField`. Possiamo

quindi pensare a una finestra divisa in due parti: la prima in alto, organizzata come una griglia di due righe e due colonne, dove vengono riportati i due `TextField` con le relative descrizioni, e la seconda in basso con i bottoni da premere. La classe adatta potrebbe essere la seguente:

```
import java.awt.*;
import java.awt.event.*;

public class AutoDialogo extends Frame {
  private TextField t1;
  private TextField t2;
  public static void main (String argv[]) {
    AutoDialogo ist = new AutoDialogo();
  }
  AutoDialogo () {
    Panel pannelloTesto = new Panel (new GridLayout(0, 2,
                                                    3, 3));
    Panel pannelloBottoni = new Panel ();
    setTitle (getClass().getName());

    pannelloTesto.add (new Label("Testo 1", Label.LEFT));
    pannelloTesto.add (t1 = new TextField(""));
    pannelloTesto.add (new Label("Testo 2", Label.LEFT));
    pannelloTesto.add (t2 = new TextField(""));

    pannelloBottoni.setLayout (new FlowLayout());
    ((Button) pannelloBottoni.add (new Button("Scambia")))
                .addActionListener (new ActionListener() {
       public void actionPerformed(ActionEvent e) {
          String scambio = t1.getText();
          t1.setText (t2.getText());
          t2.setText (scambio);
       }
    });
    ((Button) pannelloBottoni.add (new Button("Esci")))
                .addActionListener (new ActionListener() {
       public void actionPerformed(ActionEvent e) {
          System.exit (0);
       }
    });

    add (pannelloTesto, "Center");
    add (pannelloBottoni, "South");
    pack();

    setVisible (true);
  }
}
```

Figura 17.15 Finestra di dialogo

Nell'esempio sono stati usati tutti gli impaginatori visti e cioè `GridLayout`, `BorderLayout` e `FlowLayout`. Gli ultimi due non appaiono esplicitamente in quanto sono stati usati quelli di default associati rispettivamente all'istanza di `Frame` e di `Panel`. Notare come con un'unica operazione sia stato possibile creare un bottone, aggiungerlo all'istanza di `Panel` e definirne il comportamento all'atto dell'attivazione, senza dover definire alcuna variabile. Questo è stato reso possibile dal fatto che il metodo `add()` restituisce l'argomento con cui è stato invocato. Dato però che `add()` restituisce sempre un oggetto di tipo `Component`, è stato necessario usare un cast. Questo modo di concatenare metodi è molto comune nei linguaggi a oggetti; in Java se ne fa solitamente un uso più limitato rispetto ad altri linguaggi in quanto la sua notazione non incoraggia questo uso e tende a divenire molto complessa da interpretare.

Domande di verifica

1. A cosa serve la classe `Button` e quali sono i suoi metodi?
2. Descrivere le funzioni della classe `TextComponent`, delle sue classi derivate e dei loro metodi.
3. Quali sono i costruttori della classe `Label`? E i suoi metodi presentati nel testo?
4. Quali sono i costruttori della classe `Checkbox`? E i suoi metodi presentati nel testo?
5. Come agisce ognuno dei metodi disponibili in `CheckboxGroup`?
6. Perché si usa la classe `Choice` e quali costruttori e metodi possiede?
7. Perché si usa la classe `List` e quali costruttori e metodi possiede?
8. In relazione alla *scrollbar* cosa significano le proprietà orientamento, valore corrente, valore minimo, valore massimo, area visibile, incremento unitario e incremento di blocco?
9. Quali sono i costruttori e i metodi della classe `Scrollbar`?

10. Descrivere la classe `MenuComponent` e i suoi metodi? In che relazione è con la classe `MenuItem` e quali sono i costruttori di quest'ultima? Quali sono i metodi della classe `Menu` e da quale classe deriva?
11. A cosa serve la classe `BorderLayout`? Quali costruttori mette a disposizione?
12. Qual è lo scopo della classe `Panel`? Quali sono i suoi costruttori?

Esercizi

1. Risolvere tutti gli esercizi dati dal Capitolo 4 utilizzando per l'interazione con l'utente l'interfaccia grafica. Scegliere di volta in volta gli oggetti grafici adeguati e eventualmente utilizzare più finestre per l'interazione.
2. Riprendere il programma per la gestione dell'agenda telefonica esposto come esercizio nel Capitolo 12 e fornirlo di un'adeguata interfaccia grafica.
3. Scrivere un programma che mostri tutti i file di testo presenti in una directory utilizzando una list box e quindi ne mostri il contenuto.
4. Scrivere un programma che permetta di disegnare liberamente linee, rettangoli, cerchi ecc. utilizzando il mouse. Limitare a propria scelta il numero delle funzionalità disponibili.
5. Realizzare un editore di testi con funzionalità minime (questo e gli esercizi seguenti implicano un notevole sforzo e necessitano di tempo; si deve inoltre avere la possibilità di consultare i manuali del linguaggio).
6. Nell'Esercizio 4, fare in modo che a ogni figura sullo schermo sia associato un oggetto, in modo tale che un disegno sia rappresentato da un vettore di oggetti. Permettere all'utente di salvare questo vettore come oggetto persistente e quindi ripristinarlo, in modo tale che i disegni non siano perduti alla fine della sessione.
7. Realizzare un foglio elettronico con funzionalità minime.

Capitolo 18

Swing

Obiettivi didattici

- I componenti grafici
- Applicazioni
- Look and Feel
- Disegno di componenti

Abbiamo visto che usando l'AWT si è portati a creare interfacce per l'utente con il look and feel caratteristico del sistema operativo che esegue l'applicazione. Abbiamo detto anche che questa è una caratteristica voluta inizialmente dagli sviluppatori per non confondere gli utilizzatori con un'interfaccia differente da quella che sono abituati a usare normalmente. Nel corso del tempo però, questo approccio ha dimostrato diversi punti deboli, tra cui:

- per avere la certezza che un programma grafico funzioni correttamente è necessario testarlo su tutte le piattaforme su cui si prevede debba girare. Non è inusuale, infatti, che un programma sviluppato su una determinata piattaforma sia praticamente inutilizzabile su un'altra a causa di differenti modalità di visualizzazione.
- il set di componenti disponibile su AWT è limitato dal fatto di dover essere disponibile su tutte le piattaforme supportate, mentre oggi le interfacce grafiche più diffuse ne hanno a disposizione sempre di più.

Per risolvere questi e altri problemi, è stata sviluppata una libreria di componenti grafici totalmente scritti in Java, detta *Swing*. In pratica, mentre in un'applicazione AWT a ogni oggetto grafico in una finestra corrisponde una risorsa richiesta al siste-

ma operativo sottostante, usando Swing solo le finestre esterne (top level container) vengono richieste al sistema operativo tramite chiamate all'AWT; il resto dei componenti è gestito totalmente in Java. Componenti di questo tipo vengono detti *leggeri* (lightweight), in quanto usano meno risorse del sistema operativo, escludendo il tempo di CPU. Per contrapposizione, i componenti di AWT vengono detti *pesanti* (heavyweight).

Per fare un esempio, la classe `AutoDialogo`, che abbiamo visto come ultimo esempio nel capitolo sull'AWT, tipicamente richiede al sistema operativo l'apertura di ben nove finestre, vale a dire la cornice esterna, i due pannelli, le due etichette, i due campi testo e i due bottoni. La classe equivalente scritta usando Swing, che vedremo nel prossimo paragrafo, richiede invece l'apertura di un'unica finestra.

I componenti leggeri permettono di ottenere un look and feel indipendente dalla piattaforma su cui vengono eseguite le applicazioni. Swing permette di impostare diversi tipi di look and feel da programma in qualsiasi momento. Si dice quindi che Swing permette di avere un *look and feel inseribile* (pluggable look and feel). Se comunque si desidera fornire un'applicazione che non disorienti l'utente con un aspetto alieno rispetto alle altre presenti sul computer, per i sistemi più diffusi è disponibile un look and feel che emula in modo fedele quello del sistema operativo ospite.

Tutti i componenti grafici disponibili in AWT sono stati reimplementati e arricchiti in Swing; per esempio i bottoni possono contenere immagini e possono essere di forma non rettangolare. In più sono stati aggiunti numerosi componenti di alto livello, come griglie (grid), viste ad albero (tree view), barre di strumenti (toolbar), finestre di dialogo a linguette (tabbed dialog) ecc. I componenti con medesime funzioni in AWT e Swing possono essere riconosciuti facilmente dal nome avendo questi ultimi una `J` maiuscola in testa (per esempio, `Button` e `JButton`). Questa `J` si spiega con il fatto che Swing rappresenta il nucleo di una serie di package più ampia, chiamata JFC (Java Foundation Classes), che ha lo scopo di permettere lo sviluppo di applicazioni grafiche.

Swing è stato fornito come package separato dopo i primi rilasci del JDK 1.1 e fa parte della distribuzione standard a partire dal JDK 1.2.

18.1 Un'applicazione Swing

Swing e AWT usano lo stesso modello di programmazione, gli stessi meccanismi di delega di gestione degli eventi e le stesse classi per rappresentare gli eventi.

La classe base dei componenti di Swing è `JComponent` che deriva da `java.awt.Container` e quindi da `java.awt.Component`. Essa è stata sviluppata in modo da essere facilmente estensibile e da racchiudere tutte le caratteristiche di un componente Swing in modo tale che sia semplice derivarne nuovi componenti omogenei con quelli già esistenti.

Riprendiamo allora la classe `AutoDialogo`, che abbiamo visto parlando dell'AWT, e vediamo quali cambiamenti sono necessari per ottenerne una analoga che usi Swing.

✔ **NOTA**

Nell'esempio non abbiamo messo alcuna istruzione per impostare un look and feel. Su diverse piattaforme testate, in questi casi viene sempre usato il look and feel di default chiamato "Metal" o anche "Java Look and Feel" (JLF), ma la documentazione non sembra garantire che sia sempre così. Vedremo nel prossimo paragrafo come impostare i look and feel.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JAutoDialogo extends JFrame {
  private JTextField t1;
  private JTextField t2;
  public static void main (String argv[]) {
    JAutoDialogo ist = new JAutoDialogo();
  }
  JAutoDialogo () {
    JPanel pannelloTesto = new JPanel (new GridLayout(0,
                                                    2, 3, 3));
    JPanel pannelloBottoni = new JPanel ();
    setTitle (getClass().getName());

    pannelloTesto.add (new JLabel("Testo 1", JLabel.LEFT));
    pannelloTesto.add (t1 = new JTextField(""));
    pannelloTesto.add (new JLabel("Testo 2", JLabel.LEFT));
    pannelloTesto.add (t2 = new JTextField(""));

    pannelloBottoni.setLayout (new FlowLayout());
    ((JButton) pannelloBottoni.add (new JButton("Scambia")))
                .addActionListener (new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         String scambio = t1.getText();
         t1.setText (t2.getText());
         t2.setText (scambio);
      }
    });
    ((JButton) pannelloBottoni.add (new JButton("Esci")))
                .addActionListener (new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         System.exit (0);
      }
    });
```

```
        getContentPane().add (pannelloTesto, "Center");   /*Cfrt.
                                                    versione AWT */
        getContentPane().add (pannelloBottoni, "South"); /*Cfrt.
                                                    versione AWT */
        pack();
        setVisible (true);
    }
}
```

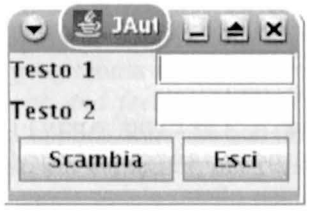


Figura 18.1 Finestra Swing

Confrontando il codice di questa classe con l'equivalente AWT sembra che, da un punto di vista di programmazione, sia sufficiente aggiungere semplicemente una `J` maiuscola davanti ai nomi dei componenti per passare dall'uso di AWT a Swing.

Dal confronto però appare evidente una differenza tra le classi `Frame` e `JFrame`: la prima, infatti, si comporta come un contenitore, permette cioè di aggiungere componenti e/o impostare impaginatori direttamente a se stessa, mentre la seconda no. La classe `JFrame` ha però sempre associato un oggetto della classe `Container` dove è possibile aggiungere oggetti e/o impostare impaginatori, e tale oggetto può essere ottenuto tramite il metodo `getContentPane()`. Questo oggetto ha sempre associato per default un impaginatore di tipo `BorderLayout` e occupa solo spazio interno della finestra: un componente grafico in posizione 0,0 appare quindi completamente visibile; se, invece, è posizionato alle stesse coordinate all'interno di un oggetto di tipo `Frame`, esso viene coperto, almeno in parte, dall'intestazione della finestra. Questa differenza di uso tra `Frame` e `JFrame` non è molto elegante e può provocare errori per disattenzione, per cui, a partire dal JDK 1.5, sono stati modificati i metodi `add` e le sue varianti, `remove` e `setLayout`, in modo da eseguire implicitamente una `getContentPane()`.

Se, nell'esempio, avessimo aggiunto i due oggetti di tipo `JPanel` all'istanza di `JFrame`, anziché all'oggetto restituito da `getContentPane()`, anche un compilatore di una versione precedente la 1.5 non avrebbe segnalato alcun tipo di problema, perché la classe `JFrame` deriva direttamente dalla classe `Frame` e quindi ne eredita tutti i metodi. Solo eseguendo il programma con una JVM di versione inferiore alla 1.5 avremmo ottenuto un'eccezione del tipo:

```
Exception in thread "main" java.lang.Error:
Do not use JAutoDialogo.add() use
JAutoDialogo.getContentPane().add() instead
```

Con una versione di JVM pari o superiore alla 1.5 invece non si manifesta alcun problema, anche se la classe è stata compilata con una versione precedente.

Mandando in esecuzione la classe compilata si può notare un'altra differenza, questa volta di comportamento, con la versione AWT. Eseguendo un click sulla crocetta in alto a destra destinata alla chiusura della finestra, essa si chiude effettivamente anche se nel sorgente non abbiamo gestito questo tipo di evento. Il programma in ogni caso non termina e deve essere concluso mediante la pressione del tasto d'interruzione.

La classe `JFrame`, infatti, gestisce automaticamente il tasto di chiusura della finestra, pur non impedendo di gestire questo tipo di eventi nel modo già visto: per default essa causa la scomparsa della finestra senza distruggerla, ma tale comportamento può essere modificato tramite l'invocazione del metodo `setDefaultCloseOperation (int modo)`. A tale metodo può essere passato uno dei seguenti argomenti:

- `WindowConstants.DO_NOTHING_ON_CLOSE`: non fa niente e quindi rende il comportamento di una `JFrame` analogo a quello di una `Frame`.
- `WindowConstants.HIDE_ON_CLOSE`: rende invisibile la finestra dopo aver invocato eventuali oggetti della classe `WindowListener` registrati; questo è il comportamento di default.
- `WindowConstants.DISPOSE_ON_CLOSE`: rende invisibile e distrugge la finestra dopo aver invocato eventuali oggetti della classe `WindowListener` registrati.

Così come nell'AWT, distruggere la finestra non è sufficiente per interrompere l'elaborazione. I valori riportati sono costanti definite nell'interfaccia `WindowConstants` che è implementata da `JFrame`.

18.2 I look and feel

L'esempio appena visto usa un look and feel uguale su tutti i sistemi operativi che è chiamato "Metal". È anche detto *Java Look and Feel* (JLF) perché è quello di default. Ogni look and feel è rappresentato da un'istanza di una classe concreta derivata dalla classe astratta `LookAndFeel`. I metodi statici della classe `UIManager` permettono di avere informazioni sui look and feel disponibili e di attivare quello desiderato. Possiamo vedere quelli a nostra disposizione con la seguente classe:

```
import javax.swing.*;

public class LookAndFeel {
  public static void main (String argv[]) {
    UIManager.LookAndFeelInfo lafi[] =
                        UIManager.getInstalledLookAndFeels();
    for (int i = 0; i < lafi.length; i++) {
```

```
        System.out.println ("Nome=" + lafi[i].getName()
                            + ",Classe=" + lafi[i].getClassName());
      }
      System.exit(0);
    }
}
```

Questo programma visualizza i nomi dei look and feel disponibili e il nome delle relative classi che lo implementano. Un tipico risultato è il seguente:

```
Nome=Metal,Classe=javax.swing.plaf.metal.MetalLookAndFeel
Nome=CDE/Motif,Classe=com.sun.java.swing.plaf.motif.MotifLookAndFeel
Nome=GTK+,Classe=com.sun.java.swing.plaf.gtk.GTKLookAndFeel
```

Notare che è necessario invocare il metodo `System.exit(0)` per terminare il programma, in quanto l'invocazione dei metodi di `UIManager` provoca l'attivazione dei thread necessari per la gestione dell'interfaccia grafica.

I seguenti metodi statici di `UIManager` permettono di ricuperare il nome della classe che implementa il look and feel più adatto alle proprie esigenze:

- `public static String getCrossPlatformLookAndFeelClassName()`: restituisce il nome della classe che implementa il look and feel adatto a qualsiasi piattaforma (JLF).
- `public static String getSystemLookAndFeelClassName()`: restituisce il nome della classe che implementa il look and feel analogo a quello del sistema operativo dove si sta eseguendo l'applicazione: se un tale look and feel non risulta disponibile, viene restituito quello adatto a tutte le piattaforme.

Per impostare un look and feel tra quelli disponibili, la classe `UIManager` mette a disposizione i seguenti metodi statici:

- `public static void setLookAndFeel(LookAndFeel nuovoLAF)`: imposta il look and feel di default. Questo metodo può lanciare l'eccezione `UnsupportedLookAndFeelException`, in quanto non è garantito che tutti i look and feel siano implementati su tutti i sistemi operativi.
- `public static void setLookAndFeel(String nomeClasse)`: imposta il look and feel di default fornendo il nome della classe che lo implementa. Questo metodo, oltre a poter lanciare l'eccezione `UnsupportedLookAndFeelException` come il precedente, può lanciare tutte le eccezioni legate al fatto di dover caricare una classe e crearne un'istanza.

Possiamo ora modificare la classe vista precedentemente, `JAutoDialogo`, in modo che possa essere rappresentata con tutti i look and feel disponibili mediante la pressione di un bottone.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
public class JAutoDialogoLAF extends JFrame {
  private JTextField t1;
  private JTextField t2;
  public static void main (String argv[]) {
    JAutoDialogoLAF ist = new JAutoDialogoLAF();
  }
  JAutoDialogoLAF () {
    JPanel pannelloTesto = new JPanel (new GridLayout(0, 2,
                                                      3, 3));
    JPanel pannelloBottoni = new JPanel ();
    setTitle (getClass().getName());

    pannelloTesto.add (new JLabel("Testo 1", JLabel.LEFT));
    pannelloTesto.add (t1 = new JTextField(""));
    pannelloTesto.add (new JLabel("Testo 2", JLabel.LEFT));
    pannelloTesto.add (t2 = new JTextField(""));

    pannelloBottoni.setLayout (new FlowLayout());
    ((JButton) pannelloBottoni.add (new JButton("Scambia")))
                  .addActionListener (new ActionListener() {
      public void actionPerformed(ActionEvent e) {
          String scambio = t1.getText();
          t1.setText (t2.getText());
          t2.setText (scambio);
      }
    });

    ((JButton) pannelloBottoni.add (new JButton("Esci")))
                  .addActionListener (new ActionListener() {
      public void actionPerformed(ActionEvent e) {
          System.exit (0);
      }
    });
    UIManager.LookAndFeelInfo lafi[] =
                        UIManager.getInstalledLookAndFeels();
    for (int i = 0; i < lafi.length; i++) {
      final UIManager.LookAndFeelInfo info = lafi[i];
      ((JButton) pannelloBottoni.add (new
                                JButton(info.getName())))
                  .addActionListener (new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          try {
            UIManager.setLookAndFeel(info.getClassName());
  SwingUtilities.updateComponentTreeUI(JAutoDialogoLAF.this);
            pack();
            repaint();
```

```
            } catch (Exception ex) {
              ((Component)e.getSource()).setEnabled(false);
              System.out.println (ex);
            }
          }
        });
    }

    getContentPane().add (pannelloTesto, "Center");
    getContentPane().add (pannelloBottoni, "South");
    pack();
    setVisible (true);
  }
}
```

Notare che, in questo esempio, dopo l'impostazione di un look and feel, è necessario, prima invocare il metodo statico `SwingUtilities.updateComponentTreeUI(JAutoDialogoLAF.this)` per fare in modo che tutti i componenti già visualizzati a partire dalla `JFrame` principale siano informati della modifica avvenuta, quindi il metodo `pack()` per fare in modo che siano ricalcolate le dimensioni dei componenti e infine il metodo `repaint()` per aggiornare la visualizzazione.

Figura 18.2 I look and feel Metal, CDE/Motif e Windows

Come si vede dalla Figura 18.2, a seconda del look and feel usato, possono cambiare le dimensioni ottimali dei componenti, per cui, se si desidera che un'applicazione possa funzionare con stili differenti, è opportuno usare gli impaginatori per la creazione delle finestre.

18.3 Disegno

Poiché i componenti di Swing sono totalmente scritti in Java, essi sono i soli artefici della loro rappresentazione grafica. Essi devono disegnarsi da soli e, per far questo, non possono far altro che ricorrere ai metodi delle classi dell'AWT, in modo particolare al metodo `paint(Graphics g)` della classe `Component`. Le classi per le applicazioni non dovrebbero quindi ridefinire questo metodo, altrimenti c'è il rischio di incorrere in problemi in fase di visualizzazione. Nella classe `JComponent` è stato così definito il metodo `paintComponent(Graphics g)` che sostituisce `paint(Graphics g)` in modo da non creare problemi.

Le classi che rappresentano una finestra esterna, come `JFrame`, non derivano però da `JComponent` e quindi non possono usare `paintComponent(Graphics g)`. Nei casi in cui si ha bisogno di fare dei disegni sulle finestre principali è opportuno quindi aggiungere un contenitore, per esempio un'istanza di `JPanel`, e usarlo per le rappresentazioni grafiche.

Bisogna fare attenzione che, a differenza del comportamento in AWT, il metodo `repaint()` in Swing non causa la pulizia della finestra, per cui, quando serve, è necessario farla esplicitamente.

Vediamo dunque una possibile implementazione in ambiente Swing della classe `Mouse` vista precedentemente.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JMouse extends JPanel {
   int x1 = -1, x2, y1, y2;

   public static void main (String argv[]) {
      JFrame f = new JFrame();
      JMouse ist = new JMouse();
      f.setBounds (30, 10, 300, 200);
      f.setTitle (ist.getClass().getName());
      f.getContentPane().add(ist);
      f.setVisible(true);
   }
   JMouse () {
      addMouseListener (new MouseAdapter() {
         public void mousePressed(MouseEvent e) {
```

```
            x1 = e.getX();
            y1 = e.getY();
        }
        public void mouseReleased(MouseEvent e) {
            x1 = -1;
        }
    });
    addMouseMotionListener (new MouseMotionAdapter() {
        public void mouseDragged(MouseEvent e) {
            x2 = e.getX();
            y2 = e.getY();
            repaint();
        }
    });
}

public void paintComponent(Graphics g) {
    if (x1 > 0) {
        int x0, y0, larghezza, altezza;
        if (x1 < x2) {
            x0 = x1;
            larghezza = x2 - x1;
        } else {
            x0 = x2;
            larghezza = x1 - x2;
        }
        if (y1 < y2) {
            y0 = y1;
            altezza = y2 - y1;
        } else {
            y0 = y2;
            altezza = y1 - y2;
        }
        g.clearRect(0, 0, getWidth(), getHeight());
        g.drawRect (x0, y0, larghezza, altezza);
    }
}
}
```

La classe `JMouse` deriva ora da `JPanel`, ma, a parte questo, è molto simile alla classe `Mouse` implementata con AWT. Da notare che prima del disegno del rettangolo, viene sempre fatta una pulizia della finestra. Nel metodo `main` è stato creato un oggetto di classe `JFrame` e a esso è stata aggiunta un'istanza di `JMouse`. Il risultato dell'esecuzione è praticamente identico alla versione AWT.

In generale è meglio evitare, quando possibile, la ridefinizione di `paintComponent` e usare i componenti per ottenere effetti grafici, per esempio usare oggetti di tipo `JLabel` per visualizzare testi e/o immagini e oggetti di tipo `Border` per costruire dei bordi. In questo modo le applicazioni risultano più efficienti e più facili da manutenere.

Questo breve tour non ha certo la pretesa di illustrare completamente le potenzialità di Swing. Esso comprende attualmente quasi cinquecento classi che possono essere adattate in modo flessibile a qualsiasi uso. Sono proprio le sue caratteristiche di indipendenza dal sistema operativo, di estensibilità, di completezza e flessibilità che lo rendono ideale per le applicazioni Java attuali e future.

Domande di verifica

1. Cos'è la classe `JComponent`? Da quale classe deriva?
2. Descrivere la classe `JFrame`, il suo metodo `setDefaultCloseOperation` e gli argomenti che possiamo passargli.
3. Cosa si intende per *Java Look and Feel* (JLF)?
4. Quali sono i metodi statici della classe `UIManager` per recuperare il nome della classe che implementa il look and feel più adatto alle esigenze?
5. A quale scopo i componenti Swing utilizzano il metodo `paint` della classe AWT?
6. Quando si debbono realizzare disegni sulle finestre principali, come si opera?

Esercizi

1. Modificare gli esempi riportati nei capitoli precedenti realizzati con AWT in modo che funzionino con Swing.