Walter Savitch

PROGRAMMAZIONE DI BASE E AVANZATA CON JAVA

A cura di Daniela Micucci

Pearson Learning Solution

Codice di accesso a MyLab

- Aula virtuale
- Risorse multimediali
- Test ed esercizi
- Autovalutazione
- Pearson eText

ALWAYS LEARNING

PEARSON

Walter Savitch

con la collaborazione di
Kenrick Mock

PROGRAMMAZIONE DI BASE E AVANZATA CON JAVA

Edizione italiana a cura di
Daniela Micucci
Università degli Studi
di Milano - Bicocca

Curatore per l'edizione italiana: Daniela Micucci

Traduzione: Francesco Fiamberti
Redazione e impaginazione: Carmelo Giarratana
Progetto grafico di copertina: Maurizio Garofalo
Stampa: Tip.Le.Co. – S. Bonico (PC)

Tutti i marchi citati nel testo sono di proprietà dei loro detentori.

978-88-6518-190-4

Printed in Italy

1ª edizione: marzo 2014

Ristampa
01 02 03 04

Anno
15 16 17 18

Indice

Capitolo 6 Array

Capitolo 7 Ricorsione

Capitolo 8 Definire classi e creare oggetti

Capitolo 9 Approfondimenti su classi, oggetti e metodi

Capitolo 14 Stream e I/O da file

Capitolo 15 Strutture dati dinamiche

Capitolo 16 Collezioni, mappe e iteratori

Capitolo 17 Interfacce utente grafiche

Prefazione all'edizione italiana

L'obiettivo del testo *Programmazione di base e avanzata con Java* è di introdurre gli studenti alla programmazione e al *problem solving* utilizzando Java come strumento programmativo. Il testo propone un percorso formativo che include sia i contenuti di un corso di programmazione di base sia un insieme di approfondimenti che supportano lo svolgimento di progetti software complessi. Tale percorso è ottenuto aggiungendo ai dodici capitoli del testo *Programmazione con Java* cinque nuovi capitoli che approfondiscono alcune rilevanti tematiche quali la gestione delle eccezioni, la lettura e la scrittura su file, le strutture dati dinamiche, le collezioni e la realizzazione di interfacce utente grafiche.

Questo testo è il risultato della mia diretta esperienza personale nell'insegnamento della programmazione. Quando mi è stato assegnato per la prima volta un insegnamento di programmazione rivolto a studenti universitari del primo anno mi sono posta il problema di quale testo adottare. L'obiettivo dell'insegnamento era chiaro: gli studenti dovevano imparare a programmare utilizzando sia il paradigma della programmazione strutturata sia quello della programmazione orientata agli oggetti perché tali paradigmi programmativi sarebbero stati, con tutta probabilità, quelli che avrebbero incontrato più di frequente nel corso dei propri studi e, successivamente, in ambito lavorativo. Lo specifico linguaggio di programmazione adottato doveva essere solo il mezzo attraverso cui gli studenti avrebbero messo in pratica le nozioni acquisite. Il linguaggio doveva essere unico e semplice, perché lo scopo era appunto insegnare le basi della programmazione, ma i concetti appresi dovevano essere facilmente trasferibili agli altri linguaggi di programmazione. La scelta è ricaduta su Java: un linguaggio di programmazione semplice e idoneo a essere utilizzato per programmare sia secondo il paradigma strutturato, sia secondo quello a oggetti. Cercando in rete e consultando quanto gentilmente inviato dalle case editrici ho riscontrato un problema piuttosto diffuso: quasi tutti i testi si concentravano sul linguaggio Java e dedicavano minore attenzione alle tecniche di programmazione. Molti di essi, inoltre, introducevano contemporaneamente le basi della programmazione strutturata e il concetto di classe mischiando, a mio avviso, due paradigmi di programmazione.

Quando Pearson mi inviò il testo *Java: An Introduction to Computer Science and Programming* di Walter Savitch non ho avuto esitazione ad adottarlo in accordo con i miei colleghi. Era un testo strutturato secondo la nostra impostazione del corso di programmazione. Col tempo, però, ci siamo resi conto che in un corso erogato al primo anno di studi il libro di testo in inglese costituiva per gli studenti una grossa difficoltà per la comprensione degli argomenti. Da qui l'idea di tradurre il volume in italiano.

Il testo scritto da Walter Savitch si caratterizza per alcune peculiarità che lo contraddistinguono dagli altri. La principale è l'obiettivo che il testo si pone: insegnare le tecniche di programmazione, non semplicemente il linguaggio di programmazione Java. Il testo spazia dalle tecniche per la risoluzione di problemi di programmazione al *debugging*, dalle regole di buon *coding* alla progettazione a oggetti (includendo cenni di UML) mostrando come utilizzare Java quale linguaggio di programmazione per la loro attuazione. La trattazione è corredata da una vasta gamma di esempi completi e chiaramente documentati, evitando l'errore di sfruttare frammenti di codice decontestualizzati. Gli esempi proposti

sono particolarmente utili alla comprensione dei concetti base perché, grazie alla loro immediatezza e semplicità, permettono allo studente di concentrarsi sull'argomento discusso e non sulla comprensione dell'esempio. Il testo propone anche esercizi di programmazione e casi di studio che aiutano lo studente nell'approfondimento della comprensione del problema. L'ordine di presentazione degli argomenti, infine, è molto naturale: viene affrontata prima la programmazione strutturata e successivamente quella a oggetti. Questo è fondamentale, poiché partire dalla programmazione a oggetti può limitare la comprensione della programmazione strutturata come paradigma programmativo indipendente. Pur facendo riferimento a un linguaggio orientato agli oggetti, il testo riesce elegantemente a trattare le tecniche di programmazione strutturata anticipando in modo semplice quei concetti relativi a classi e oggetti necessari alla comprensione degli esempi pratici, senza però complicare la trattazione. Successivamente viene affrontata la programmazione a oggetti per intero, cioè dai concetti fondamentali all'utilizzo dei generici.

La versione italiana non è solo il risultato di una traduzione della sesta edizione di *Java an Introduction to Problem Solving & Programming*, ma è il risultato di un importante processo di revisione il cui obiettivo è stato quello di separare in maniera più netta la parte inerente la programmazione strutturata da quella a oggetti. In questo modo il testo può essere adottato sia in un corso introduttivo alla programmazione strutturata, sia in un corso dedicato alla programmazione a oggetti, e sia in un corso che affronti sia la programmazione strutturata che quella a oggetti. Infine, i cinque nuovi capitoli costituiscono un'ottima base per iniziare ad affrontare problematiche applicative che possono comprendere, ad esempio, l'interfacciamento con i file, la realizzazione di interfacce utente e l'utilizzo di collezioni. Questo fa sì che il testo possa essere adottato in diversi corsi a seconda degli specifici obiettivi formativi e che lo studente possa avere un unico testo di riferimento.

Il raggiungimento di questo obiettivo ha comportato diversi cambiamenti rispetto al testo originale inglese. In particolare, è stato introdotto un nuovo capitolo interamente dedicato ai metodi nella parte iniziale del testo, in modo tale che la programmazione strutturata possa essere appresa e utilizzata anche in contesti non necessariamente legati alla programmazione a oggetti. L'originale Capitolo 7 dedicato agli array è stato suddiviso in due capitoli in modo da trattare separatamente array di tipi primitivi e array di riferimenti. È stato inserito il capitolo dedicato alla ricorsione che chiude la parte riguardante la programmazione strutturata. I capitoli relativi alla definizione delle classi e degli oggetti sono stati rivisitati in modo da presentare i metodi dal punto di vista del paradigma ad oggetti. Il Capitolo 9 include una sezione che approfondisce la modellazione delle associazioni fra classi con UML e la corrispondente implementazione in Java. Ereditarietà e polimorfismo sono trattati in due capitoli che, utilizzando anche contenuti estratti e opportunamente riadattati dal testo *Absolute Java* sempre di Walter Savitch, trattano con maggiore profondità ciascuno dei due argomenti. È stato incluso un capitolo che illustra in maniera dettagliata come realizzare strutture dati dinamiche ed è il risultato di un'opportuna rivisitazione e adattamento di due capitoli, uno dei quali tratto dal testo *Absolute Java*. Sono stati quindi introdotti capitoli più specifici indirizzati a un corso successivo sulla programmazione che riguardano l'I/O su file e la creazione di interfacce utente grafiche. Infine, il capitolo sulle collezioni, tratto da *Absolute Java* e adattato rispetto alla filosofia del testo originale, illustra alcuni tipi di collezioni e le proprietà che le caratterizzano.

Organizzazione

Il testo è organizzato in tre parti principali. I Capitoli da 1 a 7 illustrano la programmazione strutturata. I Capitoli da 8 a 12 sono dedicati alla programmazione orientata agli oggetti e alcune sue applicazioni. I Capitoli da 13 a 17 affrontano alcuni concetti avanzati relativi alla programmazione.

Il Capitolo 1 fornisce una panoramica sintetica sui componenti hardware e software degli elaboratori e sulle tecniche base di progettazione del software, con particolare enfasi verso la programmazione a oggetti.

Il Capitolo 2 introduce le prime nozioni di programmazione che permettono di sviluppare semplici programmi. In particolare sono descritte le variabili, i tipi primitivi e l'input e l'output in Java.

I Capitoli 3 e 4 introducono il flusso di controllo. In particolare il Capitolo 3 illustra le strutture decisionali e le espressioni booleane, mentre il Capitolo 4 illustra i cicli e le tecniche per progettare i cicli.

Il Capitolo 5 introduce i metodi enfatizzando il loro ruolo nella programmazione strutturata.

Il Capitolo 6 presenta gli array di tipi primitivi e illustra alcuni algoritmi notevoli di ordinamento.

Il Capitolo 7 descrive il concetto di ricorsione e illustra il suo utilizzo. Gli esempi sono numerosi e riguardano anche algoritmi notevoli di ordinamento.

I Capitoli 8 e 9 illustrano la programmazione a oggetti, introducendo le classi, gli oggetti, i metodi di istanza e statici e le variabili di istanza e di classe. In particolare, il Capitolo 9 discute approfonditamente l'*information hiding*, ponendo enfasi sulle situazioni in cui può verificarsi una privacy lack e fornendo suggerimenti su come evitarla. Infine, il Capitolo 9 tratta array di tipi riferimento.

I Capitoli 10 e 11 illustrano l'ereditarietà e concetti avanzati ad essa legati. In particolare, il Capitolo 10 presenta le basi dell'ereditarietà, mentre il Capitolo 11 illustra il polimorfismo, le classi astratte e le interfacce.

Il Capitolo 12 presenta l'`ArrayList` come esempio di struttura dati dinamica e conclude presentando i generici in Java.

Il Capitolo 13 illustra la gestione delle eccezioni.

Il Capitolo 14 introduce l'I/O sia sui file di testo e sia su quelli binari.

Il Capitolo 15 presenta alcune strutture dati dinamiche. In particolare il capitolo tratta le liste concatenate, gli insiemi, le tabelle di *hash* e gli alberi. Il Capitolo 16 fornisce una panoramica delle collezioni già presenti nelle librerie standard di Java.

Infine, il Capitolo 17 introduce la programmazione ad eventi e la realizzazione di alcune semplici interfacce utente grafiche.

Ogni capitolo è corredato da esempi completi e casi di studio, che comprendono la descrizione del problema e la costruzione guidata di una soluzione. Ciascun capitolo si conclude con diversi esercizi e proposte di progetto che gli studenti sono invitati a svolgere.

Daniela Micucci
Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano – Bicocca

Pearson Learning Solution

Il volume è corredato da un codice di registrazione che consente l'accesso per diciotto mesi alla piattaforma e-Learning MyLab. Questa nuova piattaforma integra l'attività di studio con un sistema di tutoring, esercitazioni e strumenti per l'autovalutazione. In particolare sono disponibili materiali di supporto e approfondimento tra cui domande di autoverifica (in italiano), video-note e codice sorgente di alcuni programmi.

Le video-note e il codice sorgente sono segnalati nel libro dalla seguente icona:

Prefazione per gli studenti

Questo testo è stato progettato per l'insegnamento del linguaggio di programmazione Java e, cosa ancora più importante, delle tecniche di programmazione di base. Non richiede esperienza di programmazione né particolari conoscenze di matematica, se non alcuni concetti elementari di algebra. Per trarre dal testo il massimo beneficio, sarebbe opportuno avere installato Java sul proprio computer per poter dare pratica attuazione alle tecniche illustrate.

L'Appendice 1 fornisce dei link a siti web per scaricare i compilatori e gli ambienti di programmazione Java. Per i principianti si consiglia inizialmente il JDK Oracle, per il compilatore e i relativi software, e TextPad come semplice editor per il codice. Quando si scarica il JDK, ci si assicuri di ottenere l'ultima versione disponibile.

Se non si ha esperienza di programmazione

Non occorre esperienza di programmazione per utilizzare questo testo. Se si sono avute esperienze con altri linguaggi di programmazione, non bisogna presupporre che Java sia analogo ai linguaggi sperimentati. I linguaggi sono diversi fra loro e le differenze, anche se piccole, possono creare problemi. Si sfoglino almeno i primi sei capitoli del testo e si leggano i box di riepilogo.

Qualora in precedenza si sia programmato in C o in C++, il passaggio a Java può sembrare semplice e privo di complicazioni. Anche se a prima vista può sembrare molto simile al C o al C++, in realtà Java è molto diverso da questi linguaggi ed è necessario essere consapevoli delle differenze.

Supporto all'apprendimento

Ogni capitolo contiene molte caratteristiche che aiutano nella comprensione del materiale presentato:

- la panoramica di apertura comprende un breve sommario dei contenuti, gli obiettivi, i prerequisiti e una descrizione sugli argomenti trattati dal capitolo;
- i box di riepilogo riassumono sinteticamente gli aspetti principali della sintassi Java e altri importanti concetti;
- le FAQ, o "domande frequenti", rispondono alle domande che già altri studenti hanno posto;
- i box che evidenziano le idee importanti che bisognerebbe tenere a mente sempre;
- i box che suggeriscono modi per migliorare le competenze di programmazione;
- i box che identificano alcuni potenziali errori che si possono commettere durante la programmazione;
- il riepilogo a fine capitolo sintetizza i concetti importanti.

Questo testo è anche un manuale di riferimento

Oltre che assolvere alla funzione di libro di testo del corso, questo volume può essere utilizzato come manuale di riferimento. Quando si ha la necessità di verificare un concetto che si è dimenticato o di cui si è sentito parlare ma ancora non si è studiato, basta ricercarlo nell'indice. Molte voci dell'indice analitico riportano l'indicazione della pagina in cui si trova un box di riepilogo. Si vada a quella pagina e si troverà una breve sintesi che fornisce i punti salienti sull'argomento. La stessa procedura si applica per verificare sia i dettagli del linguaggio Java sia quelli delle tecniche di programmazione. In ogni capitolo le sezioni di riepilogo forniscono una sintesi veloce dei principali punti trattati.

Ringraziamenti

Ringraziamo le numerose persone che hanno reso possibile la sesta edizione di questo testo, nonché le persone che hanno contribuito alle prime cinque edizioni. Iniziamo col ringraziare le persone coinvolte nello sviluppo di questa nuova edizione. I commenti e i suggerimenti dei seguenti revisori sono stati di inestimabile valore e ampiamente apprezzati.

Asa Ben-Hur - *Colorado State University*
Joan Boone - *University of North Carolina at Chapel Hill*
Dennis Brylow - *Temple University*
Billie Goldstein - *Temple University*
Hellen H. Hu - *Westminster College*
Tammy VanDeGrift - *University of Portland*

Molti altri revisori hanno letto le prime bozze delle edizioni precedenti del libro. Questa nuova edizione continua a beneficiare dei loro consigli. Grazie ancora una volta a:

Gerald Baumgartner – *Louisiana State University*
Jim Buffenbarger – *Idaho State University*
Robert P. Burton – *Brigham Young University*
Mary Elaine Califf – *Illinois State University*
Steve Cater – *Kettering University*
Martin Chelten – *Moorpark Community College*
Ashraful A. Chowdhury – *Georgia Perimeter College*
Ping-Chu Chu – *Fayetteville State University*
Michael Clancy – *University of California, Berkeley*
Tom Cortina – *State University of New York at Stony Brook*
Prasun Dewan – *University of North Carolina*
Laird Dornan – *Sun Microsystems, Inc.*
H. E. Dunsmore – *Purdue University, Lafayette*
Adel Elmaghraby – *University of Louisville*
Ed Gellenbeck – *Central Washington University*
Adrian German – *Indiana University*

Gobi Gopinath – *Suffolk County Community College*
Le Gruenwald – *University of Oklahoma*
Gopal Gupta – *University of Texas, Dallas*
Ricci Heishman – *North Virginia Community College*
Robert Herrmann – *Sun Microsystems, Inc., Java Soft*
Chris Hoffmann – *University of Massachusetts, Amherst*
Robert Holloway – *University of Wisconsin, Madison*
Charles Hoot – *Oklahoma City University*
Lily Hou – *Carnegie Mellon University*
Richard A. Johnson – Missouri State University
Rob Kelly – *State University of New York at Stony Brook*
Michele Kleckner – *Elon College*
Stan Kwasny – *Washington University*
Anthony Larrain – *Depaul University*
Mike Litman – *Western Illinois University*
Y. Annie Liu – *State University of New York at Stony Brook*
Michael Long – *California State University*
Blayne Mayfield – *Oklahoma State University*
Drew McDermott – *Yale University*
Gerald H. Meyer – *LaGuardia Community College*
John Motil – *California State University, Northridge*
Michael Olan – *Stockton State*
Richard Ord – *University of California, San Diego*
James Roberts – *Carnegie Mellon University*
Alan Saleski – *Loyola University Chicago*
Dolly Samson – *Hawaii Pacific University*
Nan C. Schaller – *Rochester Institute of Technology*
Arijit Sengupta – *Raj Sion College of Business, Wright State University*
Ryan Shoemaker – *Sun Microsystems, Inc.*
Liuba Shrira – *Brandeis University*
Ken Slonneger – *University of Iowa*
Donald E. Smith – *Rutgers University*
Peter Spoerri – *Fairfield University*
Howard Strambing – *Boston College*
Navabi Tadayon – *Arizona State University*
Boyd Trolinger – *Butte College*
Tom Van Drunen – *Wheaton College*
Subramanian Vijayarangam – *University of Massachusetts, Lowell*
Stephen F.Weiss – *University of North Carolina, Chapel Hill*
Richard Whitehouse – *Arizona State University*
Michael Young – *University of Oregon*

Infine, ma non per questo ultimo, ringraziamo i numerosi studenti della University of California San Diego (UCSD) che ci hanno aiutato a correggere le versioni preliminari di questo testo così come lo sono stati i docenti che hanno utilizzato in aula queste versioni

preliminari. In particolare, rivolgiamo uno speciale ringraziamento a Carole McNamee della California State University, Sacramento e a Paul Kube della UCSD. I commenti di questi studenti, i feedback dettagliati e la verifica in aula delle precedenti versioni di questo testo sono stati di inestimabile aiuto nella definizione di questo testo.

W.S., K.M.

Guida alla lettura

Riepilogo

Sintetizza concetti importanti e la sintassi Java.

La sintassi dell'istruzione `for`

Sintassi

```
for (inizializzazione; espressione_booleana; aggiornamento)
    corpo
```

Il *corpo* del ciclo può essere un'istruzione singola oppure, come accade più spesso, un'istruzione composta che consiste di un elenco di istruzioni racchiuse tra parentesi graffe { }. Si noti che i tre elementi tra le parentesi tonde sono separati da due punti e virgola, non tre.

Esempio

```
for (prossimo = 0; prossimo <= 10; prossimo = prossimo + 2) {
    somma = somma + prossimo;
    System.out.println("La somma ora corrisponde a " + somma);
}
```

Da ricordare

Evidenzia alcuni concetti importanti che dovrebbero essere ricordati.

Differenze tra tipi primitivi e tipi classe

Un metodo non può modificare il valore di un argomento di tipo primitivo che gli viene passato. Inoltre, un metodo non può sostituire un oggetto che riceve come argomento con un altro oggetto. D'altro canto, un metodo può modificare i valori delle variabili di istanza di un argomento di tipo classe.

TIP

Fornisce utili suggerimenti sulla programmazione.

Le variabili di istanza dovrebbero essere `private`

Tutte le variabili di istanza di una classe dovrebbero essere dichiarate come `private`. In questo modo si costringe un programmatore che debba usare la classe ad accedere alla variabile di istanza solo attraverso i metodi della classe. Questo permette alla classe di controllare tutte le attività di lettura e scrittura dei valori delle variabili di istanza. L'esempio che segue mostra perché è importante rendere private le variabili di istanza.

PITFALL
Evidenzia possibili errori che dovrebbero essere evitati.

Usare l'operatore = invece di = = per verificare l'uguaglianza

Il simbolo = è l'operatore di assegnamento. Sebbene questo simbolo abbia il significato di uguale in matematica, non presenta lo stesso significato in Java. Se si scrive `if (x = y)` invece di `if (x == y)` per verificare se x e y sono uguali, il compilatore restituisce un messaggio di errore di sintassi.

FAQ
Fornisce risposte a tipiche domande.

FAQ Perché i numeri in virgola mobile sono chiamati così?

I numeri in virgola mobile sono chiamati in questo modo perché l'utilizzo della notazione e permette di far "fluttuare" (dall'inglese *floating*) la virgola in una nuova posizione spostando l'esponente. Per esempio, si può spostare la virgola in `0.000483` dopo il 4 esprimendo questo numero come `4.83e-4`.

Listato
Mostra programmi completi con esempi di output.

LISTATO 8.13 Programma di dimostrazione della classe `Oracolo`.

```
public class OracoloDemo {

    public static void main(String[] args) {
        Oracolo delphi = new Oracolo();
        delphi.parla();
    }
}
```

Esempio di output

```
Sono l'oracolo. Rispondero' a qualsiasi domanda che digiterai su una riga.
Qual e' la tua domanda?
Che ore sono?
Hmm, ho bisogno di aiuto su questo.
Scrivimi una riga di aiuto.
Guardami e dovresti trovar risposta
Grazie. Mi ha aiutato molto
Hai posto la domanda:
 Che ore sono?
Ora ecco la mia risposta:
 La risposta e' nel tuo cuore.
Vuoi pormi un'altra domanda?
si
Qual e' la tua domanda?
Qual e' il senso della vita?
Hmm, ho bisogno di aiuto su questo.
Scrivimi una riga di aiuto.
Chiedi al rivenditore d'auto
```

Caso di studio

Dato un problema, identifica e descrive l'algoritmo risolutivo e la relativa codifica in Java.

CASO DI STUDIO
FORMATTAZIONE DELL'OUTPUT

Quando si usa una variabile di tipo `double` per memorizzare, per esempio, una certa somma di denaro, ci si aspetta che il programma visualizzi l'importo in un formato appropriato. Tuttavia, è probabile che si ottenga un output simile al seguente:

```
Il costo, IVA inclusa, e' di E19.98123576432
```

Ovviamente si preferirebbe avere un output come il seguente:

```
Il costo, IVA inclusa, e' di E19.98
```

Sebbene come separatore dei centesimi si usi la virgola, per rappresentare somme di denaro viene di seguito utilizzato il punto poiché si fa uso di numeri in virgola mobile. Si ricorda che i numeri in virgola mobile utilizzano come separatore per le cifre decimali il punto (.) e non la virgola.

In questo caso di studio si definisce la classe `FormattaEuro` contenente i due metodi statici `scrivi` e `scriviRiga` che possono essere utilizzati per produrre questo tipo di output ben formattato. Per esempio, se la somma di denaro è memorizzata nella variabile `double somma`, potremmo scrivere il seguente codice per ottenere l'output desiderato:

```
System.out.print("Il costo, IVA inclusa, e' di ");
FormattaEuro.scriviRiga(somma);
```

Si noti che i metodi `scrivi` e `scriviRiga` devono indicare che il denaro è espresso in Euro e devono sempre mostrare esattamente due cifre dopo la virgola. Quindi, questi metodi devono mostrare E2.10 e non E2.1.

Esempio di programmazione

Fornisce la soluzione di uno specifico problema in termini di programma Java.

ESEMPIO DI PROGRAMMAZIONE
SPESE FOLLI

Si immagini che Paola abbia vinto un buono omaggio da 100 Euro a una gara. Il buono deve essere speso in un certo negozio, ma non si possono comprare più di tre prodotti. Il computer del negozio tiene traccia di quanto rimane da spendere e del numero di prodotti acquistati. Ogni volta che Paola sceglie un prodotto, il computer dice se può essere acquistato. Sebbene in questo esempio si considerino cifre piccole, si vuole scrivere un programma in cui sia il numero di Euro a disposizione, sia il numero di prodotti acquistabili possano essere modificati facilmente.

Chiaramente questo è un processo ripetitivo: Paola continua ad acquistare finché ha a disposizione ancora soldi e finché ha acquistato meno di tre prodotti. L'espressione

Capitolo 1

Introduzione ai computer e a Java

OBIETTIVI

- Fornire una panoramica sintetica delle caratteristiche hardware e software dei computer.
- Fornire una panoramica del linguaggio di programmazione Java.
- Descrivere le tecniche base di progettazione del software, con particolare enfasi verso la programmazione a oggetti.

Questo capitolo presenta una breve panoramica sull'hardware e sul software dei computer. La maggior parte degli argomenti trattati è relativa alla programmazione in qualsiasi linguaggio, non solo alla programmazione in Java. La parte relativa al software include la descrizione di una metodologia per la progettazione di programmi nota come programmazione a oggetti (*object-oriented programming*). Il capitolo introduce, infine, il linguaggio di programmazione Java e discute la struttura di un semplice programma.

Prerequisiti

In questo primo capitolo non si presuppone alcuna esperienza di programmazione, ma si prevede la possibilità di accedere a un computer. Per poter apprendere meglio quanto descritto in questo capitolo e nel resto del testo, si dovrebbe avere a disposizione un computer con il linguaggio Java installato, in modo da poter mettere in pratica quello che si sta apprendendo. L'Appendice 1 descrive come ottenere una copia gratuita del linguaggio Java, insieme a degli utili editor.

1.1 Concetti di base sui computer

Un **computer** è un insieme di componenti hardware e software. Il termine **hardware** indica la macchina fisica. Un insieme di istruzioni che un computer deve eseguire è detto **programma**. Il termine **software** indica genericamente tutti i possibili tipi di programmi utilizzati per fornire istruzioni a un computer. In questo testo verrà trattato il software, ma, per comprenderlo al meglio, è utile conoscere alcuni aspetti base dell'hardware.

1.1.1 Hardware e memoria

La maggior parte dei computer disponibili oggigiorno è costituita dagli stessi componenti di base, solitamente configurati allo stesso modo. Tutti i computer possiedono dispositivi di ingresso (*input device*), come una tastiera e un mouse, dispositivi di uscita (*output device*), come un monitor e una stampante, e altri componenti, solitamente ospitati all'interno di un box (*cabinet* o *case*), che si occupano di archiviare i dati ed effettuare le computazioni.

La **CPU** (*Central Processing Unit*, unità centrale di elaborazione), più semplicemente chiamata **processore**, è il dispositivo interno che esegue le istruzioni di un programma. Il processore è in grado di eseguire solo istruzioni molto semplici, come spostare numeri o altri dati presenti in memoria e compiere alcune operazioni aritmetiche di base, come somme e sottrazioni.

La **memoria** di un computer conserva i **dati** che il computer deve elaborare e i risultati dei calcoli intermedi effettuati. Esistono fondamentalmente due tipi di memoria: la memoria principale e la memoria ausiliaria. La **memoria principale** conserva il programma che attualmente è in esecuzione e gran parte dei dati che il programma stesso sta utilizzando. Le informazioni immagazzinate nella memoria principale sono **volatili**, cioè sono cancellate quando il computer viene spento. Al contrario, i dati contenuti nella **memoria ausiliaria**, detta anche **memoria secondaria**, persistono anche a computer spento. Tutti i vari tipi di dischi, come gli hard disk, le flash drive, i CD (*Compact Disc*) e i DVD (*Digital Versatile Disc*), sono memorie ausiliarie.

Per esempio, in un PC (*Personal Computer*) avente 1 gigabyte di RAM e 200 gigabyte di hard disk, la **RAM** (*Random Access Memory*, memoria ad accesso casuale) è la memoria principale, mentre l'hard disk è la principale forma di memoria ausiliaria, anche se non l'unica. Un byte è una quantità di memoria: un gigabyte di RAM è costituito approssimativamente da un miliardo di byte di memoria.

La memoria principale del computer è costituita da un lungo elenco di byte numerati, ognuno con un proprio **indirizzo**. Un **byte** è la più piccola unità di memoria indirizzabile. Un singolo dato, per esempio un numero o un carattere della tastiera, può essere memorizzato in uno di questi byte. Per recuperare successivamente tale dato, il computer utilizza l'indirizzo del byte in cui era stato memorizzato.

Per convenzione, un byte contiene otto cifre (*digit*), ciascuna delle quali può assumere il valore 0 oppure il valore 1. Ciascuna di queste cifre è detta **cifra binaria** (*binary digit*) o, più comunemente, **bit**. Un byte contiene quindi otto bit di memoria. Sia la memoria principale, sia la memoria secondaria sono misurate in byte.

Dati di vario tipo, come numeri, lettere e stringhe di caratteri, sono codificati come serie di valori 0 e 1 e immagazzinati nella memoria del computer. Un byte è sufficientemente grande per contenere un singolo carattere della tastiera. Questo è uno dei motivi per cui la memoria del computer è suddivisa in byte di otto bit. Tuttavia, per memorizzare una stringa di caratteri oppure un numero di valore elevato, è necessario impiegare più di un singolo byte. Quando il computer ha necessità di memorizzare un dato che non può essere contenuto in un singolo byte, utilizza più byte adiacenti. Questi byte vengono quindi considerati come un'unica, e più grande, **area di memoria**, dove l'indirizzo del primo byte viene utilizzato come indirizzo dell'intera area di memoria. La Figura 1.1 mostra come una tipica memoria principale possa essere suddivisa in aree di memoria. Gli indirizzi di queste aree non sono fissati dall'hardware, ma dipendono dal programma che sta utilizzando la memoria.

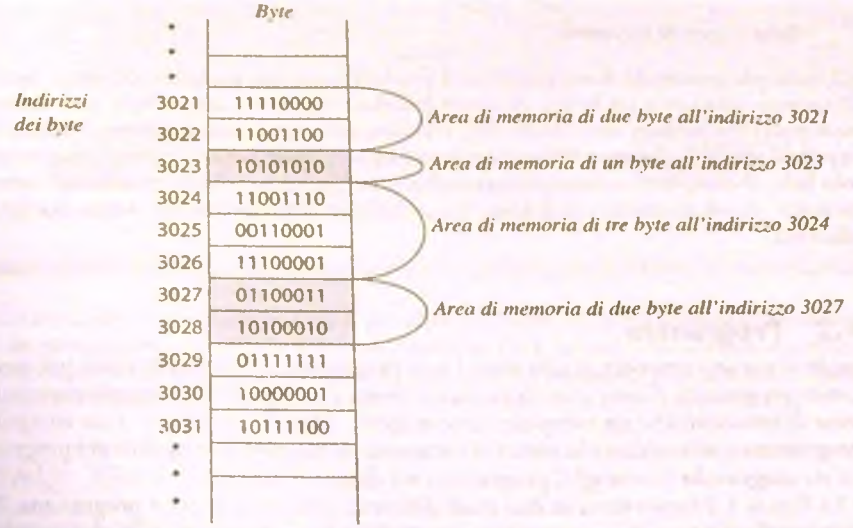


Figura 1.1 La memoria principale.

Come precedentemente detto, la memoria principale contiene il programma che è attualmente in esecuzione e gran parte dei dati che sta utilizzando. La memoria ausiliaria viene usata per conservare i dati in modo più o meno permanente. Anche la memoria ausiliaria è suddivisa in byte, ma questi sono raggruppati in unità più grandi, dette **file**. Un file può contenere pressoché qualsiasi tipo di dato, per esempio un programma, un testo, un elenco di numeri o una figura, ciascuno in forma codificata. Per esempio, quando si scrive un programma Java, lo si salva in un file, che tipicamente risiede nell'hard disk del computer. Quando il programma verrà usato, per esempio per modificarlo, il contenuto del file verrà copiato dalla memoria ausiliaria alla memoria principale.

Ciascun file ha un nome e gruppi di file possono essere organizzati in **directory** o **cartelle** (*folder*). I termini cartella e directory sono sinonimi: alcuni sistemi informatici utilizzano il primo nome, altri il secondo.

FAQ Perché solo 0 e 1?

I computer utilizzano i valori 0 e 1 perché è semplice costruire un dispositivo elettronico che ha solo due stati stabili. Tuttavia, mentre si sta programmando, normalmente non ci si deve concentrare sul fatto che i dati siano codificati come sequenze di 0 e 1. Infatti, è possibile programmare come se il computer immagazzinasse nella memoria direttamente numeri, lettere o stringhe di caratteri.

Non c'è alcun motivo particolare per cui gli stati siano stati chiamati 0 e 1; si sarebbero potuti chiamare con altri nomi, come A e B, oppure vero e falso. L'aspetto importante è che il dispositivo fisico sottostante abbia due stati stabili, come on e off oppure alta e bassa tensione elettrica. Chiamare questi due stati 0 e 1 è semplicemente una convenzione, ma è quella che viene quasi universalmente adottata.

Byte e aree di memoria

La memoria principale di un computer è suddivisa in unità numerate chiamate byte. Il numero associato a un byte è chiamato indirizzo del byte. Ciascun byte può contenere otto cifre binarie, dette anche bit, ciascuna delle quali può assumere il valore 0 oppure il valore 1. Per memorizzare un dato troppo grande per essere contenuto in un solo byte, il computer utilizza più byte adiacenti. Questi byte sono considerati come un'unica area di memoria più grande, il cui indirizzo è l'indirizzo del primo dei byte adiacenti.

1.1.2 Programmi

Quando si usa un computer, si utilizzano i suoi programmi. Gli editor di testo, per esempio, sono programmi. Come precedentemente detto, un programma è semplicemente un insieme di istruzioni che un computer deve eseguire. Quando si fornisce a un computer un programma e alcuni dati e si indica al computer di eseguire le istruzioni del programma, si sta **eseguendo** (*running*) il programma sui dati.

La Figura 1.2 rappresenta in due modi differenti l'esecuzione di un programma. Per considerare il primo modo, occorre ignorare il riquadro tratteggiato. Ciò che rimane è quello che accade quando si esegue un programma. In quest'ottica, il computer ha due tipi di input. Il primo è il programma stesso, che contiene le istruzioni che il computer dovrà eseguire. Il secondo input sono i dati per il programma, cioè le informazioni che il programma dovrà elaborare. Per esempio, nel caso di un programma di correzione ortografica, i dati saranno il testo da controllare. Per come sono concepiti i computer, sia i dati sia il programma stesso sono input. L'output, cioè quello che viene prodotto, è costituito dai risultati prodotti dal computer in seguito all'esecuzione delle istruzioni del programma sui dati forniti in input. Se il programma deve controllare l'ortografia di un testo, l'output potrebbe essere un elenco di parole che contengono errori ortografici.

Questo primo modo di rappresentare l'esecuzione di un programma raffigura quello che accade, ma non costituisce il modo in cui ci si immagina l'esecuzione di un programma. Un altro modo di rappresentare l'esecuzione di un programma considera i dati come l'unico input del programma. Secondo quest'ottica, il computer e il programma sono considerati come una sola unità. Il riquadro tratteggiato della Figura 1.2 circonda l'unità composta dal programma e dal computer. In quest'ottica, i dati sono considerati l'input del programma, mentre i risultati il suo output. Sebbene sia chiaro che il computer è necessario, esso viene visto come qualcosa che assiste il programma. Le persone che scrivono programmi, cioè i **programmatori**, quando progettano un programma, ritengono più utile il secondo modo di rappresentare l'esecuzione.

Il computer possiede molti più programmi di quanto si possa immaginare. La maggior parte di quello che si considera essere "il computer" è in realtà un programma, cioè un software, non un componente hardware. Ogni volta che si accende un computer, si sta già eseguendo un programma. Questo programma è il **sistema operativo**, una specie di programma supervisore che controlla il funzionamento del computer nella sua interezza. Se si vuole eseguire un programma, si indica al sistema operativo che cosa si intende fare. Di conseguenza, il sistema operativo recupera il programma e lo avvia. Il programma che si esegue può essere un editor di testo, un browser per navigare nel Web o qualche pro-

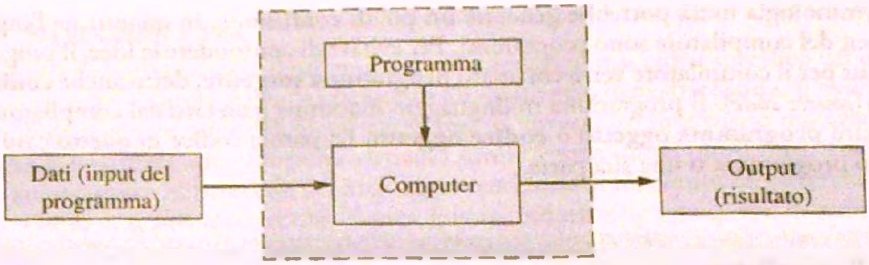


Figura 1.2 Esecuzione di un programma.

gramma scritto nel linguaggio Java. È possibile richiedere al sistema operativo l'esecuzione di un programma utilizzando il mouse e facendo click su un'icona oppure selezionando un elemento di un menu oppure, ancora, digitando un comando. Alcuni sistemi operativi comuni sono Microsoft Windows, Apple (Macintosh) Mac OS, Linux e UNIX.

FAQ Che cos'è il software?

La parola software significa semplicemente programma. Un'azienda di software è quindi un'azienda che produce programmi, mentre il software del computer è semplicemente la collezione dei programmi presenti sul computer.

1.1.3 Linguaggi di programmazione, compilatori e interpreti

La maggior parte dei linguaggi di programmazione moderni è stata progettata per essere facile da utilizzare e comprendere; per questo motivo, tali linguaggi vengono definiti **linguaggi di alto livello**. Java è un linguaggio di alto livello, così come la maggior parte dei linguaggi di programmazione più diffusi, come Visual Basic, C++, C#, Python o Ruby. Ma, sfortunatamente, l'hardware dei computer non è in grado di comprendere direttamente i linguaggi di alto livello. Prima che un programma scritto in un linguaggio di alto livello possa essere eseguito da un computer, questo deve essere tradotto in un altro linguaggio, comprensibile per il computer.

Il linguaggio che il computer è in grado di comprendere è chiamato **linguaggio macchina**. Il **linguaggio assembly** è una rappresentazione simbolica del linguaggio macchina, più semplice da interpretare da parte di una persona. Di conseguenza, il linguaggio assembly corrisponde in parte al linguaggio macchina, ma necessita comunque di alcune piccole trasformazioni prima di poter essere interpretato dal computer. Questi tipi di linguaggi sono definiti **linguaggi di basso livello**. La traduzione di un programma da un linguaggio di alto livello, come Java, a un linguaggio di basso livello viene effettuata interamente o in parte da un altro programma. Per alcuni linguaggi di alto livello, questa traduzione viene effettuata da un programma detto **compilatore**. Il compilatore elabora il programma scritto in linguaggio di alto livello in modo che possa essere eseguito sul computer. Questa operazione è detta **compilazione** del programma. Una volta compilato, è possibile eseguire il programma risultante quante volte si vuole, senza doverlo ricompilare.

La terminologia usata potrebbe generare un po' di confusione, in quanto sia l'input, sia l'output del compilatore sono programmi. Per evitare di confondere le idee, il programma di input per il compilatore verrà chiamato **programma sorgente**, detto anche **codice sorgente** (*source code*). Il programma in linguaggio macchina generato dal compilatore verrà chiamato **programma oggetto** o **codice oggetto**. La parola codice in questo caso indica l'intero programma o una sua parte.

Il compilatore

Un compilatore è un software che traduce un programma scritto in un linguaggio di alto livello (come Java), in un programma in un linguaggio più semplice che il computer è in grado di comprendere direttamente.

Alcuni linguaggi di alto livello non vengono tradotti da compilatori, ma da un altro tipo di programmi, detti **interpreti**. Come un compilatore, un interprete traduce le istruzioni di un programma da un linguaggio di alto livello a un linguaggio di basso livello. Ma, a differenza di un compilatore, un interprete esegue ogni singola porzione di codice subito dopo averla tradotta, invece di tradurre l'intero programma in una sola passata. Utilizzare un interprete significa che mentre si esegue un programma, la traduzione si alterna all'esecuzione. Inoltre, la traduzione viene ripetuta a ogni esecuzione del programma. La compilazione viene invece svolta una volta sola e il risultante codice oggetto può essere eseguito un numero indefinito di volte, senza impiegare nuovamente il compilatore. La conseguenza è che un programma compilato viene solitamente eseguito più velocemente di un corrispondente programma interpretato.

L'interprete

Un interprete è un programma che alterna la traduzione all'esecuzione delle istruzioni di un programma scritto in un linguaggio di alto livello.

Uno svantaggio dei processi appena descritti per la traduzione di programmi scritti in linguaggi di alto livello consiste nel fatto che occorre un compilatore o un interprete differente per ogni tipo di linguaggio o di computer che si sta utilizzando. Se si volesse eseguire lo stesso codice sorgente su tre diversi tipi di computer, occorrerebbe utilizzare tre diversi compilatori o interpreti. Se poi qualcuno sviluppasse un tipo di computer completamente nuovo, un gruppo di sviluppatori dovrebbe scrivere un nuovo compilatore o interprete per quel computer. Questo costituisce chiaramente un problema, poiché compilatori e interpreti sono programmi di grandi dimensioni, difficili da sviluppare e quindi costosi. Nonostante questo costo, molti compilatori e interpreti dei linguaggi di alto livello funzionano in questo modo. Java utilizza un approccio leggermente diverso e più versatile, che combina compilazione e interpretazione. Tale approccio è di seguito descritto.

1.1.4 Bytecode Java

Il compilatore Java non traduce il programma nel linguaggio macchina specifico del computer su cui è stato compilato, ma lo traduce in un linguaggio detto **bytecode**. Il bytecode non è un linguaggio macchina di alcun computer. È, invece, un linguaggio macchina di una **macchina virtuale** (un computer virtuale) simile a tutti quelli più diffusi. Tradurre un programma scritto in bytecode nel linguaggio macchina di un computer effettivo è abbastanza semplice. Il programma che effettua questa traduzione è una specie di interprete chiamato **macchina virtuale Java** (*Java Virtual Machine* – **JVM**). La macchina virtuale Java ha quindi il compito di tradurre ed eseguire il bytecode Java.

Per eseguire un programma Java su un computer occorre procedere come segue: in primo luogo si utilizza il compilatore per tradurre il programma sorgente Java nel bytecode corrispondente. Successivamente si utilizza la macchina virtuale Java del computer specifico per tradurre ciascuna istruzione bytecode in linguaggio macchina e per eseguire le istruzioni in linguaggio macchina. L'intero processo è illustrato in Figura 1.3.

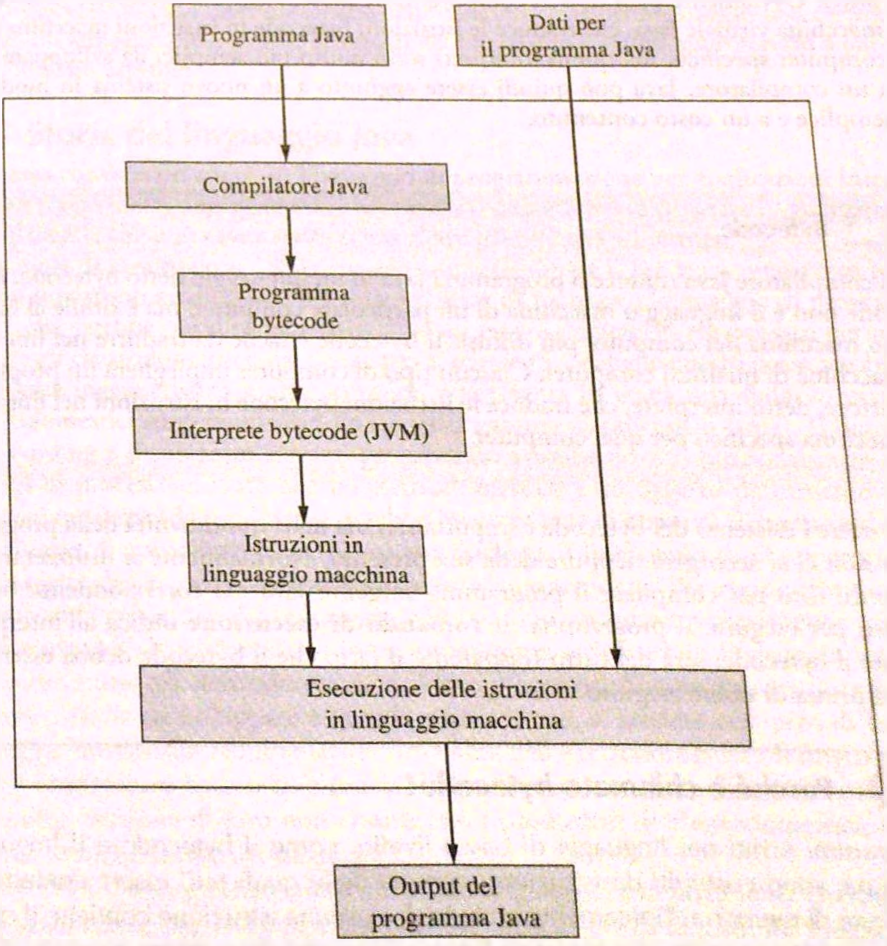


Figura 1.3 Compilare ed eseguire un programma Java.

In base a quanto descritto, sembra che il bytecode Java si limiti ad aggiungere un nuovo passo al processo. Perché, quindi, non scrivere compilatori che traducano direttamente il codice da Java al linguaggio macchina del sistema specifico? Questo può essere fatto ed è quello che viene fatto per molti altri linguaggi di programmazione. Inoltre, questo approccio permetterebbe di generare programmi oggetto più rapidi. Tuttavia, il bytecode Java dona a Java un vantaggio importante: la portabilità. Dopo aver compilato il programma Java in bytecode, è possibile eseguire quel programma su qualsiasi computer dotato di una macchina virtuale Java, senza bisogno di ricompilarlo. Questo vuol dire che è possibile inviare il bytecode a un altro computer attraverso Internet e aspettarsi che questo venga eseguito, indipendentemente dal sistema operativo che viene utilizzato sul computer remoto. Questo è uno dei motivi per cui Java è particolarmente utile per lo sviluppo di applicazioni Internet.

La portabilità presenta un ulteriore vantaggio. Quando un'azienda produce un nuovo tipo di computer, gli sviluppatori Java non sono costretti a creare un nuovo compilatore. Il compilatore Java, infatti, genera bytecode che può venir eseguito su tutti i computer. Ovviamente ogni tipo di computer deve avere il proprio interprete di bytecode, la macchina virtuale Java, che traduce le istruzioni bytecode in istruzioni macchina per quel computer specifico, ma questi interpreti sono molto più semplici da sviluppare rispetto a un compilatore. Java può quindi essere aggiunto a un nuovo sistema in modo molto semplice e a un costo contenuto.

Bytecode

Il compilatore Java traduce il programma Java in un linguaggio detto bytecode. Il bytecode non è il linguaggio macchina di un particolare computer, ma è simile al linguaggio macchina dei computer più diffusi. Il bytecode è facile da tradurre nel linguaggio macchina di qualsiasi computer. Ciascun tipo di computer impiegherà un proprio traduttore, detto interprete, che traduce le istruzioni bytecode in istruzioni nel linguaggio macchina specifico per quel computer.

Conoscere l'esistenza del bytecode è importante, ma nella quotidianità della programmazione non ci si accorgerà neppure della sua presenza. Normalmente si utilizzeranno due comandi: uno per compilare il programma sorgente Java nel corrispondente bytecode e l'altro per eseguire il programma. Il **comando di esecuzione** indica all'interprete di eseguire il bytecode; sarà del tutto trasparente il fatto che il bytecode debba essere interpretato prima di essere eseguito.

FAQ Perché è chiamato bytecode?

I programmi scritti nei linguaggi di basso livello, come il bytecode e il linguaggio macchina, sono costituiti da istruzioni, ognuna delle quali può essere contenuta in pochi byte di memoria. Tipicamente un byte di ciascuna istruzione contiene il codice operativo, detto opcode, che specifica l'operazione che deve essere eseguita. Il concetto di opcode delle dimensioni di un byte ha dato origine al termine bytecode.

1.1.5 Class Loader

Solo raramente un programma Java è contenuto in un unico file di codice sorgente. Al contrario, tipicamente consiste di diverse porzioni, dette **classi**. Si parlerà in dettaglio delle classi più avanti; per ora è sufficiente pensare alle classi come a porzioni di codice. Le classi sono spesso scritte da autori diversi e ciascuna classe viene compilata separatamente e tradotta quindi in un diverso frammento di bytecode. Per eseguire il programma, i bytecode delle diverse classi devono essere collegati fra loro. L'operazione di collegamento viene svolta da un programma detto **class loader** (letteralmente "caricatore delle classi"). L'operazione di collegamento è tipicamente svolta in maniera automatica. In altri linguaggi di programmazione, il programma corrispondente al *class loader* Java prende il nome di *linker*.

1.2 Un assaggio di Java

In questa sezione verranno descritte alcune delle caratteristiche del linguaggio Java e verrà esaminato un semplice programma. I dettagli del linguaggio saranno approfonditi a partire dal prossimo capitolo.

1.2.1 Storia del linguaggio Java

Java è spesso considerato come un linguaggio di programmazione per applicazioni Internet. Questo testo (così come numerosi sviluppatori) considera Java come un linguaggio di utilizzo generale, che può essere usato senza alcun riferimento a Internet.

La storia di Java risale al 1991, quando James Gosling e il suo team presso Sun Microsystem iniziarono a sviluppare la prima versione di un nuovo linguaggio di programmazione, che sarebbe poi diventato Java. Questo nuovo linguaggio era pensato per programmare gli elettrodomestici, dai tostapane ai televisori. Sebbene questo possa sembrare un problema ingegneristico molto semplice, rappresenta in realtà una sfida, in quanto gli elettrodomestici sono controllati da un vasta gamma di processori (*chip*). Il linguaggio che Gosling e il suo team stavano progettando avrebbe dovuto funzionare con tutti questi tipi di processore. Dato che un elettrodomestico è un oggetto tipicamente poco costoso, nessuna azienda produttrice avrebbe investito grandi quantità di tempo e denaro nello sviluppo di complicati compilatori per tradurre il linguaggio degli apparecchi in un linguaggio che il processore avrebbe potuto comprendere. Per rispondere a questi due problemi, i progettisti del linguaggio svilupparono un software che avrebbe tradotto il programma dell'elettrodomestico in un programma scritto in un linguaggio intermedio, comune a tutti gli elettrodomestici e ai loro processori. Successivamente, un piccolo programma, facile da sviluppare e quindi poco costoso, si sarebbe occupato di tradurre il linguaggio intermedio nel linguaggio macchina dell'elettrodomestico. Il linguaggio intermedio fu chiamato bytecode. Tuttavia, l'idea di produrre elettrodomestici utilizzando questa prima versione di Java non entusiasmò i produttori di elettrodomestici, ma non comportò neanche la fine del linguaggio.

Nel 1994 Gosling si rese conto che il suo linguaggio, ora finalmente chiamato Java, sarebbe stato ideale per sviluppare un browser Web in grado di eseguire programmi via Internet. Il browser Web venne prodotto da Patrick Naughton e Jonathan Payne presso Sun Microsystems. Originariamente chiamato WebRunner e successivamente HotJava

questo browser non è più supportato, ma dette inizio alla interconnessione tra Java e Internet. Nell'autunno del 1995, Netscape Communications Corporation decise di attribuire alla nuova versione del suo browser Web la capacità di eseguire programmi Java. Altri sviluppatori seguirono questo esempio e svilupparono software in grado di eseguire programmi Java.

FAQ Perché proprio il nome Java?

Gli autori dei linguaggi sono soliti scegliere un nome per le loro creature nello stesso modo in cui i genitori scelgono un nome per i propri figli. Ognuno semplicemente sceglie il nome che più gli piace. Il nome originale del linguaggio Java era Oak. I progettisti del linguaggio scoprirono, però, che esisteva già un altro linguaggio di programmazione con questo nome e perciò dovettero trovarne un altro e scelsero Java. Diverse sono le spiegazioni sulle origini del nome Java. L'origine più accreditata indica che il nome venne deciso durante una lunga e tediosa riunione in cui i partecipanti bevvero molto caffè; da qui il nome Java che, in inglese, è infatti sinonimo di caffè.

1.2.2 Applicazioni e applet

Ci sono due tipi di programmi Java: le **applicazioni** e le **applet**; l'unica differenza consiste nel fatto che le applicazioni sono concepite per esser eseguite localmente su un computer, mentre le applet sono pensate per essere inviate via Internet ed eseguite in un computer remoto.

Una volta imparato a sviluppare uno di questi due tipi di programmi, non si hanno problemi a sviluppare anche l'altro. Questo testo fornisce solo informazioni per sviluppare applicazioni.

1.2.3 Il primo programma Java

Il Listato 1.1 mostra il primo programma Java del testo. Sotto il programma è riportato un esempio dell'output che può venir generato sullo schermo quando qualcuno, un **utente**, esegue e poi interagisce con il programma. Nell'output, il testo scritto dall'utente è colorato in blu. Tuttavia, nel momento in cui verrà eseguito il programma, sia il testo mostrato dal programma, sia quello scritto dall'utente avranno lo stesso colore.

Chi utilizza il programma potrebbe essere la stessa persona che lo ha scritto. Per esempio, spesso uno studente ricopre sia il ruolo di programmatore, sia quello di utente; ma, nella quotidianità, programmatore e utente sono di solito due persone differenti. Questo testo insegna a programmare.

Una delle prime cose che occorre imparare è che non ci si può aspettare che un utente sappia esattamente come comportarsi con un programma. Per questo motivo, il programma deve fornire all'utente istruzioni comprensibili, come è stato fatto nel programma d'esempio.

Lo scopo di questa sezione è solo quello di dare un'idea del linguaggio, mostrando una breve e informale descrizione del semplice programma illustrato nel Listato 1.1. *Pertanto è perfettamente normale che alcuni dettagli del programma non siano completamente*

chiari a una prima lettura. Questa è solo un'anticipazione di quello che verrà spiegato nei prossimi capitoli. In particolare, il Capitolo 2 fornirà spiegazioni dettagliate delle funzionalità di Java utilizzate in questo semplice esempio.

La prima riga:

```
import java.util.Scanner;
```

indica al compilatore che questo programma usa la classe `Scanner`. Per il momento è possibile pensare a una classe come a un frammento di codice che è possibile usare in un programma. Questa classe è definita nel package `java.util` (abbreviazione per "Java utility"). Un **package** è una libreria di classi che sono state definite in precedenza.

Le righe rimanenti definiscono la classe `PrimoProgramma` che si estende dalla prima parentesi graffa aperta, `{`, all'ultima parentesi graffa chiusa, `}`:

```
public class PrimoProgramma {
    ...
}
```

LISTATO 1.1 Un semplice programma Java.

MyLab

```
import java.util.Scanner;
```
← Carica la classe `Scanner` dal package (libreria) `java.util`

```
public class PrimoProgramma {
```
← Nome della classe, a scelta

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

        System.out.println("Ciao!");
        System.out.println("Eseguo la somma di due numeri.");
        System.out.println("Digita entrambi i numeri sulla stessa riga:");

        int n1, n2;
```
← Indica che `n1` e `n2` sono variabili che contengono interi

```
        Scanner tastiera = new Scanner(System.in);
```
← Predispone il programma affinché possa leggere l'input dalla tastiera

```
        n1 = tastiera.nextInt();
        n2 = tastiera.nextInt();

        System.out.println("Ecco la somma dei due numeri:");
        System.out.println(n1 + n2);
    }
}
```

← Invia l'output allo schermo

← Legge un numero intero dalla tastiera

Esempio di output

```
Ciao!
Eseguo la somma di due numeri.
Digita entrambi i numeri sulla stessa riga:
12 30
Ecco la somma dei due numeri:
42
```

Tra le parentesi graffe ci sono una o più porzioni di codice, denominate **metodi**. Ogni applicazione Java ha un metodo chiamato `main` e spesso altri metodi. La definizione del metodo `main` si estende da una seconda parentesi graffa aperta, fino alla corrispondente parentesi graffa chiusa.

```
public static void main(String[] args) {
    ...
}
```

Le parole `public static void` sono necessarie, ma resteranno per ora un mistero; saranno introdotte nel Capitolo 5 e descritte in dettaglio nei Capitoli 8 e 9.

Ogni **istruzione** (*statement*) all'interno di un metodo definisce un compito; l'insieme delle istruzioni all'interno di un metodo costituisce il **corpo** del metodo. Le prime tre istruzioni del metodo `main` sono le prime azioni svolte da questo programma:

```
System.out.println("Ciao!");
System.out.println("Eseguo la somma di due numeri.");
System.out.println("Digita entrambi i numeri sulla stessa riga:");
```

Ciascuna di queste istruzioni inizia con `System.out.println` e mostra su una stessa riga ciò che si trova all'interno di una coppia di parentesi tonde e specificato fra doppi apici.

Per esempio, l'istruzione:

```
System.out.println("Ciao!");
```

presenta sullo schermo la riga:

```
Ciao!
```

Per il momento si può considerare che `System.out.println` sia un modo per dire al computer "Mostra sullo schermo quanto è contenuto tra le parentesi". Tuttavia, è utile introdurre un po' di terminologia. Per eseguire le azioni, i programmi Java utilizzano **oggetti software**, detti più semplicemente **oggetti**. Le azioni sono definite da metodi. `System.out` è un oggetto utilizzato per inviare un output sullo schermo; `println` è il metodo che esegue questa azione per l'oggetto `System.out`. In altre parole, `println` invia allo schermo ciò che è specificato all'interno della coppia di parentesi tonde. L'elemento o gli elementi tra parentesi sono detti **argomenti** e forniscono al metodo l'informazione di cui necessita per eseguire l'azione.

In ciascuna di queste tre istruzioni, l'argomento del metodo `println` è una stringa di caratteri racchiusa tra doppi apici. Questo argomento è ciò che `println` scrive sullo schermo.

Un oggetto esegue un'azione quando viene **invocato** (o **chiamato**) uno dei suoi metodi. In un programma Java si ottiene un'**invocazione di metodo** (o **chiamata di metodo**) scrivendo il nome dell'oggetto, seguito da un punto (chiamato, in gergo, **dot**), seguito dal nome del metodo e infine da una coppia di parentesi tonde. All'interno delle parentesi potrebbero essere specificati uno o più argomenti.

La riga seguente del programma nel Listato 1.1,

```
int n1, n2;
```

dice che `n1` e `n2` sono i nomi di due variabili. Una **variabile** può memorizzare dati. Il termine `int` indica che i dati devono essere di tipo intero, cioè numeri interi; `int` è un esempio di **tipo di dato** (*data type*). Un tipo di dato (o semplicemente tipo) specifica l'insieme dei valori possibili e le operazioni definite per questi valori. I valori di un particolare tipo di dato sono immagazzinati in memoria nel medesimo formato.
La riga successiva:

```
Scanner tastiera = new Scanner(System.in);
```

abilita il programma ad accettare, o **leggere**, i dati che l'utente inserisce tramite la tastiera. Questa riga di codice sarà spiegata in dettaglio nel Capitolo 2[1].
La riga successiva:

```
n1 = tastiera.nextInt();
```

legge il numero che è stato digitato alla tastiera e lo memorizza nella variabile `n1`. La riga successiva è pressoché simile, tranne per il fatto che il numero digitato alla tastiera viene memorizzato nella variabile `n2`. Perciò, se l'utente inserisce i numeri `12` e `30`, come indicato nell'output d'esempio, la variabile `n1` conterrà il numero `12` e la variabile `n2` conterrà il numero `30`.
Infine, le istruzioni

```
System.out.println("Ecco la somma dei due numeri:");
System.out.println(n1 + n2);
```

mostrano rispettivamente una frase esplicativa e la somma dei numeri memorizzati nelle variabili `n1` e `n2`. Si sottolinea che la seconda riga contiene l'**espressione** `n1 + n2` e non una stringa di caratteri racchiusa fra apici (o "quotati"). Questa espressione computa la somma dei numeri memorizzati nelle variabili `n1` e `n2`. Quando un'istruzione di output come questa contiene un numero o un'espressione che produce come risultato un numero, questo viene mostrato sullo schermo. Quindi nell'output d'esempio mostrato nel Listato 1.1, queste due istruzioni producono le righe:

```
Ecco la somma dei due numeri:
42
```

Si noti che ciascuna invocazione di `println` mostra l'output su una riga distinta.

Rimane da spiegare il significato del punto e virgola specificato alla fine solo di alcune righe. Il punto e virgola ha il ruolo di carattere di terminazione, come il punto in una frase italiana. Un punto e virgola termina quindi un'istruzione.

Chiaramente Java adotta regole precise, che indicano come si devono scrivere le varie parti di un programma. Queste regole costituiscono la **grammatica** del linguaggio, esattamente come le regole dell'italiano. Le regole grammaticali di un linguaggio, sia esso un linguaggio naturale o un linguaggio di programmazione, formano la **sintassi** del linguaggio.

[1] Come si vedrà nel prossimo capitolo, è possibile utilizzare altri nomi al posto di `tastiera`.

Invocare un metodo

Un programma Java utilizza oggetti per eseguire azioni che sono definite da metodi. Un oggetto esegue un'azione quando si effettua un'invocazione, o una chiamata, a uno dei suoi metodi. Di solito l'invocazione viene specificata in un programma scrivendo il nome dell'oggetto, seguito da un punto, detto *dot* in inglese, seguito dal nome del metodo e infine da una coppia di parentesi che può contenere degli argomenti. Gli argomenti sono informazioni per il metodo.

Esempi

```
System.out.println("Ciao!");
n1 = tastiera.nextInt();
```

Nel primo esempio, `System.out` è l'oggetto, `println` è il metodo e `"Ciao!"` è l'argomento. Quando un metodo richiede più argomenti, questi devono essere separati da una virgola. Un'invocazione di metodo è tipicamente seguita da un punto e virgola.

Nel secondo esempio, `tastiera` è l'oggetto e `nextInt` è il metodo. Questo metodo non riceve alcun argomento, tuttavia le parentesi sono obbligatorie.

FAQ Perché occorre utilizzare `import` per l'input ma non per l'output?

Il programma presentato nel Listato 1.1 usa l'istruzione

```
import java.util.Scanner;
```

per abilitare l'input da tastiera, come, per esempio, in

```
n1 = tastiera.nextInt();
```

Perché non occorre un `import` simile anche per abilitare l'output sullo schermo come, per esempio, in

```
System.out.println("Ciao!");
```

La risposta è semplice. In un programma Java, il package che include le definizioni e il codice per l'output su schermo viene importato automaticamente.

1.2.4 Scrivere, compilare ed eseguire programmi Java

Un programma Java è suddiviso in piccole porzioni chiamate classi. Ciascun programma può essere costituito da un numero indefinito di classi. Anche se per il programma rappresentato nel Listato 1.1 è stata scritta una sola classe, `PrimoProgramma`, di fatto il programma utilizza anche altre due classi: `System` e `Scanner`. Queste due classi sono fornite da Java.

È possibile scrivere una classe Java utilizzando un semplice editor di testi. Per esempio, si può utilizzare il Blocco note in Windows o TextEdit in Mac OS X. Di norma, ciascuna definizione di classe viene scritta su un file distinto. Inoltre, il nome del file deve coincidere con il nome della classe, con l'aggiunta dell'estensione `.java`. Per esempio, la classe `PrimoProgramma` deve essere definita nel file `PrimoProgramma.java`.

Prima di poter eseguire un programma Java, occorre tradurre le sue classi in un linguaggio che il computer sia in grado di comprendere. Come abbiamo già detto, questo processo di traduzione è detto compilazione. Non occorre compilare classi come `Scanner`, in quanto sono fornite da Java stesso. Normalmente occorre compilare solo le classi che si creano esplicitamente.

MyLab

Video 1.1 Compilare un programma Java

Per compilare una classe Java utilizzando il sistema Java fornito gratuitamente da Sun Microsystem per Windows, Linux o Solaris, occorre utilizzare il commando `javac` seguito dal nome del file contenente la classe. Per esempio, per compilare la classe `MiaClasse` contenuta nel file `MiaClasse.java`, occorre eseguire il seguente commando:

```
javac MiaClasse.java
```

Quindi, per compilare la classe nel Listato 1.1 occorre eseguire il comando:

```
javac PrimoProgramma.java
```

Quando si compila una classe Java, la sua versione tradotta (il suo bytecode) è posta in un file il cui nome coincide con quello della classe, ma con l'estensione `.class`. Dunque, se si compila il file `PrimoProgramma.java`, il bytecode risultante verrà salvato nel file `PrimoProgramma.class`.

Sebbene un programma Java possa includere un numero indefinito di classi, la sola classe da eseguire è quella che rappresenta l'intero programma. Questa classe conterrà un metodo `main` che inizia con una formulazione identica o molto simile alla seguente:

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

Questi termini si trovano spesso (ma non sempre) all'inizio del file. I termini chiave obbligatori sono `public static void main`. La parte rimanente della riga potrebbe impiegare termini leggermente differenti.

L'esecuzione di un programma Java si ottiene digitando il commando `java` seguito dal nome della classe che rappresenta l'intero programma. Per esempio, per eseguire il programma rappresentato nel Listato 1.1, si deve digitare il seguente commando:

```
java PrimoProgramma
```

Si noti che si deve scrivere il nome della classe, in questo caso `PrimoProgramma`, e non il nome del file che contiene il suo bytecode (`PrimoProgramma.class`). In pratica si omette l'estensione `.class`. Quando si esegue un programma Java, in effetti si richiama l'interprete Java chiedendogli di eseguire la versione compilata del programma.

Il modo più semplice per scrivere, compilare ed eseguire un programma Java è quello di utilizzare un ambiente di sviluppo integrato (*Integrated Development Environment* - **IDE**). Un IDE comprende un editor di testo dotato di un menu dei comandi necessari per compilare ed eseguire programmi Java. Ambienti IDE come BlueJ, Eclipse e NetBeans sono disponibili gratuitamente per Windows, Mac OS e altri sistemi. L'Appendice 1 fornisce tutte le informazioni necessarie per ottenere una copia gratuita di questi ambienti di sviluppo.

FAQ **Ho provato a eseguire il programma d'esempio del Listato 1.1, ma dopo aver inserito i due numeri non è successo nulla. Perché?**

Quando si digitano dei dati sulla tastiera, l'utente vede i caratteri che inserisce, ma il programma Java non li legge finché non viene premuto il tasto Invio (*Enter* o *Return*). Occorre sempre premere Invio dopo aver digitato una riga di dati sulla tastiera.

1.3 Concetti di base di programmazione

La programmazione è un processo creativo. Questo testo non può indicare esattamente come scrivere un programma che svolga le attività desiderate dal programmatore. Il testo presenta solo alcune tecniche utili. In questo paragrafo verranno trattate alcune di queste tecniche di base, che tra l'altro possono essere adottate in qualsiasi linguaggio di programmazione e non solo in Java.

1.3.1 Programmazione a oggetti

Java è un linguaggio di **programmazione a oggetti** (*Object-Oriented Programming* – **OOP**). Che cos'è, quindi, la OOP? Il mondo che ci circonda è costituito da oggetti: persone, automobili, costruzioni, alberi, negozi, navi, zucche e re. Ciascuno di questi oggetti ha la capacità di svolgere azioni e ciascuna di queste azioni può influenzare altri oggetti del mondo. La OOP è una metodologia di programmazione che considera il programma come costituito da oggetti (o istanze) che possono agire da soli e anche interagire fra loro. In un programma, un oggetto software può rappresentare un oggetto del mondo reale o una sua astrazione.

Si consideri, per esempio, un programma che simula un incrocio stradale, con lo scopo di analizzare il flusso del traffico. Questo programma utilizzerà tanti oggetti, ognuno dei quali rappresenta una singola automobile che entra nell'incrocio, e probabilmente anche altri oggetti che rappresentano ciascuna corsia della strada, i semafori e così via. Le interazioni fra questi oggetti permettono di giungere ad alcune conclusioni riguardanti la progettazione dell'incrocio.

La programmazione a oggetti ha una propria terminologia. Un oggetto ha diverse caratteristiche, dette **attributi**. Per esempio, un oggetto automobile potrebbe avere attributi come il nome, la velocità corrente e il livello di carburante. I valori degli attributi di un oggetto costituiscono lo **stato** dell'oggetto stesso. Le azioni che un oggetto può effettuare sono dette **comportamenti** (*behavior*). Come si è visto in precedenza, ciascun comportamento è definito in una porzione di codice Java, detta metodo.

Gli oggetti di uno stesso tipo condividono lo stesso tipo di dato. Una **classe** definisce il tipo di dato di un oggetto; è una sorta di "stampo" (*blueprint*) che consente di creare (in gergo "istanziare") oggetti. Il tipo di dato di un oggetto è dato dal nome della sua classe. Per esempio, in un programma di simulazione del traffico, tutte le automobili simulate possono essere create dalla stessa classe, probabilmente chiamata `Automobile`; quindi il loro tipo è `Automobile`.

Tutti gli oggetti di una classe hanno gli stessi attributi e lo stesso comportamento. Perciò, in un programma di simulazione, tutte le automobili hanno lo stesso compor-

tamento, per esempio si spostano avanti o indietro. Questo non vuol dire che tutte le automobili simulate siano identiche. Sebbene abbiano gli stessi attributi, ognuna di esse può trovarsi in uno stato differente. Cioè ogni loro attributo può assumere diversi valori per ognuna di esse. Quindi potremmo osservare tre automobili prodotte da altrettanti costruttori differenti e che viaggiano a tre diverse velocità. Tutto questo risulterà più chiaro quando si inizierà a scrivere le prime classi Java.

Come si vedrà, la stessa metodologia a oggetti può essere applicata a qualsiasi tipo di programma e non si limita ai programmi di simulazione. La programmazione a oggetti non è nuova, ma il suo utilizzo al di fuori dei programmi di simulazione non si è diffuso fino ai primi anni novanta.

Oggetti, metodi e classi

Un oggetto è un costrutto programmativo che possiede dati, detti attributi, e che può effettuare certe operazioni, note come comportamenti dell'oggetto. Una classe definisce un tipo di oggetto; rappresenta una sorta di stampo per definire gli oggetti. Tutti gli oggetti della stessa classe hanno gli stessi tipi di dato e gli stessi comportamenti. Quando un programma viene avviato, ciascun oggetto può operare da solo o interagire con altri oggetti per raggiungere gli obiettivi del programma. Le azioni effettuate dagli oggetti sono definite dai metodi.

FAQ Che cosa succede negli altri linguaggi di programmazione?

Chi sta iniziando proprio con Java lo studio dei linguaggi di programmazione può anche evitare di leggere questa risposta. Chi invece conosce altri linguaggi di programmazione, scoprirà che questo paragrafo può essere utile per comprendere il funzionamento degli oggetti, in termini di costrutti già noti. Se si conosce un altro linguaggio orientato agli oggetti, come C++, C#, Python o Ruby, si avrà già un'idea dei concetti di classe, oggetto e metodo. Essi sono fondamentalmente analoghi in tutti i linguaggi di programmazione orientati agli oggetti, sebbene altri linguaggi potrebbero utilizzare altri termini per descrivere gli stessi concetti, come, per esempio, nel caso dei metodi. Se invece si ha familiarità con un linguaggio di programmazione meno recente che non utilizza oggetti e classi, è possibile pensare agli oggetti in termini di altri costrutti programmativi. Per esempio, se si conoscono i concetti di variabile e funzione (o procedura), gli oggetti possono essere considerati come variabili che possiedono vari dati e anche alcune funzioni (o procedure). I metodi, infatti, corrispondono alle funzioni o alle procedure dei linguaggi di programmazione meno recenti.

La programmazione orientata agli oggetti utilizza classi e oggetti, ma li usa in modo diverso da come avviene in altri linguaggi meno recenti. Occorre seguire alcuni principi di progettazione. I tre principi fondamentali della progettazione orientata agli oggetti sono incapsulamento, polimorfismo ed ereditarietà.

Il termine **incapsulamento** fa pensare all'atto di mettere qualcosa in una capsula, di impacchettare le cose. Questa intuizione è fondamentalmente corretta. La caratteristica principale dell'incapsulamento, tuttavia, non consiste semplicemente nel racchiudere gli oggetti in una capsula, ma nel rendere visibile solamente una parte di questa capsula. Quando si produce un frammento di software, occorre descriverlo in modo che altri programmatori possano utilizzarlo e sorvolare su tutti i dettagli del suo funzionamento. Si sottolinea che l'incapsulamento nasconde i dettagli del contenuto della capsula. Per questo motivo, l'incapsulamento è spesso chiamato **information hiding** (letteralmente "nascondere le informazioni").

I principi dell'incapsulamento si applicano a tutta la programmazione in generale, non solamente alla programmazione orientata agli oggetti. Tuttavia, i linguaggi orientati agli oggetti non si limitano a permettere al programmatore di utilizzare questi principi; lo spingono a osservarli. Il Capitolo 8 spiegherà in dettaglio il concetto di incapsulamento.

Il termine **polimorfismo** deriva dal greco e vuol dire "molte forme". L'idea alla base del polimorfismo consiste nel permettere a una stessa istruzione di un programma di avere significati differenti in contesti differenti. Il polimorfismo ricorre comunemente nella lingua italiana e il suo utilizzo in un linguaggio di programmazione lo rende più simile a un linguaggio umano. Per esempio, la frase "vai pure agli allenamenti" assume un significato diverso a seconda della persona con cui si sta parlando. Per uno lo sport potrebbe essere la pallavolo, per un altro il calcio.

Il polimorfismo ricorre, inoltre, nelle attività quotidiane[2]. Si immagini una persona che richiama, fischiando, i propri animali per la cena. Il suo cane accorre, i suoi canarini volano e il suo pesce nuota verso il bordo della vaschetta. Tutti rispondono a modo loro. Il richiamo per la cena non indica agli animali il modo con cui arrivare a cena, ma chiede solo di arrivarci. In maniera analoga, alla pressione del pulsante "ON", un computer, un iPod o uno spazzolino elettrico, rispondono tutti in modo differente, ma appropriato.

In un linguaggio di programmazione come Java, il termine polimorfismo indica che il nome di un metodo, usato come istruzione, può causare azioni differenti a seconda degli oggetti che svolgono l'azione. Per esempio, un metodo `mostraStato` potrebbe mostrare lo stato di un oggetto. Tuttavia il tipo di attributi, il loro numero e la modalità con cui vengono presentati dipende dal tipo di oggetto che esegue l'azione. Il Capitolo 11 spiegherà in dettaglio il concetto di polimorfismo.

L'**ereditarietà** è un modo di organizzare le classi. Grazie all'ereditarietà è possibile definire una sola volta gli attributi e i comportamenti comuni e applicarli poi a un intero insieme di classi. Se si definisce una classe generica, si può utilizzare l'ereditarietà a posteriori per definire classi specializzate che aggiungono o perfezionano alcuni dettagli della classe generica.

Un esempio di un insieme di classi di questo tipo è rappresentato nella Figura 1.4, dove si può notare che a ogni livello la classificazione si specializza sempre più. La classe `Veicolo` presenta certe proprietà comuni, per esempio il fatto di possedere ruote. Le classi `Automobile`, `Motociclo` e `Autobus` "ereditano" questa proprietà di possedere ruote, ma aggiungono nuove proprietà o restrizioni. Per esempio, un oggetto `Automobile` avrà quattro ruote, un oggetto `Motociclo` ne avrà due e un oggetto `Autobus` ne avrà almeno quattro.

[2] Gli esempi sono tratti da Carl Alphone, "Pedagogy and Practice of Design Patterns and Objects First: A One-Act Play," ACM SIGPLAN Notices 39, 5 (May 2004), 7–14.

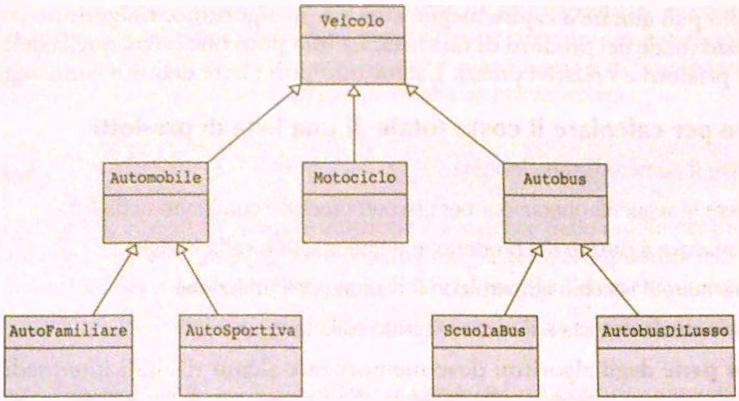


Figura 1.4 Gerarchia di ereditarietà.

L'ereditarietà permette al programmatore di evitare di ripetere le stesse istruzioni per ogni singola classe. Per esempio, tutto ciò che vale per ciascun oggetto di tipo `Veicolo`, come il fatto di possedere ruote, viene descritto una volta sola e viene ereditato dalle classi `Automobile`, `Motociclo` e `Autobus`.

Se non esistesse l'ereditarietà, ciascuna delle classi `Automobile`, `Motociclo`, `Autobus`, `Scuolabus` e così via, dovrebbero ripetere tutte le descrizioni, come il fatto di possedere ruote. Il Capitolo 10 spiegherà in dettaglio il concetto di ereditarietà.

Programmazione orientata agli oggetti

La programmazione orientata agli oggetti (OOP), è una metodologia di programmazione che definisce oggetti i cui comportamenti e le cui interazioni permettono di raggiungere un certo risultato. La OOP adotta i seguenti principi di progettazione: incapsulamento, polimorfismo ed ereditarietà.

1.3.2 Algoritmi

Gli oggetti possiedono comportamenti che sono definiti da metodi. Il programmatore ha il compito di definire tali metodi specificando le istruzioni necessarie per compiere le azioni che devono essere eseguite. La parte più difficile nel definire un metodo non consiste nell'esprimere la soluzione in un linguaggio di programmazione, ma nell'individuare una strategia risolutiva per l'azione. Questa strategia viene spesso espressa con il termine algoritmo. Un **algoritmo** è un insieme di direttive atte a risolvere un problema. Tali direttive, per poter essere considerate un algoritmo, devono essere espresse in un modo così completo e preciso che chiunque possa seguirle senza dover introdurre ulteriori dettagli o compiere scelte che non sono state specificate. Un algoritmo può essere scritto in italiano, in un linguaggio di programmazione come Java o in **pseudocodice**. Quest'ultimo è una combinazione di termini in linguaggio naturale e di termini appartenenti al linguaggio di programmazione.

Un esempio può aiutare a capire meglio cosa sia un algoritmo. L'algoritmo proposto calcola il prezzo totale dei prodotti di una lista. La lista potrebbe essere quella della spesa, che specifica i prodotti e i relativi prezzi. L'algoritmo può essere definito come segue.

MyLab

Video 1.2 Scrivere un algoritmo

Algoritmo per calcolare il costo totale di una lista di prodotti

1. Scrivere il numero 0 sulla lavagna.
2. Ripetere le seguenti operazioni per ciascun prodotto contenuto nella lista.
   - Sommare il prezzo del prodotto al numero scritto sulla lavagna.
   - Sostituire il vecchio numero con il risultato dell'addizione.
3. Indicare che la risposta è il numero scritto sulla lavagna.

La maggior parte degli algoritmi deve memorizzare alcuni risultati intermedi; questo algoritmo utilizza, per esempio, una lavagna. Se l'algoritmo fosse scritto in Java e venisse eseguito da un computer, i risultati intermedi verrebbero immagazzinati nella memoria del computer.

Algoritmo

Un algoritmo è un insieme di direttive atte a risolvere un problema. Per essere considerate un algoritmo, le direttive devono essere espresse in modo completo e preciso.

Pseudocodice

Lo pseudocodice è una combinazione di linguaggio naturale (l'italiano) e linguaggio Java. Quando si usa lo pseudocodice, si scrivono le diverse porzioni dell'algoritmo nel linguaggio che risulta più comodo. Una parte potrebbe essere più semplice da descrivere in italiano, un'altra parte potrebbe invece essere più semplice da descrivere in Java.

1.3.3 Collaudo e debugging

Il modo migliore per scrivere un programma corretto consiste nel progettare con attenzione gli oggetti e gli algoritmi che realizzano le azioni dei metodi. Successivamente si passa a trascrivere accuratamente il tutto in un linguaggio di programmazione come Java. In altre parole, il modo migliore per eliminare gli errori è quello di non farne. Tuttavia, è naturale che un programma contenga errori residui. Una volta terminata la scrittura di un programma, è necessario collaudarlo (o testarlo, dal termine inglese *testing*) per verificare se si comporta correttamente. Gli eventuali errori rilevati vanno naturalmente corretti.

Un errore in un programma viene comunemente chiamato **bug** (letteralmente "baco") o difetto (*fault*). Per questo motivo il processo di rimozione degli errori dal programma è chiamato **debugging**. Ci sono tre tipi di errori possibili: errori di sintassi, errori a run-time (cioè durante l'esecuzione) ed errori logici.

Un **errore di sintassi** è un errore grammaticale nel programma. Quando si programma, occorre seguire le rigide regole grammaticali del linguaggio. Se si viola una di

queste regole, per esempio omettendo un carattere di punteggiatura, si commette un errore di sintassi. Il compilatore individua questi errori e presenta un messaggio d'errore che ne indica la tipologia. Tuttavia, il compilatore è solo in grado di fare ipotesi sul tipo di errore. Per questo motivo la sua diagnosi potrebbe essere scorretta.

Sintassi

La sintassi di un linguaggio di programmazione è l'insieme delle regole del linguaggio, che indicano come scrivere un programma o una sua parte. Il compilatore individua gli errori di sintassi del programma e cerca di indicare al meglio l'elemento errato.

Un errore che viene individuato durante l'esecuzione del programma viene detto **errore a run-time**. Questi errori producono un messaggio d'errore. Per esempio, viene generato un errore a run-time quando si divide inavvertitamente un numero per 0 (zero). Il messaggio d'errore potrebbe non essere semplice da comprendere, ma permette di capire che qualcosa è andato storto. Alle volte, invece, il messaggio d'errore potrebbe indicare con esattezza la causa dell'errore.

Se l'algoritmo su cui si basa un programma contiene un errore o se si scrive un'istruzione corretta secondo la sintassi di Java, ma non dal punto di vista logico, il programma potrebbe essere compilato ed eseguito senza alcun problema. In pratica è stato scritto un programma Java valido, ma non il programma che si intendeva veramente scrivere. Dunque il programma verrà eseguito, ma l'output sarà errato. In questo caso, il programma contiene un **errore logico**. Per esempio, si commette un errore logico se, al posto dell'operatore +, viene scritto l'operatore -. Alle volte un errore logico porta a un errore a run-time che produce un messaggio d'errore. Tuttavia, molto spesso un errore logico non causa alcun messaggio d'errore; proprio per questo motivo, gli errori logici sono i più difficili da individuare.

Errori nascosti

Il fatto che il programma venga compilato ed eseguito senza alcun errore e che magari produca risultati plausibili, non significa necessariamente che sia corretto. È sempre buona norma eseguire il programma con alcuni dati di test che facciano generare un risultato predicibile. Per fare questo, si scelgano dati per i quali sia possibile computare il risultato corretto, per esempio con carta e penna. Anche questo collaudo non garantisce la certezza assoluta che il programma sia corretto, ma più test si effettuano e più si avrà sicurezza della correttezza del programma.

MyLab

Video 1.
Rilevare
un erron
nascosto

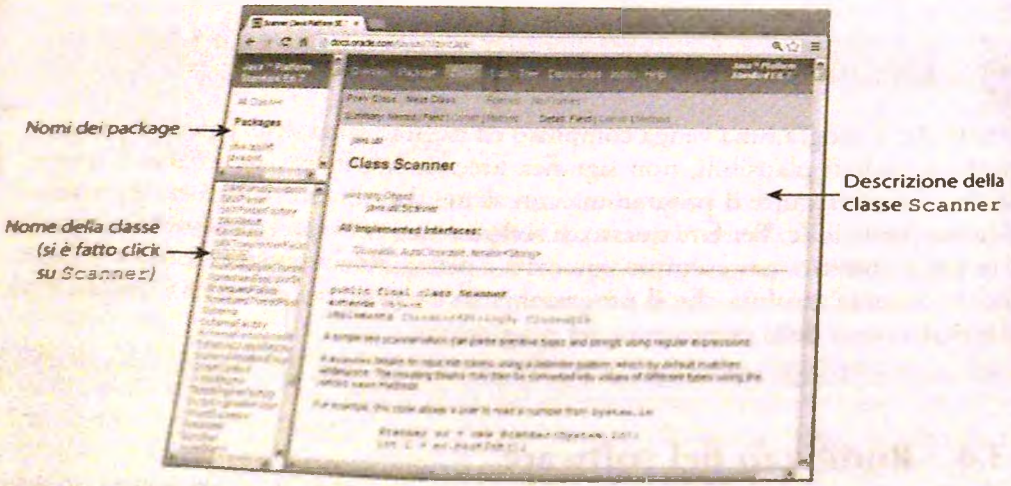
1.3.4 Riutilizzo del software

Quando si inizia a programmare, si ha l'impressione che tutto debba essere creato da zero. Tuttavia il software non viene prodotto in questo modo. La maggior parte dei programmi, infatti, contiene componenti già esistenti. Il fatto di riutilizzare questi componenti permette di risparmiare tempo e denaro. Inoltre, questi componenti, essendo stati utiliz-

zati più e più volte, sono stati ben collaudati e quindi sono certamente più affidabili del software scritto partendo da zero.

Per esempio, un programma per la simulazione del traffico potrebbe includere un nuovo oggetto `strada` per modellare un nuovo tipo di strada, ma probabilmente modellerà le automobili utilizzando una classe `Automobile` che era già stata progettata per un altro programma. Affinché le classi che si scrivono siano facilmente riutilizzabili, occorre progettarle in modo appropriato. Occorre specificare esattamente il modo in cui gli oggetti di una certa classe interagiscono con gli altri oggetti. Questo è il principio dell'incapsulamento, introdotto qualche pagina fa. Tuttavia l'incapsulamento non è l'unico principio che occorre seguire. Le classi devono essere progettate in modo che gli oggetti siano sufficientemente generici e non specifici per un solo programma. Per esempio, anche se il programma richiede che tutte le automobili simulate si spostino solo in avanti, occorre comunque includere la retromarcia nella classe `Automobile`, perché qualche altro programma di simulazione potrebbe prevedere per le automobili la possibilità di tornare indietro. Si riprenderà l'argomento del riutilizzo del software dopo aver presentato qualche dettaglio in più sul linguaggio Java, quando si avranno a disposizione più esempi su cui lavorare.

Oltre a riutilizzare le classi personalmente scritte, spesso si riutilizzano le classi fornite da Java. Per esempio, sono già state riutilizzate le classi standard `Scanner` e `System`, per le operazioni di input e output. Java viene fornito con una collezione di classi nota come **Java Class Library** (letteralmente "libreria delle classi Java"), chiamata anche **Java Application Programming Interface** o **API**. Le classi di questa collezione sono organizzate in package; la classe `Scanner`, per esempio, è contenuta nel package `java.util`. Di volta in volta verranno menzionate o utilizzate classi che fanno parte della Java Class Library. Inoltre, occorre imparare a consultare la documentazione fornita per la Java Class Library sul sito Web di Oracle. Al momento, la documentazione è all'indirizzo http://docs.oracle.com/javase/7/docs/api/. La Figura 1.5 mostra un esempio di questa documentazione.



ura 1.5 Documentazione per la classe `Scanner`.

1.4 Riepilogo

- La memoria principale del computer conserva il programma attualmente in esecuzione e gran parte dei dati che il programma stesso sta utilizzando. La memoria principale del computer è suddivisa in una serie di aree di memoria numerate, chiamate byte. Questa memoria è volatile: i dati in essa contenuti spariscono nel momento in cui il computer viene spento.
- La memoria ausiliaria del computer viene utilizzata per conservare i dati in una forma più o meno permanente. I dati in essa contenuti permangono anche quando il computer viene spento. Esempi di memoria ausiliaria sono gli hard disk, le flash drive, i CD e i DVD.
- Un compilatore è un programma che traduce un programma scritto in un linguaggio di alto livello (come Java) in un programma scritto in un linguaggio di basso livello. Un interprete è un programma che esegue una traduzione analoga, ma, a differenza del compilatore, esegue le porzioni di codice subito dopo averle tradotte, invece di tradurre il programma nella sua interezza.
- Il compilatore Java traduce un programma Java in un programma in linguaggio bytecode. Quando si lancia l'esecuzione del programma, un interprete, la Java Virtual Machine, traduce il bytecode in istruzioni in linguaggio macchina e poi le esegue.
- Un oggetto è un costrutto programmativo che esegue determinate azioni. Queste azioni, o comportamenti, sono definite dai metodi dell'oggetto. Le caratteristiche, o attributi, di un oggetto sono determinati dai suoi dati. I valori degli attributi determinano lo stato dell'oggetto.
- La programmazione orientata agli oggetti è una metodologia che considera un programma come costituito da oggetti che possono agire da soli o interagire con altri. Un oggetto software può rappresentare un oggetto del mondo reale o può essere un'astrazione.
- I tre principi fondamentali della programmazione a oggetti sono l'incapsulamento, il polimorfismo e l'ereditarietà.
- Una classe è una sorta di stampo (*blueprint*) per gli attributi e i comportamenti di un insieme di oggetti. La classe definisce il tipo di questi oggetti. Tutti gli oggetti della stessa classe hanno gli stessi metodi.
- In un programma Java, un'invocazione di metodo si ottiene scrivendo il nome dell'oggetto, seguito da un punto (chiamato dot), seguito dal nome del metodo e infine da una coppia di parentesi tonde. Nelle parentesi possono essere specificati uno o più argomenti.
- Un algoritmo è un insieme di direttive per risolvere un problema. Per essere considerate un algoritmo, le direttive devono essere espresse in maniera completa e precisa, in modo che chiunque possa seguirle senza dover introdurre ulteriori dettagli o compiere scelte che non sono state specificate nelle direttive.
- Lo pseudocodice è una combinazione di termini italiani e di termini appartenenti a un linguaggio di programmazione. È usato per scrivere le direttive di un algoritmo.

- La sintassi di un linguaggio di programmazione è costituita dall'insieme delle sue regole grammaticali. Queste regole stabiliscono se un'istruzione è corretta. Il compilatore rileva gli errori di sintassi in un programma.

1.5 Esercizi

1. In cosa differisce la memoria principale del computer da quella ausiliaria?
2. Dopo aver utilizzato un editor di testi per scrivere un programma, il programma risiede nella memoria principale o in quella ausiliaria?
3. Quando un computer esegue un programma, il programma risiede nella memoria principale o in quella ausiliaria?
4. In cosa differisce il linguaggio macchina da Java?
5. In cosa differisce il bytecode dal linguaggio macchina?
6. Cosa mostrano sullo schermo le seguenti istruzioni se inserite in un programma Java?

```
int eta;
eta = 20;
System.out.println("La mia eta' e'");
System.out.println(eta);
```

7. Scrivere la o le istruzioni che possono essere usate in un programma Java per mostrare sullo schermo il seguente output.

```
3
2
1
```

8. Scrivere le istruzioni che posso essere utilizzate in un programma Java per leggere l'età inserita attraverso la tastiera e per mostrarla sullo schermo.
9. Scrivere le istruzioni che possono essere utilizzate in un programma Java affinché, inserendo tramite tastiera l'anno di nascita di una persona e un numero n che specifica gli anni, venga determinato l'anno in cui festeggerà o ha festeggiato il suo n-esimo compleanno.
10. Scrivere le istruzioni che possono essere utilizzate in un programma Java per leggere due interi e mostrare i numeri interi nell'intervallo compresi i numeri stessi. Per esempio, tra 3 e 6, gli interi nell'intervallo sono; 3, 4, 5 e 6.
11. Un singolo bit può rappresentare due valori: 0 e 1. Due bit possono rappresentare quattro valori: 00, 01, 10 e 11. Tre bit possono rappresentare otto valori: 000, 001, 010, 011, 100, 101, 110 e 111. Quanti valori si possono rappresentare con:

    a. 8 bit? b. 16 bit? c. 32 bit?

12. Accedere alla documentazione della Java Class Library dal sito Web di Oracle (al momento la documentazione è all'indirizzo http://docs.oracle.com/javase/7/docs/api/). Si trovi la descrizione della classe `Scanner`. Quanti metodi sono descritti nella sezione *Method Summary* (sommario dei metodi)?

13. Quali potrebbero essere gli attributi di un oggetto che rappresenta una canzone? E quali quelli di un oggetto che rappresenta una *playlist* di canzoni?
14. Quali comportamenti potrebbe avere una canzone? Quali comportamenti potrebbe avere una *playlist* di canzoni? Paragonare i differenti comportamenti dei due tipi di oggetti.
15. Quali potrebbero essere gli attributi e i metodi di un oggetto che rappresenta una carta di credito?
16. Si supponga di avere un numero x maggiore di 1. Scrivere un algoritmo che calcoli il più grande intero k tale che 2^k sia minore o uguale a x.
17. Scrivere un algoritmo che trovi il valore massimo in una lista di valori.

1.6 Progetti

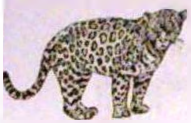
1. Si scarichi il programma mostrato nel Listato 1.1 dal sito Web specificato nella prefazione. Si nomini il file `PrimoProgramma.java`. Si compili il programma in modo da non ottenere messaggi d'errore. Quindi si lanci l'esecuzione del programma.
2. Si modifichi il programma Java descritto nel Progetto 1 in modo che sommi tre numeri invece di due. Si compili il programma in modo da non ottenere messaggi d'errore. Quindi si lanci l'esecuzione del programma.
3. Scrivere un programma Java che mostri sullo schermo la figura seguente. *Suggerimento*: scrivere una sequenza di istruzioni `println` che mostrino righe di asterischi e spazi bianchi.

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

4. Scrivere un programma completo per il problema descritto nell'Esercizio 9.
5. Scrivere un programma completo per il problema formulato nell'Esercizio 10.

Capitolo 2

Nozioni di base



OBIETTIVI

- Descrivere i tipi di dato Java utilizzati per valori semplici, come numeri e caratteri.
- Scrivere istruzioni Java per dichiarare variabili e definire costanti con nome.
- Scrivere istruzioni di assegnamento ed espressioni che contengono variabili e costanti.
- Definire stringhe di caratteri ed eseguire semplici operazioni sulle stringhe.
- Scrivere istruzioni Java che permettano di catturare un input dalla tastiera e di scrivere l'output sullo schermo.
- Aderire alle convenzioni stilistiche comuni.
- Inserire commenti significativi all'interno dei programmi.

Questo capitolo introduce le prime nozioni di programmazione Java che permettono lo sviluppo di alcuni semplici programmi. I contenuti presentati nel Paragrafo 2.1 potrebbero risultare familiari a chi conosce già altri linguaggi di programmazione, come Visual Basic, C, C++ o C#. Tuttavia questa parte iniziale è di particolare utilità per comprendere il modo in cui tali concetti vengono espressi in Java.

Prerequisiti

Chi non avesse letto il Capitolo 1, dovrebbe almeno consultare il Paragrafo 1.2.3, "Il primo programma Java", per familiarizzare con i concetti di classe, oggetto e metodo.

2.1 Variabili ed espressioni

Questo paragrafo illustra l'utilizzo di variabili ed espressioni aritmetiche all'interno di programmi Java. Alcuni dei termini utilizzati in questo capitolo sono già stati introdotti nel Capitolo 1.

2.1.1 Variabili

In un programma, le **variabili** sono utilizzate per memorizzare dati, per esempio numeri e lettere. Possono essere considerate come una sorta di contenitore. Il numero, la lettera o un qualsiasi altro dato contenuto in una variabile è chiamato il **valore** della variabile. Questo valore può essere modificato a run-time: in un certo momento la variabile potrebbe contenere un certo valore, per esempio il numero 6, ma dopo che il programma ha eseguito una serie di computazioni, questa stessa variabile potrebbe contenere un valore diverso, per esempio 4.

Si consideri per esempio il programma rappresentato nel Listato 2.1. Questo programma utilizza le variabili `numeroDiCestini`, `uovaPerCestino` e `uovaTotali`. Quando viene eseguito, l'istruzione:

```
uovaPerCestino = 6
```

assegna alla variabile `uovaPerCestino` il valore 6.

In Java le variabili sono realizzate come aree di memoria, descritte nel Capitolo 1. A ciascuna variabile è assegnata un'area di memoria. Quando alla variabile viene assegnato un valore, il valore viene codificato come una sequenza di 0 e 1 e viene riposto nell'area di memoria assegnata alla variabile.

È buona norma scegliere un nome significativo per le variabili. Il nome di una variabile, infatti, dovrebbe suggerire il suo scopo o indicare il tipo di dato che conterrà. Per esempio, una variabile utilizzata per contare degli oggetti dovrebbe chiamarsi `conteggio` oppure `count`, il corrispondente termine inglese. Se la variabile è utilizzata per contenere la velocità di un'automobile, il nome più appropriato dovrebbe essere `velocita`. Il nome di una variabile non dovrebbe mai essere costituito da un solo carattere, come `x` o `y`. Chi legge l'istruzione:

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

non ha modo di comprendere che cosa stia sommando il programma. I nomi di variabili dovrebbero inoltre seguire alcune regole di sillabazione, che saranno descritte nel Paragrafo "Identificatori Java".

Prima di poter utilizzare una variabile è necessario fornire alcune informazioni descrittive. Il compilatore ha bisogno, infatti, di conoscere il nome della variabile, quanto spazio di memoria deve allocare per essa e il modo in cui codificare i dati contenuti nella variabile. Queste informazioni vengono fornite nella **dichiarazione della variabile**. Ogni variabile in un programma Java deve essere dichiarata prima di poter essere utilizzata.

La dichiarazione della variabile indica al computer che tipo di dato sarà contenuto dalla variabile, indica, cioè, il tipo della variabile. Poiché tipi di dato differenti vengono memorizzati nella memoria del computer in modi differenti, il computer deve conoscere a priori il tipo della variabile, in modo da sapere come memorizzare e recuperare il valore di tale variabile dalla memoria.

Per esempio, la seguente riga tratta dal Listato 2.1 dichiara che le variabili `numeroDiCestini`, `uovaPerCestino` e `totaleUova` sono di tipo `int`:

```
int numeroDiCestini, uovaPerCestino, totaleUova;
```

Una dichiarazione di variabile è costituita dal nome di un tipo, seguito da un elenco di nomi di variabili separati da una virgola. La dichiarazione termina con un punto e virgola. Tutte le variabili indicate nell'elenco sono dunque dello stesso tipo, quello indicato all'inizio della dichiarazione.

Se il tipo della variabile è `int`, la variabile può contenere numeri interi come 42, -99, 0 o 2001. Il termine `int` è l'abbreviazione di ***integer*** ("intero" in inglese). Se il tipo è `double`, la variabile può contenere numeri con una parte decimale. Se il tipo è `char`, la variabile può contenere uno qualsiasi dei caratteri della tastiera del computer.

In un programma Java, ogni variabile deve essere dichiarata prima di essere usata. Normalmente una variabile viene dichiarata appena prima di essere utilizzata, oppure all'inizio di una sezione del programma racchiusa tra parentesi graffe. Nei semplici programmi visti in precedenza, le variabili sono state dichiarate prima di essere utilizzate oppure dopo la riga:

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

LISTATO 2.1 Un semplice programma Java.

MyLab

```
public class CestiniUova {

    public static void main(String[] args) {

        int numeroDiCestini, uovaPerCestino, totaleUova;

        numeroDiCestini = 10;
        uovaPerCestino = 6;

        totaleUova = numeroDiCestini * uovaPerCestino;

        System.out.println("Se hai");
        System.out.println(uovaPerCestino + " uova per cestino e");
        System.out.println(numeroDiCestini + " cestini");
        System.out.println("il numero totale di uova e' " + totaleUova);
    }
}
```

Dichiarazioni di variabili

Istruzione di assegnamento

Esempio di output

```
Se hai
6 uova per cestino e
10 cestini
il numero totale delle uova e' 60
```

Dichiarazione di variabili

In un programma Java le variabili devono essere dichiarate prima di poter esser utilizzate. Una dichiarazione di variabile ha la seguente forma.

Sintassi

tipo variabile_1, variabile_2, ... ;

Esempi

```
int totaleAssegni, totaleOperazioni;
double somma, tassoInteresse;
char risposta;
```

2.1.2 Tipi

Un tipo (o tipo di dato) specifica un insieme di valori e le operazioni che possono essere eseguite su di esso. I valori, infatti, hanno un particolare tipo perché sono memorizzati in memoria nello stesso formato e possono essere utilizzati con le stesse operazioni.

Variabili sintattiche

Parole come *tipo*, *variabile_1* o *variabile_2* compaiono in questo testo per descrivere la sintassi Java, ma non compaiono nel codice Java. Si tratta, infatti, di **variabili sintattiche**: parole chiave che devono essere sostituite con parole appartenenti alla categoria che descrivono. Per esempio, la parola *tipo* può essere sostituita da `int`, `double`, `char` o da un qualsiasi altro tipo. *variabile_1* e *variabile_2* possono essere sostituite da un nome di variabile.

Java possiede due tipi di dato principali: i tipi classe e i tipi primitivi. Come viene indicato dal nome stesso, un **tipo classe** (*class type*) è un tipo per gli oggetti di una classe. Poiché una classe è una sorta di stampo per gli oggetti, essa specifica il modo in cui sono memorizzati i valori del suo tipo e definisce le possibili operazioni che possono essere compiute su di essi. Come è stato descritto nel capitolo precedente, un tipo classe ha lo stesso nome della classe. Per esempio, stringhe tra apici come `"Java e' bello"` sono valori del tipo classe `String`, che sarà presentata più avanti in questo capitolo.

Le variabili di **tipo primitivo** sono più semplici degli oggetti (i valori delle classi), poiché questi ultimi includono sia metodi sia valori. Un valore di un tipo primitivo è un valore non decomponibile, come un singolo numero o una singola lettera. I tipi `int`, `double` e `char` sono esempi di tipi primitivi.

La Figura 2.1 mostra tutti i tipi primitivi di Java. Quattro tipi rappresentano valori interi: `byte`, `short`, `int` e `long`. L'unica differenza tra i diversi tipi di interi è l'insieme di numeri che possono rappresentare e la quantità di memoria che occupano. Quando non si è in grado di decidere quale tipo di intero utilizzare, è meglio scegliere `int`.

Un numero che presenta una parte decimale (come i numeri 9.99, 3.14159, -5.63 e 5.0) è detto **numero in virgola mobile** (*floating-point number*). Bisogna notare che 5.0 è

un numero in virgola mobile e non un intero. Se un numero possiede una parte decimale, anche se è uguale a zero, è un numero in virgola mobile. Come mostrato in Figura 2.1, Java ha due tipi per rappresentare i numeri in virgola mobile: `float` e `double`. Il seguente frammento di codice, per esempio, dichiara due variabili, una di tipo `float` e una di tipo `double`:

```
float costo;
double capacita;
```

Così come per i tipi interi, la differenza tra `float` e `double` riguarda l'insieme di valori rappresentabili e l'occupazione di memoria. Quando non si è in grado di decidere tra il tipo `float` e il tipo `double` è meglio utilizzare `double`.

Il tipo primitivo `char` viene utilizzato per rappresentare singoli caratteri, come lettere o caratteri di punteggiatura. Per esempio, il frammento di codice seguente dichiara la variabile `simbolo` di tipo `char`, salva il carattere `A` nella variabile `simbolo` e infine mostra sullo schermo il contenuto della variabile `simbolo`, ovvero `A`:

```
char simbolo;
simbolo = 'A';
System.out.println(simbolo);
```

Nei programmi Java i singoli caratteri devono essere racchiusi in una coppia di singoli apici, come `'A'`. Attenzione: in una coppia di singoli apici può essere specificato un solo carattere. Inoltre, una stessa lettera è rappresentata con un codice diverso quando è maiuscola o quando è minuscola. In altre parole, `'a'` e `'A'` corrispondono a due codici differenti.

L'ultimo tipo primitivo è `boolean`. Questo tipo può assumere soltanto due valori: `true` (vero) e `false` (falso). Una variabile `boolean` può essere utilizzata, per esempio, per memorizzare la risposta a una domanda come: "È vero che `totaleUova` è minore di 12?". Il prossimo capitolo descriverà il tipo `boolean` più in dettaglio.

Nome del tipo	Tipo di valore	Memoria usata	Intervallo di valori
byte	Intero	1 byte	da -128 a 127
short	Intero	2 byte	da -32.768 a 32.767
int	Intero	4 byte	da -2.147.483.648 a 2.147.483.647
long	Intero	8 byte	da -9.223.372.036.8547.75.808 a 9.223.372.036.854.775.807
float	Numero in virgola mobile	4 byte	da $\pm 3{,}40282347 \quad 10^{+38}$ a $\pm 1{,}40239846 \quad 10^{-45}$
double	Numero in virgola mobile	8 byte	da $\pm 1{,}79769313486231570 \quad 10^{+308}$ a $\pm 4.94065645841246544 \quad 10^{-324}$
char	Carattere singolo (Unicode)	2 byte	Tutti i valori Unicode da 0 a 65.535
boolean		1 bit	True o False

Figura 2.1 I tipi primitivi.

In Java il nome di tutti i tipi primitivi ha l'iniziale minuscola. Il prossimo paragrafo descrive una convenzione che prevede che il nome dei tipi classe (e quindi il nome delle classi), inizi con una lettera maiuscola. Sebbene sia le variabili di tipo classe, sia quelle di tipo primitivo siano dichiarate allo stesso modo, questi due tipi di variabili memorizzano i propri valori utilizzando meccanismi differenti. Il Capitolo 8 illustrerà più in dettaglio le variabili di tipo classe. Questo capitolo e i prossimi due si concentreranno sui tipi primitivi. Alcune variabili di tipo classe saranno utilizzate anche prima del Capitolo 8; per il momento, tuttavia, verranno utilizzate allo stesso modo delle variabili di tipo primitivo.

2.1.3 Identificatori Java

Il termine tecnico utilizzato nei linguaggi di programmazione per indicare un nome (per esempio di una variabile) è **identificatore** (*identifier*).

In Java un identificatore (un nome) può essere composto solo da lettere, cifre da 0 a 9 e il carattere *underscore* (_)[1]. Il primo carattere di un identificatore non può però essere una cifra. Nessun nome, inoltre, può contenere spazi o altri caratteri come il punto (.) o l'asterisco (*). Non c'è alcun limite alla lunghezza di un identificatore e Java accetta anche identificatori con un nome molto lungo. Java è **case sensitive**, termine inglese che indica il fatto che lettere maiuscole e minuscole sono considerate differenti. Per esempio, per Java i nomi `totaleuova`, `totaleUova` e `TotaleUova` sono tre identificatori distinti e perciò possono essere utilizzati per rappresentare tre variabili differenti. Scrivere variabili che differiscono solo per la presenza di maiuscole e minuscole è chiaramente una cattiva abitudine; tuttavia è un comportamento lecito e il compilatore accetterebbe le tre variabili. Qualsiasi nome che rispetti i vincoli descritti, può quindi essere utilizzato per identificare le variabili del programma. Tuttavia ci sono alcune linee guida da seguire nella scelta del nome delle variabili.

Negli esempi precedenti è stato utilizzato il nome `totaleUova`. Anche nomi come `TotaleUova` o `totale_uova` sarebbero stati legali, tuttavia questi nomi avrebbero violato alcune convenzioni sull'uso delle lettere maiuscole e minuscole nel nome delle variabili. Queste convenzioni prevedono che un nome di variabile contenga solo lettere e cifre e che nei nomi costituiti da più parole, la separazione sia ottenuta solo utilizzando l'iniziale maiuscola per ciascuna parola dopo la prima.

Ecco alcuni esempi di nomi che seguono questa convenzione.

```
lavaggioAuto
TuaClasse
CestiniDelleUova
totaleUova
```

Alcuni di questi nomi iniziano con una lettera maiuscola, mentre altri, come `totaleUova`, iniziano con una lettera minuscola. In questo testo sarà sempre seguita la convenzione che prevede che il nome di una classe inizi con una lettera maiuscola, mentre il nome di una variabile o metodo inizi con una lettera minuscola.

I seguenti identificatori non sono invece leciti in Java e provocheranno pertanto un errore di compilazione:

```
totale.uova
ri-eseguire
```

[1] Java permette l'utilizzo del carattere $ all'interno di un identificativo. Tale carattere ha però un significato particolare e perciò è meglio evitarlo.

```
Cinque*
7up
```

I primi tre nomi contengono tutti caratteri illegali, rispettivamente un punto, un trattino o un asterisco. L'ultimo nome invece è illegale in quanto inizia con una cifra.

Alcuni termini in un programma Java, come i tipi primitivi e la parola `if`, sono detti **parole chiave** (*keywords*) o **parole riservate**. Questi termini hanno uno speciale significato predefinito nel linguaggio Java e non possono essere utilizzati come nomi di variabili, metodi o per qualsiasi altro scopo che non sia quello per cui sono stati definiti. Tutte le parole chiave Java sono scritte in minuscolo. L'elenco completo di queste parole chiave è riportato nell'Appendice 4. I listati dei programmi presentati in questo testo contengono parole chiave come `public`, `class`, `static` e `void` colorati di blu. Gli editor di testo integrati negli IDE spesso evidenziano le parole chiave in modo simile.

Altre parole, come `main` e `println`, hanno un significato predefinito, ma non sono parole chiave. Questo significa che si può cambiare il loro significato. Tale comportamento è comunque sconsigliato, poiché può confondere chi legge il programma.

Identificatori (nomi)

In Java qualsiasi nome, sia esso un nome di variabile, di classe o di metodo, è chiamato identificatore. Gli identificatori non devono iniziare con un numero e possono contenere solo lettere, cifre da 0 a 9 e il carattere *underscore* (_). Anche il simbolo $ è permesso ma, poiché viene utilizzato per altri scopi specifici, è meglio non utilizzarlo in un identificatore Java.

Le lettere maiuscole e minuscole sono considerate caratteri differenti.

Sebbene non sia formalmente richiesto dal linguaggio Java, è buona pratica far iniziare il nome di una classe con una lettera maiuscola e utilizzare invece lettere minuscole come iniziali per i nomi di variabili e metodi. Questi nomi contengono tipicamente solo lettere o cifre. Se i nomi sono composti da più parole, è buona norma differenziare le diverse parole utilizzando lettere maiuscole per l'iniziale di ciascuna parola dopo la prima.

Java è case sensitive

Non bisogna mai dimenticare che Java è *case sensitive*: se in una parte del programma viene utilizzato l'identificatore `totaleOggetti` e in un'altra parte viene utilizzato `TotaleOggetti`, Java li considera differenti. Affinché Java riconosca due nomi come uguali, questi devono presentare esattamente la stessa combinazione di lettere maiuscole e minuscole.

FAQ Perché seguire le convenzioni?

Seguendo le convenzioni sui nomi, i programmi diventano più semplici da leggere e comprendere. Le convenzioni sui nomi utilizzate in questo testo sono universalmente riconosciute dai programmatori Java e saranno presentate a mano a mano nei diversi capitoli.

2.1.4 Istruzioni di assegnamento

Il modo più semplice per assegnare un valore a una variabile (o per modificarlo) è quello di utilizzare un'**istruzione di assegnamento**. Per esempio, per assegnare il valore 42 alla variabile `risposta`, di tipo `int`, si può utilizzare la seguente istruzione:

```
risposta = 42;
```

Il simbolo uguale, =, quando viene utilizzato in un'istruzione di assegnamento è detto **operatore di assegnamento** (*assignment operator*). L'istruzione di assegnamento indica al computer di cambiare il valore memorizzato nella variabile posta a sinistra dell'operatore di assegnamento, con il valore dell'**espressione** posta sul lato destro. Un'istruzione di assegnamento, quindi, consiste sempre di una singola variabile seguita dall'operatore di assegnamento, seguita da un'espressione. L'istruzione di assegnamento termina con un punto e virgola. Un'istruzione di assegnamento ha quindi la seguente forma:

variabile = *espressione*;

espressione può essere un'altra variabile, un numero oppure un'espressione più complicata, costruita utilizzando **operatori aritmetici** (*arithmetic operators*), per esempio + e -, per combinare variabili e numeri.

Di seguito sono forniti vari esempi di istruzioni di assegnamento:

```
somma = 3.99;
primaIniziale = 'B';
punteggio = partiteVinte + bonus;
uovaPerCestino = uovaPerCestino - 2;
```

I nomi `somma`, `punteggio`, `partiteVinte` e così via sono variabili. Negli esempi precedenti la variabile `somma` è stata dichiarata di tipo `double`, la variabile `primaIniziale` di tipo `char` e le rimanenti variabili di tipo `int`. Quando un'istruzione di assegnamento viene eseguita, il computer prima valuta l'espressione posta sul lato destro dell'operatore di assegnamento (=) per calcolarne il valore, poi assegna il risultato alla variabile posta sul lato sinistro dell'operatore di assegnamento. L'operatore di assegnamento chiede quindi al computer di rendere la variabile uguale al valore dell'espressione che segue.

Per esempio, se la variabile `partiteVinte` avesse il valore 7 e la variabile `bonus` avesse il valore 2, la seguente istruzione assegnerebbe alla variabile `punteggio` il valore 9:

```
punteggio = partiteVinte + bonus;
```

L'istruzione seguente, presentata nel Listato 2.1, è un altro esempio di istruzione di assegnamento:

```
totaleUova = numeroDiCestini * uovaPerCestino;
```

Questa istruzione chiede al computer di memorizzare nella variabile `totaleUova` il risultato della moltiplicazione fra il contenuto della variabile `numeroDiCestini` e il contenuto della variabile `uovaPerCestino`. Il carattere asterisco (`*`) è l'operatore di moltiplicazione in Java. Una stessa variabile può anche comparire su entrambi i lati dell'operatore di assegnamento. Si consideri la seguente istruzione:

```
somma = somma + 10;
```

Questa istruzione non indica che `somma` è uguale a 10 unità più del proprio stesso valore, che è chiaramente impossibile. Chiede invece al computer di sommare 10 unità all'attuale

valore della variabile `somma`, reinserendo poi il risultato come *nuovo* valore di `somma`. In pratica questa istruzione permette di incrementare di 10 unità il valore della variabile `somma`. Per comprendere questo comportamento, basta ricordare che, quando esegue un assegnamento, il computer prima computa il valore dell'espressione posta a destra dell'operatore di assegnamento e solo successivamente assegna il risultato alla variabile posta a sinistra dell'operatore. Un altro esempio simile è il seguente, in cui il valore della variabile `uovaPerCestino` viene decrementato di 2 unità:

```
uovaPerCestino = uovaPerCestino - 2;
```

Istruzioni di assegnamento che coinvolgono tipi primitivi

Un'istruzione di assegnamento in cui sulla sinistra del simbolo uguale, =, è presente una variabile di tipo primitivo, causa le seguenti azioni: prima viene computata l'espressione posta sul lato destro del simbolo =, poi il risultato della computazione viene assegnato alla variabile posta sul lato sinistro.

Sintassi

variabile = *espressione*;

Esempi

```
punteggio = partiteGiocate - partitePerse;
interesse = tasso * saldo;
numero = numero + 5;
```

Inizializzare le variabili

Una variabile che è stata dichiarata, ma alla quale non sia ancora stato assegnato un valore mediante un'istruzione di assegnamento (o in qualche altro modo), è detta **non inizializzata**. È probabile che essa contenga un valore di *default*, ma è comunque buona norma assegnarle esplicitamente un valore.

Un modo facile per assicurarsi che una variabile venga inizializzata consiste nell'inizializzarla al momento della dichiarazione. Semplicemente basta combinare la dichiarazione con un'istruzione di assegnamento, come nell'esempio che segue:

```
int somma = 0;
double tasso = 0.075;
char grado = 'A';
int saldo = 1000, nuovoSaldo;
```

Si noti che all'interno di una dichiarazione è possibile inizializzare anche solo alcune delle variabili dichiarate.

A volte il compilatore può segnalare che una variabile non è stata inizializzata. Nella maggior parte dei casi ciò corrisponde al vero. Solo raramente il compilatore può commettere un errore. In ogni caso, il compilatore non porterà a buon fine la compilazione finché non verrà convinto che la variabile in questione è inizializzata. Pertanto, conviene sempre inizializzare le variabili al momento della dichiarazione, anche se, prima di essere utilizzate, verranno loro assegnati valori differenti.

Combinare una dichiarazione di variabile e un assegnamento

È possibile combinare una dichiarazione di variabile con un'istruzione di assegnamento che fornisce alla variabile stessa un valore.

Sintassi

tipo variabile_1 = espressione_1, variabile_2 = espressione_2, ...;

Esempi

```
int numero = 0, incremento = 5;
double altezza = 12.34, prezzo = 7.3 + incremento;
char risposta = 's';
```

2.1.5 Semplici operazioni di input

Il Listato 2.1 contiene istruzioni che assegnano un valore specifico alle variabili `uovaPerCestino` e `numeroDiCestini`. Tuttavia sarebbe più utile se questi valori fossero forniti direttamente dall'utente in modo che il programma possa essere utilizzato più volte con valori differenti. Il Listato 2.2 mostra una versione riveduta del programma, che chiede all'utente di inserire questi numeri come input, attraverso la tastiera.

LISTATO 2.2 *Un programma con input da tastiera.*

```
import java.util.Scanner;

public class CestiniUova2 {

  public static void main(String[] args) {

    int numeroDiCestini, uovaPerCestino, totaleUova;

    Scanner tastiera = new Scanner(System.in);
    System.out.println("Inserisci il numero di uova per ciascun cestino:");
    uovaPerCestino = tastiera.nextInt();
    System.out.println("Inserisci il numero di cestini:");
    numeroDiCestini = tastiera.nextInt();

    totaleUova = numeroDiCestini * uovaPerCestino;

    System.out.println("Se hai");
    System.out.println(uovaPerCestino + " uova per cestino e");
    System.out.println(numeroDiCestini + " cestini");
    System.out.println("il numero totale di uova e' " + totaleUova);

    System.out.println("Rimuoviamo ora due uova da ciascun cestino.");

    uovaPerCestino = uovaPerCestino - 2;
    totaleUova = numeroDiCestini * uovaPerCestino;
```

Carica la classe `Scanner` dal package (libreria) `java.util`

Predispone il programma affinché possa leggere da tastiera

Legge un numero da tastiera

```
        System.out.println("Ora hai");
        System.out.println(uovaPerCestino + " uova per cestino e");
        System.out.println(numeroDiCestini + " cestini.");
        System.out.println("Il nuovo numero totale di uova e' " + totaleUova);
    }
}
```

Esempio di output

```
Inserisci il numero di uova per ciascun cestino:
6
Inserisci il numero dei cestini:
10
Se hai
6 uova per cestino e
10 cestini
il numero totale di uova e' 60
Rimuoviamo ora due uova da ciascun cestino.
Ora hai
4 uova per cestino e
10 cestini.
Il nuovo numero totale di uova e' 40
```

Il programma utilizza la classe `Scanner`, fornita da Java, per catturare l'input da tastiera. Dato che il programma deve importare la definizione della classe `Scanner` dal package `java.util`, il listato inizia con la seguente istruzione:

```
import java.util.Scanner;
```

Le righe successive eseguono alcune operazioni di inizializzazione (*setup*) che permetteranno poi di accettare l'input dalla tastiera:

```
Scanner tastiera = new Scanner(System.in);
```

La riga precedente deve comparire prima della prima istruzione che permette di catturare l'input da tastiera. Nell'esempio del Listato 2.2 questa istruzione è:

```
uovaPerCestino = tastiera.nextInt();
```

Questa istruzione di assegnamento fornisce un valore alla variabile `uovaPerCestino`. L'espressione sul lato destro dell'assegnamento,

```
tastiera.nextInt()
```

legge dalla tastiera un valore `int`. Quindi l'istruzione di assegnamento fa sì che il nuovo valore `int` introdotto alla tastiera diventi il valore della variabile `uovaPerCestino`, sostituendo un eventuale valore contenuto nella variabile. Quando l'utente scrive i numeri alla tastiera, deve separarli con uno o più spazi oppure deve scrivere i vari numeri su righe differenti. Il Paragrafo 2.3 spiegherà in dettaglio l'input da tastiera.

2.1.6 Un esempio di output su schermo

Questo paragrafo descrive brevemente come funziona l'output su schermo, in modo da poter scrivere e comprendere programmi come quello del Listato 2.2.

`System` è una classe fornita dal linguaggio Java e `out` è un particolare oggetto di questa classe. `println` è uno dei metodi dell'oggetto `out`. Il Capitolo 9 fornirà maggiori dettagli su questa notazione.

La seguente riga mostra a schermo il valore della variabile `uovaPerCestino` seguita dalla frase "*uova per cestino e*".

```
System.out.println(uovaPerCestino + " uova per cestino e");
```

In questo caso il simbolo + non indica una somma aritmetica, ma una concatenazione: questa riga può quindi essere interpretata come un'istruzione per stampare il valore di `uovaPerCestino` seguita dalla stringa "*uova per cestino e*".

Il Paragrafo 2.3 presenterà più in dettaglio l'output su schermo.

2.1.7 Costanti

Il valore di una variabile può variare nel tempo. Non per niente si chiama proprio "variabile". Un numero, per esempio il numero 2, non può mai cambiare: il suo valore resta sempre 2. In Java termini come 2 oppure 3.7 sono chiamati **costanti** (*constants*) o **letterali** (*literals*).

Le costanti non sono necessariamente numeriche, per esempio `'A'`, `'B'` e `'$'` sono tre costanti di tipo `char`. Il loro valore non può cambiare, ma possono essere utilizzate in un'istruzione di assegnamento per cambiare il valore di una variabile di tipo `char`. Per esempio, l'istruzione

```
primaIniziale = 'B';
```

assegna alla variabile `primaIniziale` di tipo `char` il valore 'B'. C'è sostanzialmente un solo modo per scrivere una costante di tipo `char`, cioè mettendo il carattere tra apici. Invece ci sono più modi per scrivere costanti numeriche.

Le costanti di tipo intero sono scritte come ci si può aspettare: per esempio 2, 3, 0, -3 o 752. Una costante intera può essere preceduta da un segno, + o -, come in +12 o -72. Le costanti numeriche non possono contenere virgole: il numero 1,000 *non è corretto* in Java. Le costanti intere non possono contenere numeri decimali. Un numero decimale è un numero in virgola mobile.

Le costanti in virgola mobile possono essere scritte in due modi: il modo semplice consiste nello scrivere le cifre decimali dopo il punto di separazione. Per esempio, 2.5 è una costante in virgola mobile. L'altro modo è leggermente più complicato ed è simile alla **notazione scientifica** comunemente utilizzata in matematica e fisica. Per esempio, il numero 865000000.0 può essere scritto molto più semplicemente come:

8.65×10^8

Java presenta una notazione simile, chiamata **e notation** oppure **notazione in virgola mobile** (*floating-point notation*). Poiché le tastiere non permettono di scrivere gli esponenti, il numero 10 viene omesso e sia il 10 sia il carattere (di moltiplicazione) x sono sostituiti dalla lettera e. In Java, quindi, il numero 8.65×10^8 si scrive come `8.65e8`. La e sta per esponente, dal momento che sostituisce la moltiplicazione per 10 e l'elevamento a potenza. Le notazioni `8.65e8` e `865000000.0` sono equivalenti in un programma Java. In maniera analoga, il numero 4.83×10^{-4}, che corrisponde al numero 0.000483, può essere scritto in Java come `0.000483` o `4.83e-4`. Anche le forme `0.483e-3` e `48.3e-5` sono valide: Java, infatti, non pone alcuna restrizione sulla posizione del punto decimale. Il numero che precede la lettera e può contenere il punto decimale (la virgola),

sebbene non sia necessario. Il numero dopo la lettera e invece non può contenerlo. Dal momento che moltiplicare per 10 corrisponde a spostare il punto decimale in un numero, il numero positivo dopo la e può essere visto come un indicatore di quante cifre decimali debba essere spostato il punto decimale. Se il numero dopo la e è negativo, la virgola viene spostata a sinistra invece che a destra. Per esempio `2.48e4` corrisponde al numero `24800.0` mentre `2.48e-2` corrisponde a `0.0248`.

FAQ Perché i numeri in virgola mobile sono chiamati così?

I numeri in virgola mobile sono chiamati in questo modo perché l'utilizzo della notazione e permette di far "fluttuare" (dall'inglese *floating*) la virgola in una nuova posizione spostando l'esponente. Per esempio, si può spostare la virgola in `0.000483` dopo il 4 esprimendo questo numero come `4.83e-4`.

FAQ C'è differenza tra le costanti 5 e 5.0?

I numeri 5 e 5.0 corrispondono concettualmente allo stesso numero. Tuttavia Java li considera differenti: il numero 5 è una costante intera di tipo `int`, mentre 5.0 è una costante in virgola mobile di tipo `double`. Il numero 5.0 contiene una parte decimale, anche se il decimale è 0. Sebbene i numeri 5 (`int`) e 5.0 (`double`) possano avere lo stesso valore, Java li memorizza diversamente. Sia i numeri interi, sia quelli in virgola mobile contengono un numero finito di cifre quando sono memorizzati su un computer, ma solo gli interi sono considerati quantità esatte. Poiché i numeri in virgola mobile hanno una parte decimale, essi sono sempre considerati approssimazioni.

Imprecisione nei numeri in virgola mobile

I numeri in virgola mobile vengono memorizzati con una precisione limitata e quindi sono quantità approssimate. Per esempio, la frazione 1/3 equivale a:

0.333333...

dove i tre puntini indicano che la cifra 3 si replica all'infinito. Il computer memorizza i numeri in un formato simile alla rappresentazione presentata nella riga precedente, ma ha spazio solo per un numero limitato di cifre. Se il computer può memorizzare solo dieci cifre dopo la virgola, il numero 1/3 viene memorizzato come:

0.3333333333

Questo numero è leggermente più piccolo del numero 1/3 e quindi ne rappresenta solo un'approssimazione. In realtà il computer memorizza i numeri in notazione binaria, invece che in base 10, ma il principio resta comunque lo stesso.

Non tutti i numeri in virgola mobile perdono accuratezza quando vengono memorizzati. Valori interi come 29.0 possono essere memorizzati in maniera esatta anche in virgola mobile, così come frazioni come 1/2. Tuttavia non è possibile sapere se un numero in virgola mobile è preciso o meno. In caso di dubbi, è meglio supporre che i numeri in virgola mobile siano approssimazioni.

2.1.8 Costanti con nome

Java fornisce un meccanismo che permette di definire una variabile, inizializzarla e far sì che questo valore non sia più modificabile. La sintassi è la seguente:

```
public static final tipo variabile = costante;
```

Per esempio, l'istruzione che segue attribuisce il nome `PI` al valore costante `3.14159`:

```
public static final double PI = 3.14159;
```

Questo è un modo per attribuire un nome (per esempio `PI`) a una costante (per esempio `3.14159`). La parte:

```
double PI = 3.14159;
```

dichiara semplicemente che `PI` è una variabile e la inizializza a 3.14159. I termini che precedono questa istruzione modificano il comportamento della variabile `PI` in vari modi. Il termine `public` indica che non ci sono restrizioni sulla porzione di codice in cui è possibile usare il nome `PI`. Il termine `static` è una parola chiave che sarà descritta nel Capitolo 9; per il momento è sufficiente sapere che deve essere utilizzata per dichiarare le costanti. Infine, il termine `final` indica che il valore 3.14159 è il valore definitivo assegnato alla variabile `PI`; in altre parole, il programma non può modificarlo.

La convenzione solitamente adottata per nominare le costanti prevede di utilizzare solo lettere maiuscole e il carattere *underscore* (_) per separare le parole che compongono il nome. Per esempio, in un'agenda elettronica è possibile dover definire la seguente costante:

```
public static final int GIORNI_PER_SETTIMANA = 7;
```

Sebbene questa convenzione non sia obbligatoria dal punto di vista della sintassi di Java, viene utilizzata dalla maggior parte degli sviluppatori, poiché i programmi risultano più facili da leggere quando le costanti e le variabili sono semplici da individuare.

Costanti con nome

Per definire il nome di una costante è necessario scrivere le parole chiave `public static final` prima della dichiarazione della variabile. La dichiarazione include anche il valore costante, per l'inizializzazione. Questa dichiarazione va posta all'interno della definizione della classe, ma al di fuori della definizione dei metodi, `main` incluso.

Sintassi

```
public static final tipo variabile = costante;
```

Esempi

```
public static final int MAX_STRIKES = 3;
public static final double TASSO_DI_INTERESSE = 6.99;
public static final String MOTTO = "Il cliente ha sempre ragione!"
public static final char SCALA = 'K';
```

Sebbene non sia esplicitamente richiesto dal linguaggio, la maggior parte dei programmatori scrive i nomi delle costanti usando solo lettere maiuscole, usando il carattere *underscore* per separare le parole.

2.1.9 Compatibilità di assegnamento

Un valore di tipo `double` come 3.5 non può essere assegnato a una variabile di tipo `int`. Ma anche il valore `double` 3.0 non può essere assegnato a una variabile di tipo `int`. In generale, un valore di un tipo non può essere memorizzato in una variabile di un altro tipo, a meno che non venga convertito in un valore compatibile con il tipo di destinazione. Tuttavia, quando si utilizzano valori numerici, a volte questa conversione viene eseguita automaticamente da Java. La conversione viene effettuata in modo automatico quando si assegna un valore intero a una variabile in virgola mobile, come nell'istruzione seguente:

```
double variabileDouble = 7;
```

La conversione viene effettuata in maniera automatica anche nel caso di assegnamenti leggermente più complessi, come nelle righe seguenti:

```
int variabileInt = 7;
double variabileDouble = variabileInt;
```

Più in generale, un valore può essere assegnato a una qualsiasi variabile il cui tipo compare alla destra del tipo del valore nell'elenco seguente:

byte → short → int → long → float → double

In altre parole, un valore di tipo `long` può essere assegnato a una variabile di tipo `float` o `double` (oltre che ovviamente a una variabile `long`), ma non può essere assegnato a una variabile di tipo `byte`, `short` o `int`. Quello indicato non è un elenco arbitrario, ma dipende dal fatto che spostandosi da sinistra verso destra i tipi diventano sempre più precisi o permettono valori di dimensioni maggiori o permettono di usare valori decimali. Inoltre si possono assegnare valori di tipo `char` a variabili di tipo `int` o di qualsiasi tipo che segue `int` nell'elenco presentato. Questa particolare compatibilità di assegnamento è utile per comprendere l'input da tastiera, descritto nei prossimi paragrafi. Tuttavia è consigliabile non assegnare valori `char` a tipi `int`, se non in particolari situazioni[2].
Per assegnare un valore di tipo `double` a una variabile di tipo `int`, occorre cambiare il tipo del valore utilizzando una conversione di tipo (*type cast*), argomento di cui si parlerà nel prossimo paragrafo.

Compatibilità di assegnamento

Un valore può essere assegnato a una qualsiasi variabile il cui tipo compare alla destra del tipo del valore nel seguente elenco:

byte → short → int → long → float → double

In particolare, occorre notare che si può assegnare un valore di un qualsiasi tipo intero a una variabile di un qualsiasi tipo in virgola mobile. È inoltre ammissibile assegnare un valore di tipo `char` a una variabile di tipo `int` o a un qualsiasi tipo numerico più a destra di `int` nell'elenco dei tipi indicato sopra.

[2] Chi avesse già utilizzato altri linguaggi di programmazione, come il C o il C++, può stupirsi del fatto che non è possibile assegnare un valore di tipo `char` a una variabile di tipo `byte`. Questo è dovuto al fatto che Java riserva due byte di memoria per ogni valore di tipo `char`, ma naturalmente riserva un solo byte per valori di tipo `byte`.

2.1.10 Conversioni di tipo

In Java e nella maggior parte dei linguaggi di programmazione, una **conversione di tipo** (*type cast*) cambia il tipo di un valore. Per esempio, per cambiare il tipo del valore 2.0 da `double` a `int` è necessaria una conversione di tipo. Il paragrafo precedente ha indicato in quali casi un valore di un tipo può essere assegnato a una variabile di un altro tipo attraverso una conversione automatica. In tutti gli altri casi, l'assegnamento di un valore di un tipo a una variabile di un altro tipo può avvenire solo grazie a una conversione esplicita. Ecco come svolgere le conversioni in Java. Si considerino le seguenti righe:

```
double distanza = 9.0;
int punti = distanza;  <-------- Questo assegnamento è illecito
```

L'ultimo assegnamento non è consentito in Java: un valore di tipo `double` non può essere assegnato a una variabile di tipo `int`, anche se il suo valore presenta tutti zeri nella parte decimale (e quindi concettualmente si tratterebbe di un intero). Per assegnare un valore di tipo `double` a una variabile di tipo `int` è necessario inserire la conversione `(int)` di fronte al valore o alla variabile che contiene il valore. Per esempio, l'istruzione precedente potrebbe essere sostituita con il seguente assegnamento:

```
int punti = (int)distanza;  <-------- Questo assegnamento è lecito
```

L'espressione `(int)distanza` è detta conversione di tipo. Questa operazione non modifica né la variabile `distanza`, né il valore in essa contenuto: la variabile `punti` conterrà però la "versione `int`" del valore memorizzato in `distanza`. Se il valore di `distanza` fosse 25.36, il valore di `(int)distanza` diventerebbe 25. `punti` conterrebbe, quindi, 25, ma il valore di `distanza` rimarrebbe 25.36. Se invece il valore di `distanza` fosse 9.0, il valore assegnato a `punti` sarebbe 9 e il valore di `distanza` resterebbe comunque inalterato.

Espressioni come `(int)25.36` o `(int)distanza` *generano* valori di tipo `int`. Una conversione di tipo non altera il valore della variabile di origine. È un po' come calcolare il numero intero di Euro nel portafogli. Se si possiedono 25.36 €, il numero intero di Euro che si possiede è 25. La somma di 25.36 € in tasca non è cambiata, è stata semplicemente utilizzata per generare il valore intero 25. Si consideri, per esempio, il seguente codice:

```
double contoCena = 25.36;
int contoCenaPiuMancia = (int)contoCena + 5;
System.out.println("Il valore di contoCenaPiuMancia e' " +
                   contoCenaPiuMancia);
```

L'espressione `(int)contoCena` genera il valore intero 25, perciò l'output di queste istruzioni risulta essere:

```
Il valore di contoCenaPiuMancia e' 30
```

Tuttavia la variabile `contoCena` conterrà ancora il valore 25.36.

Si noti che ogni volta che si compie una conversione di tipo da `double` a `int` (o da un qualsiasi tipo in virgola mobile a un qualsiasi tipo intero) il valore non viene arrotondato. La parte che segue la virgola viene semplicemente scartata. Questa operazione è detta di **troncamento** (*truncating*). Per esempio, le seguenti istruzioni:

```
double contoCena = 26.99;
int numeroEuro = (int)contoCena;
```

assegnano a `numeroEuro` il valore 26 e non 27; quindi, il valore *non viene arrotondato*.

Come si è già detto, quando si assegna un valore intero a una variabile in virgola mobile, per esempio `double`, l'intero viene automaticamente convertito al tipo della variabile. Per esempio, l'assegnamento:

```
double punto = 7;
```

è equivalente a:

```
double punto = (double)7;
```

Nella prima versione dell'assegnamento, la conversione di tipo è implicita. La seconda versione è comunque lecita.

Le conversioni di tipo

In molte situazioni non è possibile memorizzare un valore di un tipo in una variabile di un altro tipo, a meno che non venga eseguita una conversione di tipo nel tipo di destinazione.

Sintassi

(tipo)espressione

Esempi

```
double supposizione = 7.8;
int risposta = (int)supposizione;
```

Il valore memorizzato nella variabile `riposta` sarà 7. Occorre notare che il valore è stato troncato e non arrotondato. Inoltre, la variabile `supposizione` non è cambiata in alcun modo: contiene ancora il valore 7.8. L'ultima istruzione di assegnamento riguarda solo il valore memorizzato in `risposta`.

Come convertire un carattere in un intero

Alle volte Java tratta i tipi `char` come interi, ma l'assegnamento di interi a variabili di tipo `char` non ha alcuna relazione con il significato dei caratteri stessi. Per esempio, la conversione di tipo seguente produce il valore `int` corrispondente al carattere `'7'`:

```
char simbolo = '7';
System.out.println((int)simbolo);
```

Sebbene ci si aspetti che venga visualizzato il valore 7, questo non avviene: il programma visualizza infatti il valore 55. Java, come tutti gli altri linguaggi di programmazione, utilizza dei numeri per codificare i caratteri. Ciascun carattere corrisponde a un intero. In questa corrispondenza, le cifre da 0 a 9 sono caratteri, così come le lettere e il segno +. Non esiste alcuna corrispondenza tra codici e lettere. Di fatto, è come se fosse stato stilato l'elenco di tutti i possibili caratteri e, in seguito, questi fossero stati numerati nell'ordine in cui apparivano nell'elenco. Secondo quest'ordine al carattere "7" è stata attribuita la posizione numero 55. Questo tipo di numerazione è detta sistema Unicode e sarà presentata più avanti in questo stesso capitolo. Il sistema Unicode è il corrispettivo del sistema ASCII per i caratteri dell'alfabeto inglese.

2.1.11 Operatori aritmetici

In Java possono essere eseguite operazioni aritmetiche come somme, sottrazioni, moltiplicazioni e divisioni utilizzando, rispettivamente, gli operatori `+`, `-`, `*` e `/`. Le operazioni aritmetiche sono espresse come nell'aritmetica o algebra ordinaria. Le variabili e i numeri, ovvero gli **operandi**, possono essere combinati con questi operatori e con le parentesi per formare un'**espressione aritmetica**. Java possiede, oltre agli operatori sopra menzionati, un quinto operatore aritmetico, `%`, che sarà descritto brevemente in seguito.

MyLab

Video 2.1 Semplici espressioni aritmetiche

Di norma il significato di un'espressione aritmetica coincide con quello che ci si aspetta. Tuttavia esistono alcune sottigliezze riguardanti il tipo del risultato e, a volte, anche il suo valore. I cinque operatori aritmetici possono essere utilizzati con operandi di tipo intero, in virgola mobile e anche con operandi di tipo differente. Il tipo del risultato dipende dal tipo degli operandi.

Si consideri, per esempio, una semplice espressione che impiega due soli operandi, cioè due variabili, due numeri o una variabile e un numero. Se entrambi gli operandi sono dello stesso tipo, il risultato è di quello stesso tipo. Se uno degli operandi è un numero in virgola mobile e l'atro è un intero, il risultato sarà in virgola mobile.

Si consideri, per esempio, l'espressione seguente:

```
somma + variazione
```

Se le variabili `somma` e `variazione` sono entrambe di tipo `int`, il risultato, cioè il valore restituito dall'operazione, sarà di tipo `int`. Se `somma` o `variazione` o entrambe sono di tipo `double`, il risultato sarà di tipo `double`. Il tipo del risultato dell'operazione viene determinato secondo lo stesso criterio anche se al posto dell'operatore di somma, `+`, viene impiegato l'operatore `-`, `*`, `/` o `%`.

Espressioni di maggiori dimensioni che impiegano più di due operandi possono sempre essere scomposte in una serie di passi che riguardano solo due operandi per volta. Per esempio, per calcolare l'espressione seguente:

```
bilancio + (bilancio * tasso)
```

il computer calcola `bilancio * tasso`, ottenendo un risultato parziale, e quindi calcola la somma tra il risultato ottenuto e `bilancio`. Questo vuol dire che la stessa regola utilizzata per determinare il tipo di un'espressione contenente due operandi può essere utilizzata per espressioni più complicate: se tutti gli elementi combinati sono dello stesso tipo, il risultato sarà di quel tipo; se uno degli elementi è di un tipo in virgola mobile, il risultato sarà un numero in virgola mobile.

Per conoscere il tipo di valore prodotto da un'espressione aritmetica, basta operare nel seguente modo. Il tipo del risultato prodotto corrisponde a uno dei tipi presenti nell'espressione: quello più a destra nel seguente elenco (lo stesso presentato in precedenza):

byte → short → int → long → float → double

All'operatore di divisione (`/`) occorre prestare più attenzione, perché il tipo del risultato può avere varie conseguenze sul risultato prodotto. Quando si combinano due operandi con l'operatore di divisione e almeno uno degli operandi è un numero in virgola mobile, il risultato corrisponde al risultato che normalmente ci si aspetta da una divisione. Per esempio, `9.0/2` presenta un operando di tipo `double`, `9.0`, e quindi il risultato sarà di tipo `double`: `4.5`. Ma quando entrambi gli operandi sono di tipo intero, il risultato non corrisponde a quanto ci si aspetta. Per esempio, l'espressione `9/2` ha due operandi di tipo `int` e

quindi genera come risultato il valore 4, di tipo `int` e non 4.5. La parte decimale si perde. Quando si dividono due interi, il risultato *non viene arrotondato*: la parte decimale viene semplicemente scartata (cioè troncata). Di conseguenza, il risultato della divisione 11/3 è 3 e non 3.6666.... Anche se la parte decimale dopo lo 0 è 0, questa viene persa. Questa differenza, apparentemente insignificante, può essere di una certa rilevanza. Per esempio, 8.0/2 restituisce il valore di tipo `double` 4.0, che è solo una quantità approssimata. Tuttavia, 8/2 restituisce il valore `int` 4, che è una quantità esatta. La natura approssimata di 4.0 può influire sulla precisione di qualsiasi calcolo che viene eseguito utilizzando questo risultato.

Il quinto operatore Java è l'**operatore resto**, indicato con %. Quando si divide un numero per un altro, si ottiene un risultato (il quoziente) e anche un resto, cioè la quantità rimanente. L'operatore % permette di calcolare il resto della divisione. In genere, l'operatore % viene utilizzato con operandi di tipo intero per recuperare, in un certo senso, la parte decimale dopo la virgola. Pertanto, 14/4 restituisce 3, mentre 14%4 restituisce 2, in quanto 14 diviso per 4 fa 3 con il resto di 2.

L'operatore % viene utilizzato per numerosi scopi. Infatti, permette al programma di individuare con facilità i multipli di due, tre o di qualsiasi numero. Per esempio, per compiere una certa azione solo sui numeri pari occorre sapere se un numero è dispari o pari. Un intero `n` è pari se `n % 2` è uguale a 0, mentre è dispari se `n % 2` è uguale a 1. Analogamente, per compiere una certa operazione con i multipli di 3, basta considerare i numeri interi, copiare ciascun numero in una variabile `n` ed eseguire l'operazione solo se `n % 3` è uguale a 0.

FAQ **Come si comporta l'operatore `%` con i numeri in virgola mobile?**

L'operatore resto viene solitamente utilizzato con operandi interi; tuttavia Java permette di utilizzarlo anche con operandi in virgola mobile. Se `n` e `d` sono numeri in virgola mobile, `n % d` è uguale a `n - (d * q)`, dove `q` è la parte intera di `n / d`. Occorre notare che il segno di `q` è lo stesso di `n / d`. Per esempio, `6.5 % 2.0` fa 0.5, `-6.5 % 2.0` fa -0.5 e `6.5 % -2.0` fa 0.5.

Occorre, infine, notare che i simboli + e – vengono utilizzati anche per indicare il segno del numero, oltre che per indicare le operazioni di addizione e sottrazione. In ogni caso, Java tratta sempre + e – come operatori. Un **operatore unario** (*unary operator*) è un operatore che possiede un solo operando (un solo oggetto cui viene applicato), come, per esempio, l'operatore – nell'assegnamento seguente:

```
bilancioBancario = -costo;
```

Un **operatore binario** ha invece due operandi, come gli operatori + e * nell'istruzione seguente:

```
totale = costo + (tasse * sconto);
```

Occorre notare che uno stesso operatore può a volte essere utilizzato sia come operatore unario, sia come operatore binario. Per esempio, i simboli + e – possono fungere sia da operatori binari sia unari.

FAQ Gli spazi hanno un ruolo nelle espressioni aritmetiche?

Nelle istruzioni Java, gli spazi non hanno alcun ruolo. L'unica eccezione è rappresentata dagli spazi presenti fra apici semplici o doppi. In tutti gli altri casi, l'aggiunta di spazi è utile per migliorare la leggibilità del codice. Per esempio, come presentato negli esempi precedenti, è buona norma porre uno spazio prima e dopo ogni operatore binario.

2.1.12 Parentesi e regole di precedenza

Le parentesi possono essere utilizzate per raggruppare gli elementi di un'espressione aritmetica nello stesso modo in cui vengono impiegate in algebra. Con l'aiuto delle parentesi, è possibile indicare al compilatore quali operazioni svolgere per prime, quali per seconde, per terze e così via. Si considerino, per esempio, le due espressioni seguenti che differiscono solo per la posizione delle parentesi:

```
(costo + tasse) * sconto
costo + (tasse * sconto)
```

Per calcolare il risultato della prima espressione, il computer prima somma le variabili `costo` e `tasse`, e poi moltiplica il risultato per la variabile `sconto`. Per calcolare il risultato della seconda espressione, il computer moltiplica `tasse` per `sconto` e quindi somma il risultato alla variabile `costo`. Se si calcolano queste espressioni assegnando alle variabili dei numeri, si vedrà che producono risultati differenti.

Se si omettono le parentesi, il computer calcolerà comunque l'espressione. Per esempio, la seguente istruzione di assegnamento:

```
totale = costo + tasse * sconto;
```

è equivalente all'istruzione:

```
totale = costo + (tasse * sconto);
```

Quando le parentesi vengono omesse, il computer esegue prima le moltiplicazioni e poi le addizioni. Più in generale, quando l'ordine delle operazioni non è determinato dalle parentesi, il computer eseguirà le operazioni nell'ordine specificato dalle **regole di precedenza** elencate nella Figura 2.2[3]. Gli operatori elencati più in alto hanno una **maggiore precedenza.** Quando il computer deve decidere quale operazione eseguire e l'ordine non è indicato esplicitamente dalle parentesi, esegue prima le operazioni che hanno una precedenza maggiore, poi quelle che hanno una precedenza minore. Alcuni operatori hanno la stessa precedenza; in questi casi il computer esegue le operazioni nell'ordine con cui si presentano gli operatori. Le operazioni binarie fra operatori che hanno la stessa precedenza vengono eseguite da sinistra a destra; le operazioni unarie, invece, da destra a sinistra.

Queste regole di precedenza sono analoghe a quelle impiegate in algebra. Al di fuori di casi molto particolari, è sempre opportuno inserire le parentesi in un'espressione, anche qualora l'ordine di esecuzione delle operazioni fosse lo stesso indicato dalle regole di precedenza.

[3] La Figura 2.2 illustra solo gli operatori discussi in questo capitolo. Ulteriori regole di precedenza verranno presentate nel Capitolo 3.

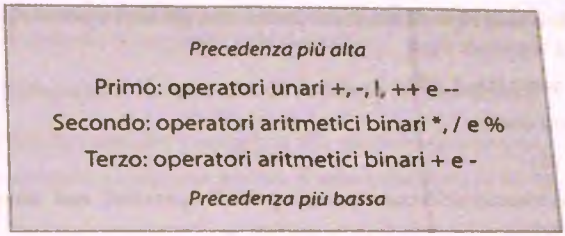


Figura 2.2 Regole di precedenza.

Le parentesi facilitano infatti la comprensibilità dell'espressione. Bisogna però considerare il caso opposto: troppe parentesi inutili potrebbero pregiudicare la comprensibilità dell'espressione. Un caso comune in cui si omettono le parentesi è la moltiplicazione affiancata da un'addizione. Per esempio, la seguente istruzione:

```
bilancio = bilancio + (tassoInteresse * bilancio);
```

di solito viene scritta come:

```
bilancio = bilancio + tassoInteresse * bilancio;
```

MyLab

Video 2.2 Espressioni aritmetiche

Entrambe le forme sono accettabili e hanno entrambe lo stesso significato. La Figura 2.3 mostra alcuni esempi di espressioni aritmetiche scritte in Java e indica alcune delle parentesi che si potrebbero omettere.

Matematica ordinaria	Java (forma preferita)	Java (forma con tutte le parentesi)
$tasso^2 + variazione$	tasso * tasso + variazione	(tasso * tasso) + variazione
$2(salario + bonus)$	2 * (salario + bonus)	2 * (salario + bonus)
$\frac{1}{tempo + 3massa}$	1 / (tempo + 3 * massa)	1 / (tempo + (3 * massa))
$\frac{a-7}{t+9v}$	(a - 7) / (t + 9 * v)	(a - 7) / (t + (9 * v))

Figura 2.3 Alcune espressioni aritmetiche in Java.

2.1.13 Operatori di assegnamento ausiliari

L'operatore di assegnamento semplice (=) può essere preceduto da un operatore aritmetico, per esempio +, con lo scopo di svolgere un assegnamento e, contemporaneamente, una modifica del valore. Per esempio, la seguente istruzione incrementa di 5 unità il valore della variabile `quantita`:

```
quantita += 5;
```

In pratica questa istruzione è un'abbreviazione della seguente:

```
quantita = quantita + 5;
```

Si può ottenere lo stesso tipo di risultato anche con gli altri operatori aritmetici -, *, / e %. Per esempio, la seguente riga:

```
quantita = quantita * 25;
```

può essere sostituita con:

```
quantita *= 25;
```

Sebbene non sia necessario utilizzare questi speciali operatori, essi sono molto apprezzati dagli sviluppatori Java.

CASO DI STUDIO
UN DISTRIBUTORE AUTOMATICO DI MONETE

I distributori automatici integrano spesso un computer che controlla le loro operazioni. Questo caso di studio mostra un programma che gestisce una delle attività che un computer di questo tipo dovrebbe svolgere. In questo programma si suppone per comodità che l'input e l'output vengano svolti rispettivamente tramite la tastiera e lo schermo. Per poter utilizzare questo programma in un vero distributore automatico, sarebbe necessario inserire questo codice in un programma di maggiori dimensioni, che si occupa di caricare i dati da una periferica differente dalla tastiera e che invia il risultato non solo allo schermo, ma anche su un componente di diverso tipo.

In questo caso di studio l'utente inserisce una quantità da cambiare compresa tra 1 e 99 centesimi. Il programma risponde indicando all'utente la combinazione di monete che corrisponde a quella cifra.

Per esempio, se l'utente scrive 45 (centesimi), il programma risponde che 45 centesimi possono essere cambiati con due monete da 20 centesimi e una da 5. Si supponga che l'interazione tra l'utente e il programma debba essere di questo tipo:

```
Inserisci un numero intero da 1 a 99.
Identifichero' una combinazione di monete
che corrisponde a tale cifra.
97
97 centesimi in moneta:
1 cinquanta centesimi
2 venti centesimi
0 dieci centesimi
1 cinque centesimi
1 due centesimi e
0 un centesimi
```

Scrivere un dialogo di esempio, come quello mostrato sopra, prima di scrivere il codice aiuta la risoluzione del problema.

Il programma richiede alcune variabili per memorizzare la quantità di monete e il numero di ciascun tipo di moneta. Sono quindi necessarie le seguenti variabili:

```
int quantita;
int cinquantaCent, ventiCent, dieciCent, cinqueCent, dueCent, unCent;
```

Dopo aver determinato le variabili, occorre individuare la soluzione del problema.

È necessario un algoritmo per calcolare il numero di ciascun tipo di moneta. Si supponga, per esempio, di definire il seguente pseudocodice.

Algoritmo per computare il numero di monete in centesimi

1. Leggi la quantità da cambiare e assegnala alla variabile `quantita`.
2. Assegna alla variabile `cinquantaCent` il valore massimo di 50 centesimi presenti in `quantita`.
3. Assegna alla variabile `quantita` la cifra che rimane dopo aver rimosso i 50 centesimi.
4. Assegna alla variabile `ventiCent` il valore massimo di 20 centesimi presenti in `quantita`.
5. Assegna alla variabile `quantita` la cifra che rimane dopo aver rimosso i 20 centesimi.
6. Assegna alla variabile `dieciCent` il valore massimo di 10 centesimi in `quantita`.
7. Assegna alla variabile `quantita` la cifra che rimane dopo aver rimosso i 10 centesimi.
8. Assegna alla variabile `cinqueCent` il valore massimo di 5 centesimi in `quantita`.
9. Assegna alla variabile `quantita` la cifra che rimane dopo aver rimosso i 5 centesimi.
10. Assegna alla variabile `dueCent` il valore massimo di 2 centesimi in `quantita`.
11. Assegna alla variabile `quantita` la cifra che rimane dopo aver rimosso i 2 centesimi.
12. `unCent = quantita`.
13. Mostra il valore di `quantita` seguito dal valore delle singole monete.

Questi passi sembrano sensati, tuttavia prima di iniziare a scrivere il codice è bene fare una prova. Se avessimo 97 centesimi, assegneremmo 97 alla variabile `quantita`. Il numero di 50 centesimi contenuti in 97 è 1, quindi `cinquantaCent` diventa 1 e abbiamo 97 – 1 * 50 = 47 centesimi rimasti in `quantita`. In 47 abbiamo 2 monete da 20 centesimi, quindi assegniamo 2 a `ventiCent`, sottraiamo 40 da 47 e quindi rimangono 7 centesimi in `quantita`. Nessun 10 centesimi è contenuto in 7, quindi `quantita` resta invariato. Un 5 centesimi è contenuto in 7, perciò `cinqueCent` vale 1 e restano 2 centesimi. Una moneta da 2 centesimi è contenuta in 2, perciò `dueCent` vale 1 e infine abbiamo che `unCent` vale 0.

Per stampare il risultato vorremmo specificare la cifra da cui siamo partiti; tuttavia, al termine dell'algoritmo, `quantita` non contiene più il valore 97 iniziale. Dato che l'algoritmo cambia continuamente il valore di `quantita`, il valore iniziale viene perso. Per correggere l'algoritmo si può o stampare subito il valore digitato dall'utente oppure memorizzare questo valore in un'altra variabile, per esempio `quantitaIniziale`, che non verrà modificata e mostrarla alla fine della computazione. Questa seconda soluzione è mostrata nel seguente pseudocodice.

Algoritmo per computare il numero di monete in centesimi

1. Leggi la somma e assegnala alla variabile `quantita`.
2. `quantitaIniziale = quantita;`.
3. Assegna alla variabile `cinquantaCent` il valore massimo di 50 centesimi presenti in `quantita`.

4. Assegna alla variabile `quantita` la cifra che rimane dopo aver rimosso i 50 centesimi.
5. Assegna alla variabile `ventiCent` il valore massimo di 20 centesimi presenti in `quantita`.
6. Assegna alla variabile `quantita` la cifra che rimane dopo aver rimosso i 20 centesimi.
7. Assegna alla variabile `dieciCent` il valore massimo di 10 centesimi in `quantita`.
8. Assegna alla variabile `quantita` la cifra che rimane dopo aver rimosso i 10 centesimi.
9. Assegna alla variabile `cinqueCent` il valore massimo di 5 centesimi in `quantita`.
10. Assegna alla variabile `quantita` la cifra che rimane dopo aver rimosso i 5 centesimi.
11. Assegna alla variabile `dueCent` il valore massimo di 2 centesimi in `quantita`.
12. Assegna alla variabile `quantita` la cifra che rimane dopo aver rimosso i 2 centesimi.
13. `unCent = quantita`.
14. Mostra il valore di `quantitaIniziale` seguito dal valore delle singole monete.

A questo punto è possibile scrivere il codice Java che esegue le operazioni descritte dallo pseudocodice. La prima riga dello pseudocodice scrive un messaggio all'utente e legge il numero dalla tastiera. Il seguente codice Java corrisponde alla prima istruzione pseudocodice:

```
System.out.println("Inserisci un intero compreso tra 1 e 99.");
System.out.println("Identifichero' una combinazione di monete");
System.out.println("che corrisponde a tale cifra.");
Scanner tastiera = new Scanner(System.in);
quantita = tastiera.nextInt();
```

La riga successiva di pseudocodice assegna il valore di `quantitaIniziale` e corrisponde già a un'istruzione Java; perciò non occorre fare alcuna traduzione.

Fino a questo punto il codice Java del nostro programma corrisponde al seguente:

```
public static void main(String[] args) {
    int quantita, quantitaIniziale;
    int cinquantaCent, ventiCent, dieciCent, cinqueCent, dueCent, unCent;

    System.out.println("Inserisci un intero compreso tra 1 e 99.");
    System.out.println("Identifichero' una combinazione di monete");
    System.out.println("che corrisponde a tale cifra.");
    Scanner tastiera = new Scanner(System.in);
    quantita = tastiera.nextInt();
    quantitaIniziale = quantita;
```

Occorre ora tradurre in codice Java lo pseudocodice seguente:

3. Assegna alla variabile `cinquantaCent` il valore massimo di 50 centesimi presenti in `quantita`.
4. Assegna alla variabile `quantita` la cifra che rimane dopo aver rimosso i 50 centesimi.

Per individuare il numero di volte in cui la moneta da 50 centesimi è contenuta in 97, basta dividere 97 per 50 e quindi usare il resto della divisione per capire quante monete

devono ancora essere restituite. Per compiere queste operazioni è possibile utilizzare gli operatori / e %. Per esempio:

97 / 50 è pari a 1 (il massimo numero di 50 in 97)
97 % 50 è pari a 47 (il resto)

Sostituendo `quantita` a 97 e `cinquantaCent` a 50 si ottengono le seguenti istruzioni:

```
cinquantaCent = quantita / 50;
quantita = quantita % 50;
```

Le monete da 20, 10, 5, 2 e 1 centesimo possono essere calcolate allo stesso modo e quindi potremmo scrivere il seguente codice:

```
ventiCent = quantita / 20;
quantita = quantita % 20;
dieciCent = quantita / 10;
quantita = quantita % 10;
cinqueCent = quantita / 5;
quantita = quantita % 5;
dueCent = quantita / 2;
quantita = quantita % 2;
```

La parte rimante del codice è semplice da derivare. Il programma finale è mostrato nel Listato 2.3.

Dopo aver scritto un programma, è necessario collaudarlo con diversi dati. Il programma presentato in questo esempio dovrebbe essere provato sia con valori che restituiscono 0 per tutte le monete e con valori che permettono di assegnare tutti i possibili valori alle diverse monete. Per esempio, potremmo collaudare il nostro programma con ciascuno dei seguenti input: 0, 1, 2, 4, 5, 10, 20, 30, 40, 50 e 60.

Sebbene tutti i test effettuati siano terminati con successo, l'output del programma non usa una grammatica corretta. Per esempio, se inseriamo come input 26 centesimi otteniamo il seguente output:

```
26 centesimi in moneta corrispondono a:
0 monete da cinquanta centesimi
1 monete da venti centesimi
0 monete da dieci centesimi
1 monete da cinque centesimi
0 monete da due centesimi e
1 monete da un centesimo
```

Sebbene i valori siano corretti, le etichette dovrebbero riportare, per esempio,

```
1 moneta da un centesimo
```

invece di:

```
1 monete da un centesimo.
```

Il prossimo capitolo mostrerà come correggere questa situazione.

LISTATO 2.3 Un programma che cambia le monete.

```
import java.util.Scanner;

public class CambiaMonete {

    public static void main(String[] args) {

        int quantita, quantitaIniziale;
        int cinquantaCent, ventiCent, dieciCent, cinqueCent, dueCent, unCent;

        System.out.println("Inserisci un intero compreso tra 1 e 99.");
        System.out.println("Identifichero' una combinazione di monete");
        System.out.println("che corrisponde a tale cifra.");
        Scanner tastiera = new Scanner(System.in);
        quantita = tastiera.nextInt();
        quantitaIniziale = quantita;

        cinquantaCent = quantita / 50;
        quantita = quantita % 50;
        ventiCent = quantita / 20;
        quantita = quantita % 20;
        dieciCent = quantita / 10;
        quantita = quantita % 10;
        cinqueCent = quantita / 5;
        quantita = quantita % 5;
        dueCent = quantita / 2;
        quantita = quantita % 2;
        unCent = quantita;

        System.out.println(quantitaIniziale +
                           " centesimi in moneta corrispondono a:");
        System.out.println(cinquantaCent + " monete da cinquanta centesimi");
        System.out.println(ventiCent + " monete da venti centesimi");
        System.out.println(dieciCent + " monete da dieci centesimi");
        System.out.println(cinqueCent + " monete da cinque centesimi");
        System.out.println(dueCent + " monete da due centesimi e");
        System.out.println(unCent + " monete da un centesimo");
    }
}
```

50 centesimi stanno in 97 una volta con resto di 47.

97 / 50 è 1
97 % 50 è 47.

97 corrisponde a una moneta da 50 centesimi più 47 centesimi

Esempio di output

```
Inserisci un intero compreso tra 1 e 99.
Identifichero' una combinazione di monete
che corrisponde a tale cifra.
97
97 centesimi in moneta corrispondono a:
1 monete da cinquanta centesimi
2 monete da venti centesimi
0 monete da dieci centesimi
```

```
1 monete da cinque centesimi
1 monete da due centesimi e
0 monete da un centesimo
```

La struttura base di un programma

Molte applicazioni, come per esempio il programma scritto nei paragrafi precedenti, condividono una struttura molto simile. I passi fondamentali corrispondono a un estratto di un discorso tenuto da Dale Carnegie (1888-1955): "Anticipate agli ascoltatori quello che state per raccontargli, quindi ditelo e infine riassumete quello che avete detto".

I programmi di solito impiegano i seguenti passi.

1. Preparare: dichiarare le variabili e spiegare il programma all'utente.
2. Input: chiedere e leggere un input da parte dell'utente.
3. Processare: eseguire le attività.
4. Output: mostrare il risultato.

Questa struttura, che può essere abbreviata come PIPO, coincide con il comportamento dei programmi presentati nella prima parte di questo testo. Tenere a mente questi passi è utile per organizzare i pensieri mentre si progettano e sviluppano i programmi.

2.1.14 Operatori di incremento e decremento

Java possiede due operatori utilizzati per incrementare o decrementare di una unità il valore di una variabile. Sono operatori particolari e si può anche evitare di utilizzarli, tuttavia sono spesso di grande comodità.

L'**operatore di incremento** si scrive utilizzando due segni più (++). Il seguente codice, per esempio, incrementa di una unità il valore della variabile `contatore`:

```
contatore++;
```

Se prima di questa istruzione la variabile contatore avesse avuto il valore 5, dopo l'esecuzione dell'istruzione assumerebbe il valore 6. Questa istruzione è pertanto equivalente a:

```
contatore = contatore + 1;
```

L'**operatore di decremento** è analogo, con l'unica differenza che sottrae invece di aggiungere. L'operatore di decremento viene scritto con due segni meno (--). Per esempio, la seguente istruzione decrementa di una unità il valore della variabile `contatore`:

```
contatore--;
```

Se prima dell'esecuzione la variabile `contatore` avesse avuto il valore 5, dopo l'esecuzione dell'istruzione avrebbe assunto il valore 4. Questa istruzione è perciò equivalente a:

```
contatore = contatore - 1;
```

Gli operatori di incremento e decremento possono essere utilizzati con variabili di qualsiasi tipo numerico; tuttavia vengono per lo più utilizzati con variabili di tipo intero, come `int`.

Il motivo per cui Java presenta questi operatori è storico: li ha ereditati dai linguaggi C e C++. Gli operatori di incremento e decremento sono presenti in vari linguaggi di programmazione poiché l'aggiunta e la rimozione di un'unità è un'operazione molto frequente in programmazione.

2.1.15 Note aggiuntive sugli operatori di incremento e decremento

Gli operatori di incremento e decremento possono essere utilizzati nelle espressioni, sebbene ciò sia sconsigliabile. Quando vengono utilizzati in un'espressione, questi due operatori cambiano il valore della variabile cui sono stati applicati e **restituiscono** (*return*) un valore.

Nelle espressioni, gli operatori ++ e -- possono essere posti prima o dopo una variabile; è importante notare che il significato cambia a seconda della loro posizione. Si consideri, per esempio, il seguente codice:

```
int n = 3;
int m = 4;
int risultato = n * (++m);
```

Dopo che questo codice viene eseguito, il valore di `n` resta invariato a 3, il valore di `m` diventa 5 e il valore della variabile `risultato` è 15. Quindi, l'istruzione `++m` cambia il valore di `m` e restituisce il nuovo valore.

Nell'esempio precedente, l'operatore è stato posto *prima* della variabile. Se fosse stato inserito *dopo* la variabile `m`, il risultato sarebbe stato differente. Si consideri quindi il codice:

```
int n = 3;
int m = 4;
int risultato = n * (m++);
```

In questo caso, dopo l'esecuzione, il valore di `n` è 3, il valore di `m` è 5, come nel caso precedente, ma il risultato è 12, non 15. Questo accade perché, sebbene entrambe le espressioni `n * (++m)` e `n * (m++)` incrementino il valore di `m`, la prima incrementa il valore di `m` *prima* che avvenga la moltiplicazione, mentre la seconda incrementa il valore di `m` solo *dopo* l'esecuzione della moltiplicazione. Sia `++m` che `m++` hanno lo stesso effetto sul valore finale di `m`, ma quando utilizzati all'interno di un'espressione aritmetica hanno un diverso effetto sull'espressione.

Anche l'operatore -- si comporta in maniera simile quando viene utilizzato in un'espressione aritmetica. Sia `--m` che `m--` hanno lo stesso effetto sul valore finale di `m`, ma quando vengono utilizzati in un'espressione restituiscono un valore diverso. Nel caso in cui venga utilizzato `--m` il valore di `m` viene decrementato prima che il suo valore sia utilizzato all'interno dell'espressione; nel caso di `m--` il valore di `m` viene decrementato dopo essere utilizzato all'interno dell'espressione.

Quando un operatore di incremento o di decremento è posto prima di una variabile si usa la **forma prefissa**; quando invece è posto dopo una variabile, si usa una **forma postfissa**. Gli operatori di incremento e decremento possono essere applicati solo alle variabili; non possono essere applicati né alle costanti, né a espressioni aritmetiche più complicate.

2.2 La classe `String`

Le stringhe di caratteri, come "`Inserisci l'ammontare:`", sono trattate in maniera differente dai valori di tipo primitivo. Java non offre un tipo primitivo per le stringhe, tuttavia fornisce una classe chiamata `String` che può essere utilizzata per creare ed elaborare stringhe di caratteri. Le classi costituiscono il cuore di Java. Questa discussione sulla classe `String` permette di rivedere la notazione e la terminologia per le classi introdotta nel Capitolo 1.

2.2.1 Stringhe costanti e variabili

Negli esempi presentati nei paragrafi precedenti sono state già utilizzate costanti di tipo `String`. Per esempio, la stringa tra apici:

```
"Inserisci un numero compreso tra 1 e 99."
```

che compare nell'istruzione seguente tratta dal Listato 2.3:

```
System.out.println("Inserisci un numero compreso tra 1 e 99.");
```

è una costante di tipo `String`.

Un valore di tipo `String` è una stringa racchiusa tra doppi apici. Si tratta, quindi, di una sequenza di caratteri considerati come se fossero un singolo elemento. Una variabile di tipo `String` può assegnare un nome a questi valori. L'istruzione che segue dichiara che `saluto` è una variabile di tipo `String`:

```
String saluto;
```

L'istruzione successiva assegna a `saluto` il valore `String` "`Ciao!`":

```
saluto = "Ciao!";
```

Queste due istruzioni vengono spesso accorpate in una sola:

```
String saluto = "Ciao!";
```

Quando un valore viene assegnato a una variabile di tipo `String`, come `saluto`, questa può essere visualizzata sullo schermo come segue:

```
System.out.println(saluto);
```

Questa istruzione visualizza la seguente riga:

```
Ciao!
```

Una stringa può contenere un numero qualsiasi di caratteri; per esempio "`Ciao!`" contiene cinque caratteri. Una stringa può anche contenere zero caratteri: una stringa di questo tipo viene detta **stringa vuota** e viene rappresentata con due doppi apici adiacenti: "".

La stringa vuota viene utilizzata piuttosto spesso. È inoltre importante considerare la differenza esistente fra la stringa vuota "" e la stringa " ": la seconda stringa non è vuota, in quanto contiene il carattere spazio.

2.2.2 Concatenazione di stringhe

Due stringhe possono essere unite per formare una stringa di maggiori dimensioni. Questa operazione è detta **concatenazione** e viene effettuata con l'operatore +. Quando que-

sto operatore viene applicato a stringhe viene chiamato **operatore di concatenamento**. Si consideri, per esempio, il seguente codice:

```
String saluto, frase;
saluto = "Ciao";
frase = saluto + "amico mio";
System.out.println(frase);
```

Questo codice assegna alla variabile `frase` la stringa "`Ciaoamico mio`" e scrive il seguente messaggio sullo schermo:

```
Ciaoamico mio
```

Come si può notare dall'esempio, nessuno spazio viene aggiunto quando vengono concatenate due stringhe con l'operatore +. Per far sì che la variabile `frase` contenga la stringa "`Ciao amico mio`", l'operazione di assegnamento dovrebbe essere la seguente:

```
frase = saluto + " amico mio";
```

In questo caso è stato aggiunto uno spazio prima della parola `amico`.

L'operatore + può essere utilizzato per concatenare un numero qualsiasi di oggetti `String`. L'operatore `String` può inoltre essere utilizzato per concatenare una stringa con un qualsiasi altro tipo di oggetto. In questo caso il risultato è sempre un oggetto di tipo `String`. Java, infatti, si preoccuperà di rappresentare come una stringa ogni oggetto prodotto dal concatenamento di stringhe tramite l'operatore +. Per elementi semplici come i numeri, Java esegue un'operazione ovvia. Per esempio:

```
String soluzione = "La risposta e' " + 42;
```

assegna alla variabile di tipo `String soluzione` la stringa "`La risposta e' 42`".

Sebbene questo risultato sembri ovvio, Java deve eseguire una conversione di tipo. La costante 42 è, infatti, un numero, mentre "`42`" è un oggetto `String` costituito dal carattere "`4`" seguito dal carattere "`2`". Java converte la costante numerica 42 nella costante stringa "`42`" e quindi concatena le due stringhe "`La risposta e' `" e "`42`" per ottenere la stringa "`La risposta e' 42`".

Usare il simbolo + con le stringhe

Due stringhe possono essere concatenate usando l'operatore +.

Esempio:

```
String nome = "Laura";
String saluto = "Ciao " + nome;
System.out.println(saluto);
```

Queste istruzioni assegnano a `saluto` la stringa "`Ciao Laura`" e quindi mostrano la seguente frase sullo schermo:

```
Ciao Laura
```

Si noti inoltre che è stato aggiunto uno spazio in coda alla stringa "`Ciao`" per separare le parole sull'output.

2.2.3 Metodi di `String`

Una variabile `String` non è una variabile semplice, come una variabile di tipo `int`; si tratta di un oggetto appartenente alla classe `String`. Gli oggetti possiedono metodi e dati. Per esempio, gli oggetti della classe `String` memorizzano dati costituiti da stringhe di caratteri, come "`Ciao`". I metodi forniti dalla classe `String` consentono di elaborare questi dati.

La maggior parte dei metodi di `String` restituisce un valore. Per esempio, il metodo `length` restituisce il numero di caratteri presenti in un oggetto di tipo `String`. Quindi l'istruzione seguente:

```
"Ciao".length();
```

restituisce il valore intero 4. In altre parole, il valore di `"Ciao".length()` è 4, un numero, che può essere memorizzato in una variabile di tipo `int` nel seguente modo:

```
int n = "Ciao".length();
```

L'invocazione di un metodo si ottiene scrivendo il nome dell'oggetto seguito da un punto (*dot*), dal nome del metodo e infine da una coppia di parentesi. Sebbene in questo caso l'oggetto sia una costante, "`Ciao`", di solito i metodi vengono invocati su variabili, come nelle istruzioni seguenti:

```
String saluto = "Ciao";
int n = saluto.length();
```

Per alcuni metodi, come `length`, non è necessario specificare alcun argomento, quindi la coppia di parentesi è vuota. Per altri metodi, come sarà descritto nei prossimi paragrafi, devono essere specificate alcune informazioni tra le parentesi.

Tutti gli oggetti appartenenti a una stessa classe hanno in dotazione gli stessi metodi, ma ciascun oggetto può contenere dati differenti. Per esempio, i due oggetti di tipo `String` "`Ciao`" e "`Arrivederci`" contengono dati diversi, cioè diverse stringhe di caratteri. Tuttavia, questi oggetti hanno gli stessi metodi. Questo implica che sia l'oggetto `String` "`Ciao`", sia l'oggetto `String` "`Arrivederci`" hanno il metodo `length`, che è in dotazione a tutti gli oggetti di tipo `String`.

Nel calcolo della lunghezza di una stringa, vengono considerati tutti i suoi caratteri, compresi gli spazi, i simboli e i caratteri ripetuti. Per esempio, si supponga di dichiarare le seguenti variabili `String`:

```
String comando = "Siediti Fido!"
String risposta = "bau-bau";
```

L'invocazione del metodo `comando.length()` restituisce 13, mentre l'invocazione del metodo `risposta.length()` restituisce 7. L'invocazione del metodo `length` può essere eseguita in qualsiasi punto in cui possa essere utilizzato un valore di tipo `int`. Tutte le seguenti istruzioni sono pertanto consentite in Java:

```
int somma = comando.length();
System.out.println("La lunghezza e' " + comando.length());
somma = comando.length() + 3;
```

Molti dei metodi della classe `String` dipendono dalla posizione dei caratteri nella stringa. Nelle stringhe la posizione di un carattere si conta a partire da 0 e non da 1. Nella stringa

`"Ciao Mamma"`, il carattere `'C'` è nella posizione 0, `'i'` è in posizione 1, la prima `'a'` è in posizione 2 e così via. La posizione di un carattere è spesso chiamata con il termine tecnico **indice** (*index*). È quindi usuale dire che l'indice di `'C'` è 0, quello di `'i'` è 1 e così via. La Figura 2.4 mostra la posizione degli indici in una stringa. I 14 caratteri della stringa `"Java e' bello."` hanno pertanto indici che vanno da 0 a 13.

Il termine **sottostringa** (*substring*) indica una porzione di una stringa. Per esempio, la stringa definita dall'istruzione:

```
String frase = "Java e' bello.";
```

ha la sottostringa `"bello"` che inizia all'indice 8. Il metodo `indexOf` restituisce proprio l'indice di una sottostringa passata come argomento.

L'invocazione `frase.indexOf("bello")` restituisce 8, in quanto la sottostringa `"bello"` parte dall'indice 8, dove si trova la lettera `"b"`. Se la sottostringa si ripresenta più volte in una stringa, `indexOf` restituisce l'indice della prima occorrenza dell'argomento.

Si noti che anche i caratteri di spaziatura, l'apice e il punto sono contati come caratteri nella stringa.

Figura 2.4 Indici nelle stringhe.

La Figura 2.5 descrive alcuni dei metodi della classe `String`. Il prossimo capitolo descrive l'uso dei metodi `equals` e `compareTo` per confrontare due stringhe. Gli altri metodi elencati nella figura potrebbero diventare utili nei prossimi capitoli. La documentazione della Java Class Library sul sito Web di Oracle fornisce informazioni dettagliate sui metodi della classe `String`.

FAQ Qual è il termine tecnico che indica l'oggetto su cui viene invocato un metodo?

Un oggetto ha diversi metodi; quando viene invocato uno di questi metodi, l'oggetto riceve la chiamata e svolge le attività previste dal metodo. Per questo motivo l'oggetto è definito con il termine di **oggetto ricevente** (*receiving object*) o semplicemente **ricevitore** (*receiver*). La documentazione, come quella riportata in Figura 2.5, spesso descrive gli oggetti riceventi semplicemente come **oggetti this** (letteralmente "questo").

FAQ Che cos'è uno spazio bianco (whitespace)?

Tutti i caratteri che non sono visibili quando visualizzati sullo schermo sono chiamati spazi bianchi (*whitespace*) o caratteri di spaziatura. Questi caratteri includono lo spazio singolo (*blank*), il carattere tabulazione (*tab*) e il carattere fine riga (*new-line*).

nome_stringa.`charAt`(*indice*) Restituisce il carattere che si trova alla posizione *indice* della stringa corrente *nome_stringa* (this). Gli indici sono numerati a partire da 0.
nome_stringa.`compareTo`(*altra_stringa*) Confronta la stringa corrente *nome_stringa* (this) con *altra_stringa* per individuare quale viene prima in ordine lessicografico. L'ordine lessicografico corrisponde all'ordine alfabetico quando entrambe le stringhe sono costituite solo da lettere maiuscole o solo da lettere minuscole. Il metodo restituisce un valore intero negativo se la stringa corrente viene prima, 0 (zero) se sono uguali o un numero positivo se la stringa corrente viene dopo.
nome_stringa.`concat`(*altra_stringa*) Restituisce una nuova stringa che presenta gli stessi caratteri della stringa corrente *nome_stringa* (this) concatenati con quelli in *altra_stringa*. Invece di `concat` può essere utilizzato l'operatore + .
nome_stringa.`equals`(*altra_stringa*) Restituisce `true` se la stringa corrente *nome_stringa* (this) e *altra_stringa* sono uguali. Altrimenti restituisce `false`.
nome_stringa.`equalsIgnoreCase`(*altra_stringa*) Si comporta come il metodo `equals`, ma considera uguali le lettere maiuscole e le lettere minuscole della stringa.
nome_stringa.`indexOf`(*altra_stringa*) Restituisce l'indice della prima occorrenza della sottostringa *altra_stringa* nella stringa corrente *nome_stringa* (this). Restituisce -1 se la sottostringa *altra_stringa* non compare. Gli indici sono numerati a partire da 0.
nome_stringa.`lastIndexOf`(*altra_stringa*) Restituisce l'indice dell'ultima occorrenza della sottostringa *altra_stringa* all'interno della stringa corrente *nome_stringa* (this). Restituisce -1 se la sottostringa *altra_stringa* non compare. Gli indici sono numerati a partire da 0.
nome_stringa.`length()` Restituisce la lunghezza della stringa corrente *nome_stringa* (this).
nome_stringa.`toLowerCase()` Restituisce una nuova stringa che presenta gli stessi caratteri della stringa corrente *nome_stringa* (this), ma in cui tutte le lettere maiuscole sono state sostituite con le minuscole corrispondenti.
nome_stringa.`toUpperCase()` Restituisce una nuova stringa che presenta gli stessi caratteri della stringa corrente *nome_stringa* (this), ma in cui tutte le lettere minuscole sono state sostituite con le corrispondenti lettere maiuscole.
nome_stringa.`replace`(*vecchio_carattere*, *nuovo_carattere*) Restituisce una nuova stringa che presenta gli stessi caratteri della stringa corrente *nome_stringa* (this), ma in cui tutte le occorrenze del carattere *vecchio_carattere* sono state sostituite dal carattere *nuovo_carattere*.
nome_stringa.`substring`(*inizio*) Restituisce una nuova stringa che presenta gli stessi caratteri della sottostringa che inizia all'indice *inizio* della stringa corrente *nome_stringa* (this) fino alla fine della stringa. Gli indici sono numerati a partire da 0.
nome_stringa.`substring`(*inizio*, *fine*) Restituisce una nuova stringa che presenta gli stessi caratteri della sottostringa che inizia all'indice *inizio* della stringa corrente *nome_stringa* (this) fino all'indice *fine* escluso. Gli indici sono numerati a partire da 0.
nome_stringa.`trim()` Restituisce una nuova stringa che presenta gli stessi caratteri della stringa corrente *nome_stringa* (this), ma in cui sono stati rimossi i caratteri di spaziatura in testa e in coda alla stringa.

Figura 2.5 Alcuni metodi della classe `String`.

2.2.4 Elaborazione delle stringhe

Tecnicamente gli oggetti di tipo `String` non possono essere modificati. Si noti che nessuno dei metodi elencati nella Figura 2.5 cambia il valore di un oggetto `String`. La classe `String` presenta più metodi di quelli mostrati in Figura 2.5, ma nessuno di questi permette di scrivere istruzioni del tipo: "Cambia in 'z' il quinto carattere della stringa". Questa caratteristica è stata introdotta intenzionalmente in Java per rendere più efficiente l'implementazione della classe `String`, cioè per rendere più veloce l'esecuzione dei metodi e per utilizzare meno memoria. Java ha un'altra classe per rappresentare le stringhe, chiamata `StringBuilder`, che possiede metodi in grado di modificare i dati dei propri oggetti. La classe `StringBuilder` non viene presentata in quanto non è necessaria per la trattazione.

Sebbene il valore di un oggetto `String`, per esempio `"Ciao"`, non possa essere modificato, si possono comunque scrivere programmi che cambiano il valore di una variabile di tipo `String`. Questa operazione è solitamente sufficiente per soddisfare molte necessità di programmazione. Per modificare il valore di una stringa basta utilizzare un operatore di assegnamento come nell'esempio seguente:

```
String nome = "Savitch";
nome = "Walter " + nome;
```

L'assegnamento sulla seconda riga modifica il valore della variabile nome da `"Savitch"` a `"Walter Savitch"`. Il Listato 2.4 mostra un programma che svolge alcune semplici operazioni di elaborazione di stringhe e cambia il valore di una variabile `String`. Il carattere *backslash* (\), che compare nell'argomento passato al metodo `println` sarà descritto nel prossimo paragrafo.

LISTATO 2.4 Usare la classe `String`.

```
public class StringDemo {

public static void main(String[] args) {

        String frase = "Elaborazione di testi? Difficile!";
        int posizione = frase.indexOf("Difficile");
        System.out.println(frase);
        System.out.println("01234567890123456789012345678901234567");
        System.out.println("La parola \"Difficile\" inizia all'indice " +
                           posizione);

        frase = frase.substring(0, posizione) + "Facile!";
        frase = frase.toUpperCase();
        System.out.println("La stringa modificata e':");
        System.out.println(frase);

    }
}
```

Il significato di \" è discusso nel paragrafo "Caratteri di escape"

Esempio di output

```
Elaborazione di testi? Difficile!
01234567890123456789012345678901234567
```

```
La parola "Difficile" inizia all'indice 23
La stringa modificata e':
ELABORAZIONE DI TESTI? FACILE!
```

Indice della stringa fuori dal limite (String index out of bounds)

Il primo carattere di una stringa si trova all'indice 0, non 1. Per questo motivo, se una stringa contiene *n* caratteri, l'ultimo carattere si trova all'indice *n-1*. Tutte le volte che viene invocato un metodo della stringa che riceve come argomento un indice, per esempio `charAt`, occorre prestare attenzione al fatto che il valore di questo indice sia valido. Il valore di un indice è valido se è maggiore o uguale a 0 e minore della lunghezza della stringa. Un indice esterno a questo intervallo di valori, è detto essere **al di fuori dei limiti** (*out of bounds*) o semplicemente **non valido** (*invalid*). Un indice di questo tipo causa un errore a run-time.

2.2.5 Caratteri di escape

Si supponga di voler visualizzare una stringa contenente apici. Per esempio, si supponga di voler visualizzare la seguente stringa:

```
Il termine "Java" indica il nome di un linguaggio!
```

L'istruzione seguente non funziona:

```
System.out.println("Il termine "Java" indica il nome di un linguaggio!");
```

Questa istruzione genera un errore in compilazione. Il problema risiede nel fatto che il compilatore interpreta

```
"Il termine "
```

come una stringa tra apici. Quindi, il compilatore osserva che

```
Java"
```

non rappresenta un'istruzione valida nel linguaggio Java (sebbene il compilatore possa supporre che si tratti di una stringa senza apici oppure senza un simbolo +). Tuttavia il compilatore non può sapere quali siano le intenzioni del programmatore e in particolare non può immaginare che gli apici facciano parte della stringa da visualizzare. Per indicare al compilatore che gli apici fanno parte della stringa, è necessario far loro precedere un carattere *backslash* (\):

```
System.out.println("Il termine \"Java\" indica il nome di un " +
                   "linguaggio!");
```

La Figura 2.6 mostra un elenco di caratteri speciali che devono essere riportati usando il carattere *backslash*. Questi caratteri sono spesso chiamati **sequenze di escape** (*escape sequences*) o **caratteri di escape** (*escape characters*), in quanto sottraggono (*escape* in inglese) al carattere il suo normale significato.

`\"`	Apice doppio.
`\'`	Apice singolo.
`\\`	*Backslash*.
`\n`	Nuova linea. Sposta l'output all'inizio della nuova riga.
`\r`	*Carriage return*. Sposta l'output all'inizio della riga corrente.
`\t`	Tab. Aggiunge spazi bianchi fino al nuovo punto di tabulazione.

Figura 2.6 Caratteri di escape.

È importante notare che ciascuna sequenza di escape rappresenta un solo carattere, anche se viene scritta usando due simboli. La stringa `"\"Ciao\""` non contiene quindi otto caratteri, ma sei: un doppio apice, le lettere C, i, a, o e un altro doppio apice. Comprendere questo aspetto è fondamentale per gestire correttamente gli indici.

Anche l'inclusione di un carattere *backslash* in una stringa è problematica. Per esempio, la stringa `"abc\def"` causa, in fase di compilazione, il seguente messaggio d'errore: "Invalid escape character". Per includere un carattere *backslash* in una stringa è necessario specificare due *backslash*. Pertanto, la stringa `"abc\\def"` visualizzerà il seguente risultato:

```
abc\def
```

Il carattere di escape `\n` indica invece che, nella posizione in cui compare, deve iniziare una nuova riga. Per esempio l'istruzione:

```
System.out.println("Il motto e'\nVincere!")
```

Fa comparire le seguenti due righe sullo schermo:

```
Il motto e'
Vincere!
```

Includere un apice singolo all'interno di una stringa, per esempio `"Java e' bello"`, è perfettamente legale. Se invece si vuole definire una costante contenente un apice singolo occorre utilizzare il carattere di escape `\'`, come nella riga seguente

```
char apiceSingolo ='\'';
```

2.2.6 Set di caratteri Unicode

Un **set di caratteri** è una lista di caratteri a ciascuno dei quali è associato a un numero. Il **set di caratteri ASCII** include tutti i caratteri normalmente usati su una tastiera inglese (ASCII è un acronimo per *American Standard Code for Information Interchange*, letteralmente "codice standard americano per lo scambio di informazioni"). Ciascun carattere ASCII è rappresentato con un numero binario che occupa un solo byte. Questa codifica fornisce quindi 256 caratteri ed è adottata da diversi linguaggi di programmazione.

Il **set di caratteri Unicode** include, oltre all'intero set ASCII, anche i caratteri utilizzati in lingue diverse dall'inglese. Un carattere Unicode occupa 2 byte e pertanto la codifica Unicode fornisce più di 65.000 caratteri differenti.

Al fine di facilitare tutti gli utilizzatori del linguaggio, non solo gli inglesi, gli sviluppatori di Java hanno adottato il set di caratteri Unicode. Il fatto di utilizzare la codifica

Unicode invece che ASCII non ha alcun impatto per chi utilizza una tastiera inglese. In questo caso, infatti, gli sviluppatori possono programmare come se Java utilizzasse la codifica ASCII, che è un sottoinsieme della codifica Unicode. Il vantaggio del set di caratteri Unicode è che permette di gestire con facilità lingue molto diverse dall'inglese. Lo svantaggio principale è invece rappresentato dal fatto che richiede più memoria rispetto ad ASCII per rappresentare i singoli caratteri.

2.3 Operazioni di I/O: la tastiera e lo schermo

Le attività di input e output di un programma sono spesso abbreviate con il temine **I/O**. Un programma Java può svolgere operazioni di I/O in diversi modi. Questo paragrafo descrive alcuni semplici modi per gestire il testo che viene digitato su una tastiera o per visualizzarlo sullo schermo.

2.3.1 Output su schermo

Già i primi esempi presentati in questo testo contenevano alcune istruzioni per la generazione di output sullo schermo. Questo paragrafo riassume e spiega il significato delle istruzioni di output fin qui utilizzate.

Nel Listato 2.3 sono state utilizzate le seguenti istruzioni per inviare l'output sullo schermo:

```
System.out.println("Inserisci un numero intero compreso tra 1 e 99.");
...
System.out.println(cinquantaCent + " monete da cinquanta centesimi");
```

All'inizio di questo capitolo si è detto che `System` è una classe, `out` è un oggetto di questa classe e `println` è un metodo dell'oggetto `out`. Chiaramente non è necessario conoscere tutti questi dettagli per utilizzare un'istruzione di output, basta trattare `System.out.println` come se fosse un'istruzione unica, specifica per l'output.

Per utilizzare istruzioni di output in questa forma, basta specificare dopo l'istruzione `System.out.println` ciò che deve essere visualizzato, racchiuso tra parentesi, con un punto e virgola finale. Si possono visualizzare stringhe di testo tra doppi apici, come `"Inserisci un numero intero compreso tra 1 e 99."`, variabili, come `cinquantaCent`, numeri come `5` o `7.3` e qualsiasi altro oggetto o valore. Per visualizzare più elementi, basta separarli con un segno +. Per esempio,

```
System.out.println("Numero fortunato = " + 13 +
                   "Numero segreto = " + numero);
```

se il valore di `numero` è 7 l'output dell'istruzione sarà:

```
Numero fortunato = 13Numero segreto = 7
```

Si noti che non è stato aggiunto alcuno spazio. Per avere uno spazio tra il numero 13 e parola `Numero` occorre specificarlo all'inizio della stringa:

```
"Numero segreto = "
```

che diventerà:

```
" Numero segreto = "
```

`\"`	Apice doppio.
`\'`	Apice singolo.
`\\`	*Backslash*.
`\n`	Nuova linea. Sposta l'output all'inizio della nuova riga.
`\r`	*Carriage return*. Sposta l'output all'inizio della riga corrente.
`\t`	Tab. Aggiunge spazi bianchi fino al nuovo punto di tabulazione.

Figura 2.6 Caratteri di escape.

È importante notare che ciascuna sequenza di escape rappresenta un solo carattere, anche se viene scritta usando due simboli. La stringa `"\"Ciao\""` non contiene quindi otto caratteri, ma sei: un doppio apice, le lettere C, i, a, o e un altro doppio apice. Comprendere questo aspetto è fondamentale per gestire correttamente gli indici.

Anche l'inclusione di un carattere *backslash* in una stringa è problematica. Per esempio, la stringa `"abc\def"` causa, in fase di compilazione, il seguente messaggio d'errore: "Invalid escape character". Per includere un carattere *backslash* in una stringa è necessario specificare due *backslash*. Pertanto, la stringa `"abc\\def"` visualizzerà il seguente risultato:

```
abc\def
```

Il carattere di escape `\n` indica invece che, nella posizione in cui compare, deve iniziare una nuova riga. Per esempio l'istruzione:

```
System.out.println("Il motto e'\nVincere!")
```

Fa comparire le seguenti due righe sullo schermo:

```
Il motto e'
Vincere!
```

Includere un apice singolo all'interno di una stringa, per esempio `"Java e' bello"`, è perfettamente legale. Se invece si vuole definire una costante contenente un apice singolo occorre utilizzare il carattere di escape `\'`, come nella riga seguente

```
char apiceSingolo ='\'';
```

2.2.6 Set di caratteri Unicode

Un **set di caratteri** è una lista di caratteri a ciascuno dei quali è associato a un numero. Il **set di caratteri ASCII** include tutti i caratteri normalmente usati su una tastiera inglese (ASCII è un acronimo per *American Standard Code for Information Interchange*, letteralmente "codice standard americano per lo scambio di informazioni"). Ciascun carattere ASCII è rappresentato con un numero binario che occupa un solo byte. Questa codifica fornisce quindi 256 caratteri ed è adottata da diversi linguaggi di programmazione.

Il **set di caratteri Unicode** include, oltre all'intero set ASCII, anche i caratteri utilizzati in lingue diverse dall'inglese. Un carattere Unicode occupa 2 byte e pertanto la codifica Unicode fornisce più di 65.000 caratteri differenti.

Al fine di facilitare tutti gli utilizzatori del linguaggio, non solo gli inglesi, gli sviluppatori di Java hanno adottato il set di caratteri Unicode. Il fatto di utilizzare la codifica

Unicode invece che ASCII non ha alcun impatto per chi utilizza una tastiera inglese. In questo caso, infatti, gli sviluppatori possono programmare come se Java utilizzasse la codifica ASCII, che è un sottoinsieme della codifica Unicode. Il vantaggio del set di caratteri Unicode è che permette di gestire con facilità lingue molto diverse dall'inglese. Lo svantaggio principale è invece rappresentato dal fatto che richiede più memoria rispetto ad ASCII per rappresentare i singoli caratteri.

2.3 Operazioni di I/O: la tastiera e lo schermo

Le attività di input e output di un programma sono spesso abbreviate con il temine **I/O**. Un programma Java può svolgere operazioni di I/O in diversi modi. Questo paragrafo descrive alcuni semplici modi per gestire il testo che viene digitato su una tastiera o per visualizzarlo sullo schermo.

2.3.1 Output su schermo

Già i primi esempi presentati in questo testo contenevano alcune istruzioni per la generazione di output sullo schermo. Questo paragrafo riassume e spiega il significato delle istruzioni di output fin qui utilizzate.

Nel Listato 2.3 sono state utilizzate le seguenti istruzioni per inviare l'output sullo schermo:

```
System.out.println("Inserisci un numero intero compreso tra 1 e 99.");
...
System.out.println(cinquantaCent + " monete da cinquanta centesimi");
```

All'inizio di questo capitolo si è detto che `System` è una classe, `out` è un oggetto di questa classe e `println` è un metodo dell'oggetto `out`. Chiaramente non è necessario conoscere tutti questi dettagli per utilizzare un'istruzione di output, basta trattare `System.out.println` come se fosse un'istruzione unica, specifica per l'output.

Per utilizzare istruzioni di output in questa forma, basta specificare dopo l'istruzione `System.out.println` ciò che deve essere visualizzato, racchiuso tra parentesi, con un punto e virgola finale. Si possono visualizzare stringhe di testo tra doppi apici, come `"Inserisci un numero intero compreso tra 1 e 99."`, variabili, come `cinquantaCent`, numeri come `5` o `7.3` e qualsiasi altro oggetto o valore. Per visualizzare più elementi, basta separarli con un segno +. Per esempio,

```
System.out.println("Numero fortunato = " + 13 +
                   "Numero segreto = " + numero);
```

se il valore di `numero` è 7 l'output dell'istruzione sarà:

```
Numero fortunato = 13Numero segreto = 7
```

Si noti che non è stato aggiunto alcuno spazio. Per avere uno spazio tra il numero 13 e la parola `Numero` occorre specificarlo all'inizio della stringa:

```
"Numero segreto = "
```

che diventerà:

Si noti l'utilizzo degli apici doppi (e non singoli) e anche il fatto che gli apici a sinistra e a destra sono lo stesso carattere.

Infine occorre notare che se l'istruzione è troppo lunga, può essere scritta su più righe. Non si può però troncare una riga nel mezzo del nome di una variabile o di una stringa tra apici. Per migliorare la leggibilità del codice è bene andare a capo prima o dopo un operatore + e far rientrare (indentare) la riga successiva.

Il metodo `println` può essere utilizzato anche per visualizzare il valore di una variabile di tipo `String`, come illustrato dalle seguenti istruzioni:

```
String saluto = "Ciao Programmatori!";
System.out.println(saluto);
```

Queste istruzioni provocano la visualizzazione della seguente frase:

```
Ciao Programmatori!
```

Ciascuna invocazione di `println` chiude la riga con un carattere di fine riga.

Si considerino, per esempio, le seguenti istruzioni:

```
System.out.println("Inserisci un intero");
System.out.println("compreso tra 1 e 99.");
```

Queste due istruzioni provocano la visualizzazione delle seguenti righe:

```
Inserisci un intero
compreso tra 1 e 99.
```

Per far sì che più istruzioni di output scrivano sulla stessa riga occorre utilizzare il metodo `print` al posto di `println`. Per esempio le istruzioni:

```
System.out.print("Inserisci");
System.out.print(" un intero");
System.out.println(" compreso tra");
System.out.println("1 e 99.");
```

generano il seguente output:

```
Inserisci un intero compreso tra
1 e 99.
```

Si noti che non viene iniziata una nuova riga finché non viene utilizzato `println` al posto di `print`. Si noti, inoltre, che la nuova riga inizia *dopo* che gli oggetti specificati in `println` sono stati visualizzati. Questa è l'unica differenza tra `print` e `println`.

Le istruzioni fin qui descritte permettono di scrivere programmi che generano questo semplice tipo di output. In realtà, è possibile fare anche qualche cosa di più. Si consideri, per esempio, la seguente istruzione:

```
System.out.println("La risposta e' " + 42);
```

L'espressione all'interno della parentesi:

```
"La risposta e' " + 42
```

dovrebbe risultare familiare. Infatti, nel Paragrafo 2.2 si è detto che l'operatore + può essere utilizzato per concatenare una stringa (per esempio `"La risposta e' "`) con un altro elemento (per esempio la costante numerica 42). L'operatore + all'interno di un'istruzione `System.out.println` è lo stesso operatore che esegue la concatenazione

tra stringhe. Nell'esempio precedente, Java converte la costante numerica 42 nella stringa "42" e quindi utilizza l'operatore + per generare la stringa "La risposta e' 42", che viene poi visualizzata dall'istruzione `System.out.println`. Il metodo `println` visualizza sempre stringhe: tecnicamente non produce mai numeri, ma solo sequenze di caratteri.

`println`

Il metodo `System.out.println` può essere utilizzato per visualizzare righe di testo. Gli elementi visualizzati possono essere stringhe tra apici, variabili, costanti (per esempio numeri) o qualsiasi altro oggetto definibile in Java.

Sintassi

```
System.out.println(output_1 + output_2 + ... + output_n);
```

Esempio

```
System.out.println("Ciao a tutti!");
System.out.println("Area = " + area + " metri quadri.");
```

Usare `println` o `print`?

`System.out.println` e `System.out.print` sono due metodi molto simili, l'unica differenza sta nel fatto che, *dopo* aver visualizzato l'output, il metodo `println` crea una nuova riga di testo. Per esempio le seguenti istruzioni:

```
System.out.print ("Uno ");
System.out.print ("Due ");
System.out.println("Tre ");
System.out.print ("Quattro ");
```

producono il seguente output:

```
Uno Due Tre
Quattro
```

L'output sembrerebbe lo stesso anche nel caso in cui l'ultima istruzione fosse stata `println` invece di `print`. Tuttavia, proprio perché l'ultima istruzione utilizza `print`, l'eventuale successivo output sarà visualizzato sulla stessa riga di `Quattro`.

2.3.2 Input da tastiera

Come già indicato nella prima parte di questo capitolo, la classe `Scanner` consente di gestire l'input da tastiera. Questa classe è fornita con il pacchetto `java.util`. Per poter utilizzare la classe `Scanner`, occorre scrivere nelle prime righe del programma la riga seguente:

```
import java.util.Scanner;
```

Per leggere l'input inserito alla tastiera occorre utilizzare un oggetto della classe `Scanner`. Per creare un oggetto di questo tipo bisogna utilizzare un'istruzione con la seguente forma:

```
Scanner nome_oggetto_scanner = new Scanner(System.in);
```

dove *nome_oggetto_scanner* indica un nome qualsiasi per la variabile di tipo `Scanner`. Per esempio, nel Listato 2.3, per l'oggetto `Scanner` è stato utilizzato l'identificatore `tastiera`, che suggerisce il fatto che l'input proviene dalla tastiera. Naturalmente è possibile utilizzare altri nomi come, per esempio, `oggettoScanner`.

Dopo aver definito un oggetto `Scanner`, per leggere i valori digitati alla tastiera si possono utilizzare i metodi di tale classe. Per esempio, l'invocazione del metodo:

```
tastiera.nextInt()
```

legge e restituisce un valore di tipo `int` digitato sulla tastiera. Il valore restituito può essere assegnato a una variabile di tipo `int` come segue:

```
int n1 = tastiera.nextInt();
```

Per leggere dati di altro tipo occorre utilizzare appositi metodi. Per esempio, il metodo `nextDouble` si comporta come `nextInt`, con l'unica differenza che legge un valore di tipo `double`. La classe `Scanner` presenta metodi analoghi per leggere anche altri tipi di valori numerici.

Il metodo `next` legge una parola, come mostrato dalle seguenti istruzioni:

```
String s1 = tastiera.next();
String s2 = tastiera.next();
```

Se l'input fosse il seguente:

```
forchette coltelli
```

alla variabile `s1` verrebbe assegnata la stringa "`forchette`" e alla variabile `s2` verrebbe assegnata la stringa "`coltelli`". Si noti che i valori digitati alla tastiera dovrebbero essere separati da un carattere di spaziatura, per esempio uno o più spazi, uno o più caratteri di fine riga oppure una loro combinazione. In questo contesto, i caratteri di separazione sono detti **delimitatori** (*delimiters*). Per il metodo `next`, una *parola* corrisponde a una qualsiasi stringa di caratteri che non contiene caratteri di spaziatura (che verrebbero considerati delimitatori).

Per leggere un'intera riga occorre invece utilizzare il metodo `nextLine`. Per esempio, l'istruzione:

```
String frase = tastiera.nextLine();
```

legge una riga di input e inserisce la stringa nella variabile `frase`. La fine di una riga di input è indicata dal carattere di escape `'\n'`, digitato nel momento in cui si preme il tasto Invio (*Enter* o *Return*) sulla tastiera. Sullo schermo viene visualizzato semplicemente l'inizio della nuova riga (il cursore va a capo). Quando `nextLine` legge una riga di testo, trova il carattere `'\n'`, ma non lo inserisce nella stringa restituita come risultato. Nell'esempio precedente, perciò, la stringa assegnata alla variabile `frase` non termina con il carattere `'\n'`. Il Listato 2.5 mostra un programma che illustra l'utilizzo dei metodi della classe `Scanner` descritti in questo paragrafo.

LISTATO 2.5 Un esempio di input da tastiera.

MyLab

```
import java.util.Scanner;
```
← Carica la classe `Scanner` dal package `java.util`

```
public class ScannerDemo {

    public static void main(String[] args) {

        Scanner tastiera = new Scanner(System.in);
```
← Inizializza gli oggetti in modo che il programma possa leggere l'input da tastiera

```
        System.out.println("Digita due numeri interi");
        System.out.println("separati da uno o piu' spazi:");

        int n1, n2;
        n1 = tastiera.nextInt();
```
← Legge un valore di tipo `int` dalla tastiera

```
        n2 = tastiera.nextInt();
        System.out.println("Hai digitato " + n1 + " e " + n2);

        System.out.println("Ora digita altri due numeri.");
        System.out.println("E' ammesso anche il separatore decimale.");

        double d1, d2;
        d1 = tastiera.nextDouble();
```
← Legge un valore di tipo `double` dalla tastiera

```
        d2 = tastiera.nextDouble();
        System.out.println("Hai digitato " + d1 + " e " + d2);

        System.out.println("Ora digita due parole:");

        String s1, s2;
        s1 = tastiera.next();
```
← Legge una parola dalla tastiera

```
        s2 = tastiera.next();
        System.out.println("Hai digitato \"" +
            s1 + "\" e \"" + s2 + "\"");

        s1 = tastiera.nextLine(); //Necessario per gestire il '\n'

        System.out.println("Digita ora una riga di testo:");
        s1 = tastiera.nextLine();
```
← Legge un'intera riga

```
        System.out.println("Hai digitato: \"" + s1 + "\"");
    }
}
```

Questa riga è spiegata nel prossimo box "Problemi comuni con i metodi `next` e `nextline`"

Esempio di output

```
Digita due numeri interi
Separati da uno o più spazi:
42 43
Hai digitato 42 e 43
Ora digita altri due numeri.
E' ammesso anche il separatore decimale.
9.99 28
```

```
Hai digitato 9.99 e 21.0
Ora digita due parole:
forchette coltelli
Hai digitato "forchette" e "coltelli"
Digita ora una riga di testo:
Ho imparato l'input da tastiera.
Hai digitato "Ho imparato l'input da tastiera."
```

Input da tastiera con la classe `Scanner`

Un oggetto della classe `Scanner` consente di leggere l'input dalla tastiera. Per utilizzare questi oggetti occorre inserire la seguente istruzione all'inizio del programma:

```
import java.util.Scanner;
```

Inoltre, prima di poter leggere l'input, occorre specificare un'istruzione simile alla seguente:

```
Scanner nome_oggetto_scanner = new Scanner(System.in);
```

dove il termine *nome_oggetto_scanner* indica un qualsiasi identificatore Java che non sia una parola chiave, per esempio:

```
Scanner tastiera = new Scanner(System.in);
```

I metodi `nextInt`, `nextDouble` e `next` leggono e restituiscono rispettivamente un valore di tipo `int`, di tipo `double` e una parola (cioè un oggetto di tipo `String`). Il metodo `nextLine` legge la parte rimanente della stringa corrente e la restituisce come oggetto di tipo `String`. Il carattere di terminazione `'\n'` viene letto, ma non viene incluso nel valore di tipo stringa restituito.

Sintassi

```
variabile_int = nome_oggetto_scanner.nextInt();
variabile_double = nome_oggetto_scanner.nextDouble();
variabile_string = nome_oggetto_scanner.next();
variabile_string = nome_oggetto_scanner.nextLine();
```

Esempi

```
int somma = tastiera.nextInt();
double distanza = tastiera.nextDouble();
String parola = tastiera.next();
String interaRiga = tastiera.nextLine();
```

La Figura 2.7 mostra un elenco di metodi offerti dalla classe `Scanner`.

Richiesta di input (Prompt for input)

Un programma deve sempre mostrare una **frase di richiesta** (comunemente detta *prompt* in inglese) che spieghi all'utente quali dati deve digitare, per esempio:

```
System.out.println("Digita un numero intero:");
```

Problemi comuni con i metodi `next` e `nextLine`

I metodi `next` e `nextLine` della classe `Scanner` leggono il testo a partire dall'ultima posizione raggiunta dall'ultimo comando di lettura. Per esempio, si supponga di creare un oggetto `Scanner` come segue:

```
Scanner tastiera = new Scanner(System.in);
```

e si supponga di proseguire con il seguente codice:

```
int n = tastiera.nextInt();
String s1 = tastiera.nextLine();
String s2 = tastiera.nextLine();
```

Infine, si supponga che venga digitato il seguente input:

```
42 e' la risposta
e non lo
dimenticare.
```

Alla variabile `n` verrà assegnato il numero 42, alla variabile `s1` la stringa `"e' la risposta"` e alla variabile `s2` la stringa `"e non lo"`.
Quanto accade è, sostanzialmente, quello che ci si aspetta. Si supponga, invece, di aver digitato il seguente input:

```
42
e non lo
dimenticare.
```

In queste circostanze, ci si potrebbe aspettare che 42 venga assegnato a `n`, che la stringa `"e non lo"` venga assegnata a `s1` e che la stringa `"dimenticare."` venga assegnata a `s2`. Quello che succede in realtà è che alla variabile `n` viene effettivamente assegnato il valore 42, ma alla variabile `s1` viene assegnata una stringa vuota e alla variabile `s2` viene assegnata la stringa `"e non lo"`. Questo è dovuto al fatto che il metodo `nextInt` legge il valore 42, ma non legge il carattere di fine riga `'\n'`. Per questo motivo l'invocazione successiva di `nextLine` legge la parte rimanente della riga sulla quale si trova il numero 42. Su quella riga c'è ancora qualcosa, il carattere `'\n'`, e quindi `nextLine` legge e scarta, come previsto, il carattere `'\n'` e restituisce una stringa vuota. Infine la seconda invocazione di `nextLine` riparte dalla riga successiva e legge `"e non lo"`. Quando si combinano metodi che leggono numeri dalla tastiera e metodi che leggono stringhe, spesso è necessario includere un'invocazione aggiuntiva a `nextLine` per superare il problema del carattere di fine riga `'\n'`. Le ultime istruzioni del programma nel Listato 2.5 mostrano proprio un esempio di questo tipo.

La stringa vuota

Si ricordi che la stringa vuota contiene zero caratteri e viene scritta come `""`. Se il programma esegue il metodo `nextLine` e l'utente preme semplicemente il tasto Invio (*Return*), il metodo `nextLine` restituisce la stringa vuota.

nome_oggetto_scanner`.next()` Restituisce un valore `String` che corrisponde al prossimo input da tastiera fino al primo carattere di delimitazione escluso. I caratteri di delimitazione di default sono i caratteri di spaziatura.
nome_oggetto_scanner`.nextLine()` Legge la parte rimanente della riga corrente e restituisce i caratteri letti come un valore di tipo `String`. Si noti che il carattere di terminazione di riga `'\n'` viene letto, ma scartato. Infatti non viene incluso nella stringa restituita.
nome_oggetto_scanner`.nextInt()` Legge il prossimo input da tastiera come un valore di tipo `int`.
nome_oggetto_scanner`.nextDouble()` Legge il prossimo input da tastiera come un valore di tipo `double`.
nome_oggetto_scanner`.nextFloat()` Legge il prossimo input da tastiera come un valore di tipo `float`.
nome_oggetto_scanner`.nextLong()` Legge il prossimo input da tastiera come un valore di tipo `long`.
nome_oggetto_scanner`.nextByte()` Legge il prossimo input da tastiera come un valore di tipo `byte`.
nome_oggetto_scanner`.nextShort()` Legge il prossimo input da tastiera come un valore di tipo `short`.
nome_oggetto_scanner`.nextBoolean()` Legge il prossimo input da tastiera come un valore di tipo `boolean`. I valori *true* e *false* devono essere scritti proprio come `true` e `false`. Viene accettata qualsiasi combinazione di lettere maiuscole e minuscole.
nome_oggetto_scanner`.useDelimiter(`*parola_di_delimitazione*`)` Fa sì che la stringa *parola_di_delimitazione* sia l'unico delimitatore utilizzato per separare l'input. Solo le stringhe corrispondenti a questa parola saranno considerate delimitatori. In particolare, i caratteri spazio, nuova riga e gli altri caratteri di spaziatura non saranno più considerati delimitatori, a meno che non facciano parte della parola *parola_di_delimitazione*. Questo è un semplice esempio d'uso del metodo `useDelimiter`. Ci sono diversi modi per cambiare i delimitatori, che tuttavia non verranno presentati in questo testo.

Figura 2.7 Alcuni metodi della classe `Scanner`.

2.3.3 Altri delimitatori di input (opzionale)

I delimitatori utilizzati dalla classe `Scanner` per gestire l'input da tastiera possono essere modificati durante l'esecuzione del programma. Sebbene siano possibili diverse modalità per gestire i delimitatori, il testo illustra solo un caso semplice: i delimitatori predefiniti (i caratteri di spaziatura) vengono sostituiti da una stringa di delimitazione a scelta.

Si supponga, per esempio, di creare un oggetto `Scanner`:

```
Scanner tastiera2 = new Scanner(System.in);
```

Il delimitatore per `tastiera2` può essere cambiato con la stringa `"##"` nel seguente modo:

```
tastiera2.useDelimiter("##");
```

Dopo l'invocazione del metodo `useDelimiter`, la stringa `"##"` sarà l'**unico** delimitatore di input per l'oggetto `tastiera2`. Si noti che i caratteri di spaziatura non saranno più considerati delimitatori per l'input da tastiera gestito con `tastiera2`. In altre parole, se viene passato il seguente input:

```
Esempio diver##tente
```

Il codice seguente leggerà le due stringhe `"Esempio diver"` e `"tente"`.

```
System.out.println("Si inseriscano due parole su una riga:")
String s1 = tastiera2.next();
String s2 = tastiera2.next();
```

Si noti che nessun carattere di spaziatura, nemmeno i ritorni a capo (\n), verranno utilizzati come delimitatori per l'input. Si noti inoltre che nello stesso programma è possibile utilizzare due diversi oggetti di classe `Scanner` che utilizzano delimitatori differenti per l'input. Il Listato 2.6 ne mostra un esempio.

LISTATO 2.6 Cambiare i delimitatori (opzionale).

MyLab

```
import java.util.Scanner;

public class DelimitatoriDemo {

   public static void main(String[] args) {

      Scanner tastiera1 = new Scanner(System.in);
      Scanner tastiera2 = new Scanner(System.in);
      tastiera2.useDelimiter("##");
      //I delimitatori di tastiera1 sono i caratteri di spaziatura.
      //L'unico delimitatore di tastiera2 è la stringa ##.

      String s1, s2;

      System.out.println("Scrivi una riga di testo con due parole:");
      s1 = tastiera1.next();
      s2 = tastiera1.next();
      System.out.println("Le due parole sono \"" + s1 +
                         "\" e \"" + s2 + "\"");

      System.out.println("Scrivi una riga di testo con due parole");
      System.out.println("delimitate da ##:");
      s1 = tastiera2.next();
      s2 = tastiera2.next();
```

tastiera1 e tastiera2 usano delimitatori differenti.

```
        System.out.println("Le due parole sono \"" + s1 +
                           "\" e \"" + s2 + "\"");
    }
}
```

Esempio di output

```
Scrivi una riga di testo con due parole:
esempio diver##tente##
Le due parole sono "esempio" e "diver##tente##"
Scrivi una riga di testo con due parole
delimitate da ##
esempio diver##tente##
Le due parole sono "esempio diver" e "tente"
```

2.3.4 Output formattato con `printf`

A partire dalla versione 5, Java include un metodo, chiamato `printf`, che può essere utilizzato per generare un output in un formato specifico. Esso è utilizzato nello stesso modo della funzione `printf` presente nel linguaggio di programmazione C.

Il metodo `printf` funziona in modo simile al metodo `print`, ma, a differenza di quest'ultimo, consente di aggiungere informazioni di formattazione per specificare aspetti come il numero di cifre da includere dopo il punto decimale. Per esempio, si consideri il codice seguente:

```
double prezzo = 19.5;
System.out.println("Prezzo stampato con println:" + prezzo);
System.out.printf("Prezzo stampato con la formattazione
                  di printf:%6.2f", prezzo);
```

Questo codice produce le linee seguenti:

```
Prezzo stampato con println:19.5
Prezzo stampato con la formattazione di printf: 19.50
```

Se si utilizza `println` il prezzo è stampato come "`19.5`" subito dopo i due punti, dato che non è stato aggiunto alcuno spazio. Se si utilizza `printf`, la stringa dopo i due punti è " `19.50`" con uno spazio prima di `19.50`. In questo semplice esempio, il primo argomento di `printf` è una stringa denominata **specifica di formato**, mentre il secondo è il numero o in generale il valore da stampare in quel formato.

La specifica di formato `%6.2f` indica di produrre un numero in virgola mobile in un **campo** (numero di caratteri) di dimensione sei (quindi in uno spazio atto a contenere fino a sei caratteri) mostrando esattamente due cifre dopo il punto decimale. Di conseguenza, `19.5` è formattato come "`19.50`" in un campo di dimensione sei. Dato che la stringa "`19.50`" è composta da soli cinque caratteri, viene aggiunto all'inizio uno spazio vuoto per ottenere la stringa da sei caratteri " `19.50`". Gli altri eventuali spazi sono quindi riportati prima del risultato. Se il valore da formattare richiede più caratteri di quelli specificati nel campo (ad esempio, se nel caso precedente la dimensione del campo fosse fissata a uno utilizzando la specifica `%1.2f`), quest'ultimo è allargato automaticamente fino all'esatta dimensione dell'output. Nell'esempio, il campo avrebbe avuto dimensione cinque e si sarebbe ottenuta la stringa "`19.50`".

Specifica di formato	Tipo del valore da formattare	Esempi
`%c`	Carattere	Un carattere singolo: `%c`
		Un carattere singolo in un campo di lunghezza 2: `%2c`
`%d`	Numero intero	Un intero: `%d`
		Un intero in un campo di lunghezza 5: `%5d`
`%f`	Numero in virgola mobile	Un numero in virgola mobile: `%f`
		Un numero in virgola mobile con due cifre dopo il punto decimale: `%1.2f`
		Un numero in virgola mobile con due cifre dopo il punto decimale e in un campo di lunghezza sei: `%6.2f`
`%e`	Numero in virgola mobile in notazione scientifica	Un numero in virgola mobile in notazione scientifica: `%e`
`%s`	Stringa	Una stringa formattata in un campo di lunghezza dieci: `%10s`

Figura 2.8 Alcune specifiche di formato per il metodo `System.out.printf`.

Infine, il carattere `f` in `%6.2f` indica che il valore da formattare è un numero in virgola mobile, cioè un numero con il punto decimale.

La Figura 2.8 riassume le specifiche di formato più comunemente utilizzate. È possibile combinare più specifiche di formato nella stessa stringa. Per esempio, date le istruzioni

```
double prezzo = 19.5;
int quantita = 2;
String elemento = "Oggetti";
System.out.printf("%10s venduti:%4d a €%5.2f. Totale = €%1.2f", elemento,
                  quantita, prezzo, quantita * prezzo);
```

il risultato è: " `Oggetti venduti:  2 a €19.50. Totale = €39.00`". Ci sono tre spazi prima della parola "`Oggetti`" in modo da ottenere un campo di lunghezza dieci. Analogamente, ci sono tre spazi prima del 2 per avere un campo da quattro caratteri. Il `19.50` occupa esattamente i cinque caratteri, mentre l'ultimo campo per il totale è esteso automaticamente da uno a cinque caratteri affinché possa contenere il valore `39.00`.

2.4 Documentazione e stile

Un programma che produce un output corretto non è necessariamente un buon programma. Naturalmente è necessario che il programma generi un output corretto, tuttavia occorre fare di più. Molti programmi vengono riutilizzati più volte, modificandoli per correggere eventuali difetti (detti anche *bug*) o per rispondere alle nuove richieste degli utilizzatori. Se il programma non è di facile comprensione, sarà difficile o addirittura impossibile da modificare senza uno sforzo considerevole.

Anche se il programma verrà utilizzato una sola volta, è bene curare la sua leggibilità. Questo paragrafo presenta quattro aspetti che migliorano la leggibilità di un programma: nomi significativi, commenti, indentazione e nomi di costanti.

2.4.1 Nomi significativi per le variabili

Come indicato all'inizio del capitolo, i nomi x e y non rappresentano mai una buona scelta per le variabili. Il nome assegnato a una variabile dovrebbe suggerire lo scopo per cui viene utilizzata. Se la variabile contiene la somma di alcuni valori, il suo nome potrebbe essere `somma`. Se contiene un tasso d'interesse, il nome potrebbe essere `tassoInteresse`.

Oltre ad assegnare alle variabili un nome significativo e accettabile per il compilatore, è anche necessario scegliere un nome che segua le pratiche comunemente adottate dai programmatori. In questo modo il codice risulterà più semplice da leggere e da riutilizzare da parte di altri sviluppatori. Tipicamente i nomi delle variabili sono interamente costituiti da lettere e cifre. Il nome di una variabile dovrebbe iniziare con una lettera minuscola. Questa è una convenzione comunemente adottata dai programmatori Java. I nomi che iniziano con una lettera maiuscola sono adottati per altri scopi, per esempio per le classi, come `String`. Se un nome è costituito da più parole, queste possono essere evidenziate con l'iniziale maiuscola, come in `tassoInteresse`, `numeroDiProve` o `tempoResiduo`.

2.4.2 Commenti

La documentazione di un programma indica gli scopi e il funzionamento del programma. I programmi ben realizzati sono **auto-esplicativi** (*self-documenting*). Questo vuol dire che, grazie a uno stile di programmazione pulito e a una scelta oculata dei nomi degli identificatori, qualsiasi programmatore dovrebbe essere in grado di comprendere il funzionamento del programma semplicemente leggendolo. Sebbene sia utile sforzarsi di scrivere programmi auto-esplicativi, spesso è necessario aggiungere alcune spiegazioni per chiarire il funzionamento del programma. Queste spiegazioni possono essere espresse sotto forma di commenti.

I **commenti** sono note scritte all'interno del programma per facilitarne la comprensione, ma che vengono ignorate dal compilatore. Vi sono tre modi per inserire commenti nei programmi Java. Il primo prevede l'uso della sequenza `//` all'inizio di un commento. Tutto ciò che segue questi simboli fino alla fine della riga è considerato un commento e viene ignorato dal compilatore. Questa tecnica è utile per commenti brevi, per esempio nell'istruzione seguente:

```
String frase; //Versione italiana
```

Per estendere un commento di questo tipo su più righe, ciascuna riga deve iniziare con la sequenza `//`.

Il secondo modo permette di scrivere commenti che si estendono su più righe. Tutto quello che viene incluso tra la coppia di simboli `/*` e `*/` viene considerato un commento e viene pertanto ignorato dal compilatore. Per esempio,

```
/*
Questo programma mostra
come possono essere configurati
i delimitatori nella classe Scanner
```

Molti editor di testi evidenziano i commenti con un colore differente dal resto del programma, proprio per renderli facilmente identificabili.

Si consideri il seguente commento:

```
/**
Questo programma mostra
come possono essere configurati
i delimitatori nella classe Scanner
*/
```

*Si noti che questo commento, a differenza del precedente, utilizza due asterischi invece di uno all'inizio del commento (/**). Questa sequenza è necessaria quando si adotta il programma* `javadoc` *per estrarre in modo automatico la documentazione di un programma Java. Il programma* `javadoc` *verrà presentato in seguito, tuttavia il doppio asterisco sarà utilizzato fin dai primi esempi di commento presentati.*

Spiegare quando inserire e quando non inserire un commento non è semplice. Troppi commenti possono essere altrettanto dannosi quanto troppo pochi. Con troppi commenti un'informazione davvero rilevante si può perdere in un mare di commenti ovvi. Per ogni funzionalità Java descritta in questo testo sarà indicata la posizione in cui è opportuno inserire commenti. Per ora i commenti sono utili in due sole situazioni.

In primo luogo, ogni programma dovrebbe presentare all'inizio un commento esplicativo. Questo commento dovrebbe fornire tutte le informazioni utili sul file: cosa fa il programma, il nome dell'autore, come contattarlo e la data in cui il file è stato modificato per l'ultima volta, più altre informazioni che dipendono dal contesto in cui è stato creato il programma, per esempio il numero dell'esercitazione nel caso di un programma creato per un corso di programmazione. Questo commento dovrebbe essere simile a quello mostrato all'inizio del Listato 2.7.

In secondo luogo, i commenti dovrebbero spiegare tutti i dettagli non ovvi. Si osservi, per esempio, il programma del Listato 2.7. Si noti la presenza delle due variabili `raggio` *e* `area`*. Ovviamente queste due variabili conterranno rispettivamente i valori del raggio e dell'area di un cerchio. Non si dovrebbero includere commenti come il seguente:*

```
double raggio;   //il raggio del cerchio
```

Risulta, invece, necessario includere informazioni non ovvie, per esempio l'unità di misura del raggio.

```
double raggio;   //in metri
double area;     //in metri quadri
```

Questi due commenti sono presenti nel Listato 2.7.

Scrivere codice auto-esplicativo

Un frammento di codice **auto-esplicativo** *fa uso di nomi ben selezionati e presenta uno stile chiaro. Lo scopo del programma e il suo funzionamento dovrebbero essere chiari per ogni programmatore che legge il programma, anche se il programma non contiene commenti. Sarebbe opportuno che ogni programmatore si sforzasse, per quanto possibile, di rendere auto-esplicativi i propri programmi.*

MyLab

LISTATO 2.7 Commenti e indentazione.

```
import java.util.Scanner;
/**
Programma che calcola l'area di un cerchio.
Autore: Paola M. Programmatore
E-mail: paolam@qualchemacchina.etc.etc.

Esercitazione 2.

Ultima modifica: 17 Marzo 2013.
*/
public class MisuraCerchio {
    public static void main(String[] args) {
        double raggio; //in metri
        double area; //in metri quadri
        Scanner tastiera = new Scanner(System.in);

        System.out.println("Scrivi il raggio del cerchio in metri:");
        raggio = tastiera.nextDouble();
        area = 3.14159 * raggio * raggio;
        System.out.println("Un cerchio di raggio " + raggio + " metri");
        System.out.println("ha un'area di " + area + " metri quadri.");
    }
}
```

L'istruzione di `import` può anche essere posta dopo il commento al programma.

Le linee verticali indicano il modello di indentazione.

Più avanti in questo capitolo sarà presentata una versione avanzata di questo programma.

Esempio di output

```
Scrivi il raggio del cerchio in metri:
2.5
Un cerchio di raggio 2.5 metri
ha un'area di 19.6349375 metri quadri.
```

Commenti in Java

Ci sono tre modi per aggiungere commenti in Java.

- *Tutto ciò che segue i simboli* `//` *fino alla fine della riga è un commento e viene ignorato dal compilatore.*
- *Tutto ciò che viene scritto tra la coppia di simboli* `/*` *e* `*/` *è un commento e viene ignorato dal compilatore.*

Tutto ciò che viene scritto tra la coppia di simboli `/**` *e* `*/` *è un commento che viene elaborato dal programma* `javadoc`*, ma che viene ignorato dal compilatore.*

2.4.3 Indentazione

Un programma è composto da varie parti; in particolare ci sono parti più piccole contenute all'interno di parti più grandi. Per esempio, una parte potrebbe iniziare con:

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

Il corpo del metodo `main` comincia con una parentesi graffa aperta, {, e termina con una parentesi graffa chiusa, }. All'interno di queste parentesi si trovano varie istruzioni Java che si indentano (*indent* in inglese) utilizzando un numero appropriato di spazi.

Il programma del Listato 2.7 presenta tre livelli di indentazione, come indicato dalle linee verticali, che evidenziano la **struttura annidata** del programma. Il livello più esterno, che definisce la classe `MisuraCerchio`, non è indentato. Il livello successivo, che corrisponde al metodo `main`, è stato indentato di una posizione. Infine, il corpo del metodo `main` è stato indentato di due posizioni.

Si è soliti utilizzare un'indentazione di quattro spazi per ciascun livello. Se si utilizzasse un numero maggiore di spazi, resterebbe troppo poco spazio per l'istruzione, mentre un'indentazione minore sarebbe poco visibile. Indentare di due o tre spazi potrebbe essere comunque sensato, tuttavia quattro spazi risultano spesso più chiari. In un corso di programmazione le regole da utilizzare per le dimensioni delle singole indentazioni sono fornite dal docente, mentre nei progetti software sono definite da regole condivise all'interno del team. In generale è importante mantenere un numero costante di rientri all'interno del programma.

Se un'istruzione non può essere contenuta in una sola riga, può essere scritta su due o più righe. Tuttavia, quando si è costretti a scrivere un'istruzione su due o più righe è bene indentare la parte rimanente dell'istruzione.

Nel Listato 2.7, i livelli di annidamento sono delimitati dalle parentesi graffe. L'utilizzo delle parentesi graffe per delimitare i livelli di annidamento non è una regola. È comunque buona norma, anche in assenza delle parentesi graffe, indentare ogni livello di annidamento del programma.

2.4.4 Utilizzare le costanti con nome

Si osservi il programma nel Listato 2.7. Il numero 3.14159 è l'approssimazione del valore *pi greco*, un numero utilizzato in vari calcoli che riguardano i cerchi e che spesso è scritto come π. In generale è difficile capire il significato di un valore costante inserito nel codice. Per esempio, un altro programmatore potrebbe non capire quale sia l'origine del numero 3.14159. Per evitare questa confusione è bene dare un nome a queste costanti e utilizzare tale nome invece di scrivere direttamente il numero. Per esempio, al numero 3.14159 si potrebbe assegnare il nome `PI`, come avviene nell'istruzione seguente:

```
public static final double PI = 3.14159;
```

L'istruzione di assegnamento:

```
area = 3.14159 * raggio * raggio;
```

risulterebbe più chiara se venisse scritta come:

Il Listato 2.8 è una rivisitazione del Listato 2.7, in cui al posto del valore 3.14159 viene utilizzato il nome PI. Si noti che la definizione di PI si trova al di fuori del metodo main. Sebbene non sia necessario, è buona pratica porre le definizioni delle costanti all'inizio del file. In questo modo risulta più semplice modificarne il valore, se necessario.

Si supponga, per esempio, che in un programma bancario che contiene la seguente costante

```
public static final double TASSO_INTERESSE_PASSIVO = 6.99;
```

il tasso di interesse passi al 8.5%. Per modificare il programma è sufficiente modificare il valore della costante nel seguente modo:

```
public static final double TASSO_INTERESSE_PASSIVO = 8.5;
```

Questa modifica richiede solo la ricompilazione del programma, ma non rende necessario nessun altro cambiamento in esso.

Utilizzare una costante come TASSO_INTERESSE_PASSIVO, può far risparmiare molto lavoro. Infatti, per cambiare il valore del tasso di interesse passivo da 6.99 a 8.5 è necessario modificare un solo numero. Se non fosse stata utilizzata una costante con nome, sarebbe stato necessario modificare tutte le occorrenze di 6.99 presenti nel programma. Sebbene un editor di testo faciliti questa operazione, per esempio sostituendo automaticamente tutte le occorrenze di 6.99, la modifica potrebbe introdurre errori. Per esempio, alcune occorrenze del numero 6.99 rappresenterebbero certamente il tasso di interesse passivo, ma altre potrebbero avere un altro significato: il programmatore sarebbe pertanto costretto a distinguere manualmente questi due casi volta per volta. Chiaramente questa operazione potrebbe generare confusione e portare a introdurre errori nel programma.

MyLab

LISTATO 2.8 Dare nomi alle costanti.

```
import java.util.Scanner;

/**
Programma che calcola l'area di un cerchio.
Autore: Paola M. Programmatore
E-mail: paolam@qualchemacchina.etc.etc.

Esercitazione 2.

Ultima modifica: 17 Marzo 2013.
*/

public class MisuraCerchio2 {

    public static final double PI = 3.14159;

    public static void main(String[] args) {

        double raggio; //in metri
        double area; //in metri quadri
        Scanner tastiera = new Scanner(System.in);
```

```
        System.out.println("Scrivi il raggio del cerchio in metri:");
        raggio = tastiera.nextDouble();
        area = PI * raggio * raggio;
        System.out.println("Un cerchio di raggio " + raggio + " metri");
        System.out.println("ha un area di " + area + " metri quadri.");
    }
}
```

Sebbene sia meno chiaro, è possibile mettere la definizione di `PI` anche in questa posizione.

Esempio di output

```
Scrivi il raggio del cerchio in metri:
2.5
Un cerchio di raggio 2.5 metri
ha un area di 19.6349375 metri quadri.
```

2.5 Riepilogo

- Una variabile può contenere un valore, per esempio un numero. Il tipo della variabile deve corrispondere al tipo del valore memorizzato al suo interno.
- Alle variabili e agli altri elementi di un programma dovrebbe essere assegnato un nome che ne rappresenti il significato. In Java questi nomi sono chiamati identificatori.
- A tutte le variabili deve essere assegnato un valore iniziale prima che possano essere utilizzate in un programma. Questo può essere fatto con un'istruzione di assegnamento, eventualmente combinata con la dichiarazione della variabile.
- Le parentesi nelle espressioni aritmetiche indicano l'ordine di esecuzione delle operazioni.
- Quando si assegna un valore a una variabile con un'istruzione di assegnamento, il tipo della variabile deve essere compatibile con il tipo del valore. In caso contrario è necessario eseguire una conversione di tipo.
- I metodi della classe `Scanner` consentono di leggere l'input dalla tastiera.
- È bene che il programma visualizzi un messaggio all'utente ogni volta che è richiesto un input da tastiera.
- Il metodo `println`, a differenza di `print`, introduce una nuova riga dopo aver visualizzato l'output.
- Si può utilizzare il metodo `printf` per ottenere un output formattato.
- Si possono avere sia variabili e sia valori costanti di tipo `String`.
- `String` è una classe che si comporta in modo analogo a un tipo primitivo.
- Il simbolo + può essere utilizzato anche per indicare la concatenazione di due stringhe.
- La classe `String` offre metodi per l'elaborazione delle stringhe.
- È bene definire dei nomi per le costanti numeriche di un programma e utilizzare tali nomi al posto del numero stesso.

- I programmi dovrebbero essere il più possibile auto-esplicativi. Tuttavia si possono anche inserire dei commenti, per spiegare i punti meno chiari. Per specificare un commento di una sola riga si utilizza il simbolo //; per scrivere commenti di più righe si possono utilizzare le coppie di simboli `/**` e `*/` oppure `/*` e `*/`.

2.6 Esercizi

1. Si scriva un programma che dimostri la natura approssimativa dei numeri in virgola mobile effettuando le seguenti attività.
   - Utilizzare `Scanner` per leggere un numero in virgola mobile *x*.
   - Calcolare 1.0 / *x* e memorizzare il risultato in *y*.
   - Visualizzare *x*, *y* e il prodotto di *x* e *y*.
   - Sottrarre *x* dal prodotto di *x* e *y* e mostrarne il risultato.

   Si provi a eseguire il programma con valori di *x* che vanno da `2e-11` a `2e11` e si traggano delle conclusioni.
2. Si scriva un programma che dimostri la conversione di tipo per valori `double`, effettuando le seguenti attività.
   - Utilizzare `Scanner` per leggere un numero in virgola mobile *x*.
   - Convertire *x* in un valore intero e memorizzare il risultato in *y*.
   - Visualizzare in maniera distinta *x* e *y*.
   - Convertire *x* in un valore di tipo `byte` e memorizzare il risultato in *z*.
   - Visualizzare in maniera distinta *x* e *z*.

   Si esegua il programma con valori positivi e negativi di *x* che vanno da `2e-11` a `2e11` e si traggano delle conclusioni.
3. Si scriva un programma che dimostri le funzionalità dell'operatore % effettuando le seguenti attività.
   - Utilizzare `Scanner` per leggere un numero in virgola mobile *x*.
   - Calcolare *x* % 2.0 e memorizzare il risultato in *y*.
   - Visualizzare in maniera distinta *x* e *y*.
   - Effettuare una conversione di tipo di *x* in un valore `int` e memorizzare il risultato in *z*.
   - Visualizzare in maniera distinta *x*, *z* e *z* % 2 opportunamente etichettati.

   Si esegua il programma con valori positivi e negativi di *x*. Che cosa cambia nel comportamento dell'applicazione quando i valori di *x* sono positivi o negativi?
4. Se `u = 2, v = 3, w = 5, x = 7` e `y = 11`, qual è il valore di ciascuna delle seguenti espressioni, supponendo che si tratti di valori di tipo `int`?
   - `u + v * w + x`
   - `u + y % v * w + x`
   - `u++ / v + u++ * w`

5. Quali cambiamenti sono necessari nel programma del Listato 2.3 per far sì che accetti anche monete da 1 Euro?
6. Se la variabile `int` di nome `x` contiene il valore `10`, che cosa visualizzeranno le seguenti istruzioni?

```
System.out.println("Test 1" + x * 3 * 2.0);
System.out.println("Test 2" + x * 3 + 2.0);
```

Si spieghi perché la seguente istruzione non viene compilata:

```
System.out.println("Test 3" + x * 3 - 2.0);
```

7. Si scrivano delle istruzioni Java che utilizzino i metodi `indexOf` e `substring` della classe `String` per trovare la prima parola di una stringa. Per parola si intende una stringa di caratteri senza spazi. Per esempio la prima parola della stringa `"Ciao, mio caro!"` è la stringa `"Ciao"`, mentre la seconda è `"mio"`.
8. Si ripeta l'esercizio precedente cercando la seconda parola.
9. Che cosa visualizza la seguente istruzione Java?

```
System.out.println("\"\tTest\\\\\rIt\'");
```

Sostituendo la lettera `r` con la lettera `n`, che effetto sia ha sull'output?

10. Si scriva una e una sola istruzione Java che visualizzi le parole "uno", "due" e "tre" su tre righe distinte.
11. Che cosa visualizza il seguente codice Java:

```
Scanner tastiera = new Scanner(System.in);
System.out.println("Inserisci una stringa.");
int n = tastiera.nextInt();
String s = tastiera.next();
System.out.println("n e' " + n);
System.out.println("s e' " + s);
```

se l'input da tastiera è:

```
2e'l'input
```

12. Che cosa visualizza a schermo il seguente codice Java:

```
Scanner tastiera = new Scanner(System.in);
tastiera.useDelimiter("i");
System.out.println("Scrivi una stringa.");
String a = tastiera.next();
String b = tastiera.next();
System.out.println("a e' " + a);
System.out.println("b e' " + b);
```

se l'input da tastiera è:

```
Ciao a tutti
```

13. Si ripeta l'esercizio precedente sostituendo `next` con `nextLine` nell'istruzione che assegna un valore a `b`.

14. Diversi sport hanno delle costanti nelle proprie regole. Per esempio, il baseball ha 9 inning, 3 out per inning, 3 strike in un fuori out e 4 palle (*balls*) per corsa (*walk*). È possibile codificare le costanti per un programma che riguarda il baseball nel seguente modo:

```
public static final int INNINGS = 9;
public static final int OUTS_PER_INNING = 3;
public static final int STRIKES_PER_OUT = 3;
public static final int BALLS_PER_WALK = 4;
```

Per ciascuno dei seguenti sport, si indichino delle costanti Java che potrebbero essere utilizzate in un programma.

- Basket.
- Calcio.
- Tennis.
- Pallavolo.
- Bowling.

2.7 Progetti

1. Si scriva un programma che legge tre numeri interi e visualizza la media dei tre numeri.
2. Si scriva un programma che usi `Scanner` per leggere due stringhe dalla tastiera e che visualizzi ciascuna stringa su una riga distinta, indicandone la lunghezza. In seguito si crei una nuova stringa concatenando le due stringhe, ma separandole con uno spazio. Infine, visualizzare su una terza riga la terza stringa e la sua lunghezza.
3. Scrivere un programma che legga l'ammontare di un pagamento mensile per un'ipoteca e l'ammontare ancora dovuto (il debito rimanente). Il programma deve quindi visualizzare la parte di pagamento che serve per coprire gli interessi a debito e l'ammontare che serve per ridurre il debito. Si supponga che il tasso di interesse annuo sia del 7.49 %. Si utilizzi una costante per memorizzare il tasso di interesse. Si noti che i pagamenti vengono fatti mensilmente: l'interesse è quindi un dodicesimo del tasso di interesse annuale del 7.49 %.
4. Si scriva un programma che legge un intero di quattro cifre, per esempio 2010, e quindi lo visualizzi, una cifra per riga:

```
2
0
1
0
```

Il programma dovrebbe chiedere esplicitamente all'utente di inserire un numero di quattro cifre. Per la risoluzione del problema si può supporre che l'utente segua le indicazioni. *Suggerimento*: utilizzare gli operatori di divisione e resto.

5. Ripetete l'esercizio precedente, ma leggendo il numero di quattro cifre come una stringa. Risolvere il problema utilizzando i metodi della classe `String`.

6. Si scriva un programma che converta la temperatura da Fahrenheit a Celsius utilizzando la formula *gradiCelsius* = 5 * (*gradiFahreneit* – 32) / 9.

   Si chieda all'utente di digitare una temperatura in gradi Fahrenheit come un intero. La temperatura deve essere visualizzata in Celsius con un numero in virgola mobile con una precisione di un decimo di grado. Un possibile esempio di output potrebbe essere il seguente:

```
Scrivi la temperatura in gradi Fahrenheit:
72
72 gradi Fahreneit corrispondono a 22.2 gradi Celsius.
```

7. Si scriva un programma che legga una riga di testo e poi la visualizzi sostituendo la prima occorrenza della parola *odio* con *amore*. Un possibile esempio di output potrebbe essere:

```
Scrivi una riga di testo:
Io ti odio.
La riga è stata modificata in:
Io ti amo.
```

   Si può supporre che nell'input compaia la parola *odio*. Se la parola si presenta più volte, il programma deve sostituire solo la prima occorrenza.

8. Si scriva un programma che legga una riga di testo come input e che la visualizzi dopo aver spostato la prima parola alla fine della riga. Un possibile esempio del risultato da ottenere è il seguente:

```
Scrivi una riga di testo senza punteggiatura.
Java e' un linguaggio
La riga è stata modificata in:
e' un linguaggio Java
```

9. Si scriva un programma che chieda all'utente di scrivere il colore preferito, il piatto preferito, l'animale preferito, il nome di un amico o di un parente. Il programma dovrebbe quindi scrivere le due righe seguenti, dove l'input dell'utente sostituisce le lettere in corsivo:

```
Ho sognato che Nome aveva mangiato un Animale Colore
e aveva detto che sapeva di Piatto.
```

   Per esempio, se l'utente avesse inserito `blu` per il colore, `hamburger` per il piatto, cane per l'animale, e `Luca` per il nome della persona, il risultato dovrebbe essere:

```
Ho sognato che Luca aveva mangiato un cane blu
e aveva detto che sapeva di hamburger.
```

10. Si scriva un programma che determini il resto che deve essere restituito da un distributore automatico. Un prodotto del distributore automatico può costare da 25 centesimi a 1 Euro, con incrementi di 5 centesimi (25, 30, 35, …, 90, 95, 100) e il distributore accetta solo monete da 1 Euro.

Un possibile dialogo con l'utente potrebbe essere:

```
Scrivi il prezzo del prodotto
(da 25 centesimi a un Euro, con incrementi di 5 centesimi): 45
Hai comprato un prodotto da 45 centesimi inserendo un Euro,
il tuo resto è:
    1 monete da cinquanta centesimi,
    0 monete da venti centesimi,
    0 monete da dieci centesimi,
    1 monete da cinque centesimi.
```

11. Si scriva un programma che legga dalla tastiera un numero binario di quattro bit sotto forma di stringa e che poi lo converta in un numero decimale. Per esempio, se l'input è 1100, l'output deve essere 12. *Suggerimento*: si suddivida la stringa in sottostringhe e quindi si converta ciascuna sottostringa in un valore per ciascun bit. Se i bit sono b0, b1, b2 e b3 il numero decimale equivalente potrà esser calcolato con la formula 8 * b0 + 4 * b1 + 2 * b2 + b3.

12. Molti pozzi per l'acqua a uso privato producono solo tra 4 e 8 litri d'acqua al minuto. Un metodo per evitare di rimanere senz'acqua utilizzando uno di questi pozzi a produzione ridotta è quello di utilizzare un serbatoio. Una famiglia di quattro persone utilizza circa 1000 litri d'acqua al giorno. Tuttavia, il pozzo stesso costituisce un serbatoio "naturale". Più profondo è il pozzo, più acqua potrà essere immagazzinata per poi essere pompata fuori per l'uso domestico. Ma quanta acqua sarà disponibile?

    Scrivere un programma che consenta all'utente di inserire il raggio del tubo che costituisce il pozzo in centimetri (un pozzo tipico ha un raggio di 8 cm) e la profondità del pozzo in metri (si assuma per semplicità che l'acqua occupi l'intera profondità del pozzo, anche se in pratica ciò non sarà vero, dato che l'acqua arriverà solo a circa 15 metri dalla superficie). Il programma dovrà stampare il numero di litri d'acqua immagazzinati nel tubo. Si ricordi che:

    Il volume di un cilindro è $\pi r^2 h$, dove r è il raggio e h è l'altezza. 1 m^3 = 1000 litri.

    Per esempio, un pozzo pieno d'acqua, profondo 100 metri e con un raggio di 8 cm contiene circa 2000 litri d'acqua, più che sufficienti per una famiglia di quattro persone, senza la necessità di installare un serbatoio aggiuntivo.

13. L'equazione di Harris-Benedict stima il numero di calorie richieste dal corpo umano per mantenere costante il peso se non si svolge attività fisica. Tale valore è detto metabolismo basale o MB.

    Le calorie necessarie a una donna per mantenere il suo peso sono date da:

    MB = 655 + (9.6 * peso in kg) + (1.8 * altezza in cm) – (4.7 * età in anni)

    Le calorie necessarie a un uomo sono date invece da:

    MB = 66 + (13.8 * peso in kg) + (5.0 * altezza in cm) – (6.8 * età in anni)

    Una barretta di cioccolato contiene circa 230 calorie. Scrivere un programma che consenta all'utente di inserire il proprio peso in kg, la propria altezza in cm e la propria età. Il programma dovrà stampare il numero di barrette di cioccolato che si dovrebbero consumare per mantenere il proprio peso sia nel caso di un uomo che di una donna con i valori specificati di peso, altezza ed età.

Capitolo 3

Flusso di controllo: la selezione

OBIETTIVI

- Utilizzare in un programma le strutture decisionali di Java `if-else` e `switch`.
- Confrontare valori di tipo primitivo.
- Confrontare oggetti di tipo stringa.
- Usare il tipo primitivo `boolean`.
- Usare semplici enumerazioni all'interno di un programma.

Con il termine flusso di controllo (*flow of control*) si intende l'ordine con cui vengono eseguite o valutate le diverse azioni di un programma. L'ordine delle azioni svolte nei programmi presentati finora è sempre stato semplice: le azioni venivano eseguite nell'ordine in cui erano state scritte (esecuzione *sequenziale*). Questo capitolo spiega come scrivere programmi con un flusso di controllo più articolato.

Java, come la maggior parte dei linguaggi di programmazione, usa due tipi di istruzioni (o costrutti sintattici) per regolare il flusso di controllo: la selezione e il ciclo. La selezione (*branching*) sceglie un'azione da un elenco di una o più possibili azioni, il ciclo (*loop*), invece, ripete un'azione più volte, fino a quando non viene incontrata una qualche condizione di terminazione. Questi due tipi di istruzioni costituiscono le **strutture di controllo** in un programma. Dal momento che le istruzioni di selezione scelgono tra più azioni, sono anche chiamate *strutture decisionali*. Questo capitolo descrive proprio le strutture decisionali, mentre il prossimo presenta i cicli.

Prerequisiti

Per leggere questo capitolo bisognerebbe avere familiarità con gli argomenti trattati nel Capitolo 2.

3.1 Istruzione `if-else`

3.1.1 Istruzione `if-else` semplice

Nei programmi, così come nella vita di tutti i giorni, le cose possono andare in modi diversi. Per esempio, se una persona possedesse un conto corrente in attivo, la banca verserebbe un interesse. Dall'altro lato, se una persona possedesse un conto corrente, ma prelevasse più della reale disponibilità, dovrebbe pagare una penalità che renderebbe il saldo del conto ancora più negativo. Se si inserisse questa regola all'interno di un programma che gestisce i conti correnti di una banca, si utilizzerebbe la seguente istruzione `if-else`:

```
if (saldo >= 0)
    saldo = saldo + (TASSO_INTERESSE * saldo) / 12;
else
    saldo = saldo - PENALITA;
```

La coppia di simboli >= in Java ha il significato di *maggiore o uguale a*. Viene utilizzata questa notazione in quanto il simbolo ≥ non è presente sulla tastiera.

Il significato di un'istruzione `if-else` è analogo a quello di un "*se... allora... altrimenti...*" in una frase in italiano. Quando il programma esegue un'istruzione `if-else`, in primo luogo controlla il risultato dell'espressione posta tra parentesi dopo la parola chiave `if`. Questa espressione deve avere un risultato che può essere o *true* (vero) o *false* (falso). Se il risultato è *true*, viene eseguita l'istruzione successiva (prima della parola chiave `else`). Se il risultato è *false*, viene eseguita l'istruzione che segue l'istruzione `else`. In altri termini, l'istruzione `if-else` permette di scegliere tra due **rami** (*branch*): il ramo `if` e il ramo `else`.

Nell'esempio precedente, se la variabile `saldo` è positiva o uguale a 0, viene intrapresa la seguente azione:

```
saldo = saldo + (TASSO_INTERESSE * saldo) / 12;
```

(dal momento che si aggiungono al saldo soltanto gli interessi corrispondenti a un mese, l'interesse annuale viene diviso per 12). Dall'altro lato, se il valore di `saldo` è negativo, viene eseguita la seguente azione:

```
saldo = saldo - PENALITA;
```

La Figura 3.1 mostra l'azione eseguita da questo blocco `if-else`, mentre il Listato 3.1 mostra questa azione inserita in un programma completo.

LISTATO 3.1 Un programma che utilizza l'istruzione `if-else`.

```
import java.util.Scanner;

public class SaldoBanca {

   public static final double PENALITA = 6.00;
   public static final double TASSO_INTERESSE = 0.02; //2% annuo

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

```
        System.out.print("Inserisci il saldo del tuo conto: ");
        Scanner tastiera = new Scanner(System.in);
        saldo = tastiera.nextDouble();
        System.out.println("Saldo originale: " + saldo);

        if (saldo >= 0)
            saldo = saldo + (TASSO_INTERESSE * saldo) / 12;
        else
            saldo = saldo - PENALITA;

        System.out.print("Dopo gli adeguamenti del mese corrente, ");
        System.out.println("legati a interessi e penalita'");
        System.out.print("il saldo corrente e': " + saldo);
    }
}
```

Esempio di output 1

```
Inserisci il saldo del tuo conto: 506.79
Saldo originale: 506.79
Dopo gli adeguamenti del mese corrente, legati a interessi e penalita'
il saldo corrente e': 507.63465
```

Esempio di output 2

```
Inserisci il saldo del tuo conto corrente: -23
Saldo originale: -23.0
Dopo gli adeguamenti del mese corrente, legati a interessi e penalita'
il saldo corrente e': -31.0
```

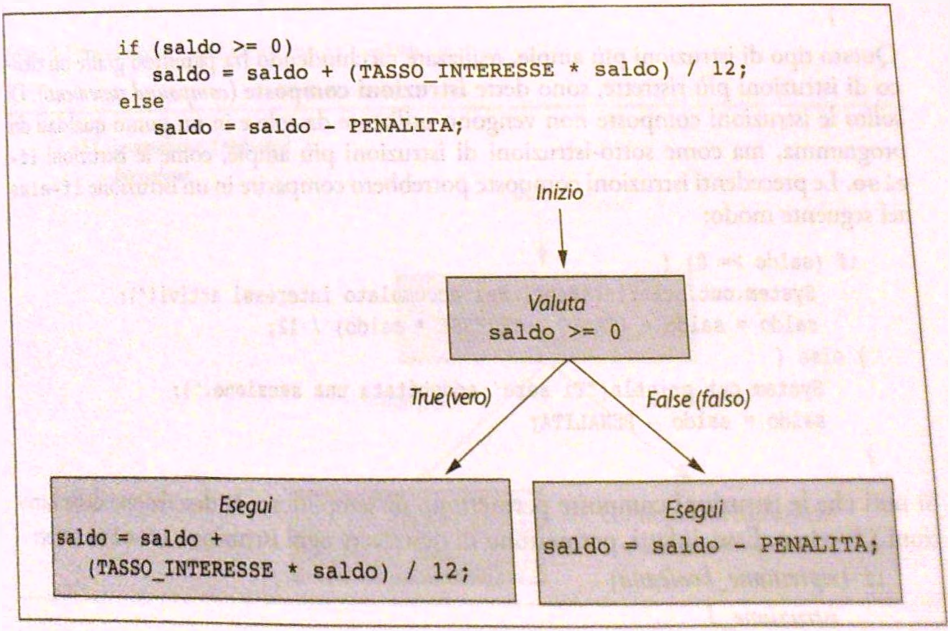


Figura 3.1 Esecuzione dell'istruzione `if-else` del Listato 3.1.

L'espressione `saldo >= 0` è un esempio di **espressione booleana**. Semplicemente si tratta di un'espressione che può avere valore *true* (vero) o *false* (falso).

L'aggettivo "booleano" (*boolean*) deriva dal nome di George Boole, un logico e matematico inglese del 19° secolo, il cui lavoro è il fondamento matematico di questo tipo di espressioni. Il prossimo paragrafo descrive più nel dettaglio le espressioni booleane.

Si noti che un'istruzione `if-else` contiene altre due istruzioni. Per esempio l'istruzione `if-else` del Listato 3.1 contiene le due istruzioni seguenti:

```
saldo = saldo + (TASSO_INTERESSE * saldo) / 12;
saldo = saldo - PENALITA;
```

Si noti che queste due istruzioni sono state fatte rientrare di un livello in più rispetto alle istruzioni `if` ed `else`.

Indentazione

Il capitolo precedente aveva già trattato le indentazioni. Si tratta di un'ottima abitudine. Infatti, sebbene il compilatore ignori le indentazioni, un'indentazione non coerente può confondere chi legge il programma e lo stesso sviluppatore.

Se occorre includere più di un'istruzione in ciascuno dei due rami definiti dall'istruzione `if-else`, è sufficiente racchiudere le diverse istruzioni tra parentesi graffe { }. Un insieme di istruzioni racchiuse tra parentesi graffe è considerato come un'unica istruzione "più ampia". Il codice seguente presenta un'istruzione che include due istruzioni:

```
{
    System.out.println("Bene! Hai accumulato degli interessi attivi!");
    saldo = saldo + (TASSO_INTERESSE * saldo) / 12;
}
```

Questo tipo di istruzioni più ampie, realizzate racchiudendo fra parentesi graffe un elenco di istruzioni più ristrette, sono dette **istruzioni composte** (*compound statements*). Di solito le istruzioni composte non vengono utilizzate da sole e in un punto qualsiasi del programma, ma come sotto-istruzioni di istruzioni più ampie, come le istruzioni `if-else`. Le precedenti istruzioni composte potrebbero comparire in un'istruzione `if-else` nel seguente modo:

```
if (saldo >= 0) {
    System.out.println("Bene! Hai accumulato interessi attivi!");
    saldo = saldo + (TASSO_INTERESSE * saldo) / 12;
} else {
    System.out.println("Ti sara' addebitata una sanzione.");
    saldo = saldo - PENALITA;
}
```

Si noti che le istruzioni composte permettono di semplificare la descrizione delle istruzioni `if-else`. Esse, infatti, permettono di descrivere ogni istruzione `if-else` come:

```
if (espressione_booleana)
    istruzione_1
else
    istruzione_2
```

Se uno o entrambi i rami dell'istruzione `if-else` devono contenere più istruzioni (invece di una sola), occorre utilizzare un'istruzione composta al posto di *istruzione_1* e/o *istruzione_2*. La Figura 3.2 riassume la **semantica** (il significato) di un'istruzione `if-else`.

Se si omette l'istruzione `else` e ciò che la segue, il programma semplicemente non esegue *istruzione_1* nel caso in cui l'espressione booleana dell'istruzione `if` sia *false*, così come viene mostrato dalla Figura 3.3. Per esempio, se la banca non addebitasse una penalità per i conti scoperti, l'istruzione mostrata in precedenza si accorcerebbe come segue:

```
if (saldo >= 0) {
   System.out.println("Bene! Hai accumulato interessi attivi!");
   saldo = saldo + (TASSO_INTERESSE * saldo) / 12;
}
```

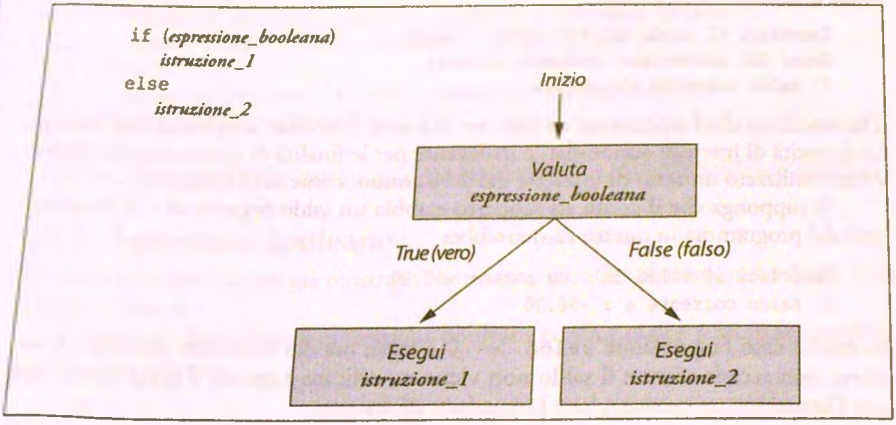


Figura 3.2 La semantica dell'istruzione `if - else`.

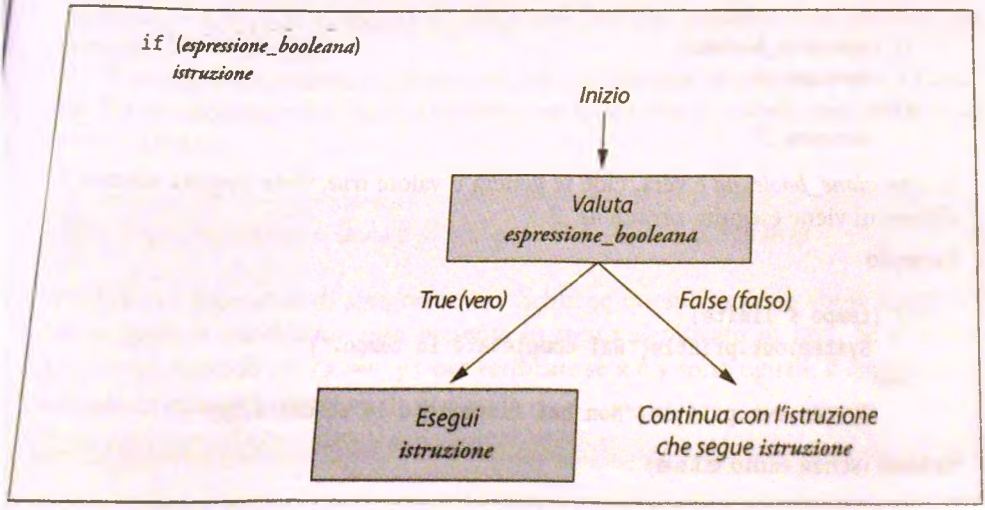


Figura 3.3 La semantica dell'istruzione `if` senza `else`.

Per vedere come funziona questa istruzione, si consideri l'esempio seguente in cui sono state aggiunte alcune istruzioni che permettono di contestualizzare meglio il tutto (si supponga che la variabile `tastiera` sia un oggetto di tipo `Scanner`):

```
System.out.print("Inserisci il saldo del tuo conto: ");
saldo = tastiera.nextDouble();
if (saldo >= 0) {
    System.out.println("Bene! Hai accumulato interessi attivi!");
    saldo = saldo + (TASSO_INTERESSE * saldo) / 12;
}
System.out.println("Il saldo corrente e': " + saldo);
```

L'esecuzione di queste istruzioni nel caso in cui il saldo del conto sia di 100 Euro produrrebbe la seguente interazione:

```
Inserisci il saldo del tuo conto: 100.00
Bene! Hai accumulato interessi attivi!
Il saldo corrente e': 100.16
```

Dal momento che l'espressione `saldo >= 0` è vera, sono stati accumulati degli interessi. La quantità di interessi accumulati è irrilevante per le finalità di questo esempio (tuttavia è stato utilizzato un tasso di interesse del 2 % annuo, come nel Listato 3.1).

Si supponga che il conto sia scoperto e abbia un saldo negativo di - 50 Euro. L'output del programma in questo caso sarebbe:

```
Inserisci il saldo del tuo conto: -50.00
Il saldo corrente e': -50.00
```

In questo caso l'espressione `saldo >= 0` è falsa; ma dal momento che manca il ramo `else`, non accade niente: il saldo non viene modificato e quindi il programma procede con l'istruzione successiva, che è l'istruzione di output.

La sintassi dell'istruzione `if-else` (forma semplice)

```
if (espressione_booleana)
    istruzione_1
else
    istruzione_2
```

Se *espressione_booleana* è vera, cioè se genera il valore *true*, viene eseguita *istruzione_1*, altrimenti viene eseguita *istruzione_2*.

Esempio

```
if (tempo < limite)
    System.out.println("Hai completato in tempo.");
else
    System.out.println("Non hai rispettato la scadenza.");
```

Sintassi (senza ramo `else`)

```
if (espressione_booleana)
    istruzione
```

Se *espressione_booleana* è vera, viene eseguita *istruzione*; altrimenti *istruzione* viene ignorata e il programma prosegue con l'istruzione successiva.

Esempio

```
if (peso > ideale)
   calorieGiornaliere = calorieGiornaliere - 500;
```

Istruzioni composte

Ogni istruzione, sia *istruzione_1*, sia *istruzione_2*, può essere un'istruzione composta. Per eseguire più istruzioni in un ramo del costrutto `if-else`, occorre raggruppare le istruzioni tra parentesi graffe, come nell'esempio che segue:

```
if (saldo >= 0) {
   System.out.println("Bene! Hai accumulato interessi attivi!");
   saldo = saldo + (TASSO_INTERESSE * saldo) / 12;
} else {
   System.out.println("Ti sara' addebitata una sanzione.");
   saldo = saldo - PENALITA;
}
```

3.1.2 Espressioni booleane

Gli esempi precedenti hanno già mostrato come usare semplici espressioni booleane nelle istruzioni `if-else`.

La forma più elementare di espressione booleana confronta due espressioni semplici, come in questi esempi:

```
saldo >= 0
```

e

```
tempo < limite
```

La Figura 3.4 mostra gli **operatori di confronto** Java che possono essere utilizzati per confrontare due espressioni.

Si noti che un'espressione booleana non deve cominciare né terminare con le parentesi. Tuttavia è necessario racchiudere l'espressione tra parentesi quando viene usata in un costrutto `if-else`.

Usare l'operatore = invece di = = per verificare l'uguaglianza

Il simbolo = è l'operatore di assegnamento. Sebbene questo simbolo abbia il significato di uguale in matematica, non presenta lo stesso significato in Java. Se si scrive `if (x = y)` invece di `if (x == y)` per verificare se x e y sono uguali, il compilatore restituisce un messaggio di errore di sintassi.

Notazione matematica	Nome	Notazione Java	Esempio Java
=	Uguale a	==	saldo == 0 risposta == 'y'
≠	Diverso da	!=	entrata != tassa risposta != 'y'
>	Maggiore di	>	spese > entrate
≥	Maggiore o uguale a	>=	punti >= 60
<	Minore di	<	pressione < max
≤	Minore o uguale a	<=	spese <= entrate

Figura 3.4 Operatori di confonto Java.

Usare = = oppure ! = per confrontare i numeri in virgola mobile

Il capitolo precedente ha sottolineato il fatto che bisogna sempre considerare i numeri in virgola mobile come approssimazioni. I numeri che possiedono una parte decimale, infatti, hanno un numero infinito di cifre. Dal momento che il computer può memorizzare in un'area di memoria solo un numero limitato di cifre, i numeri in virgola mobile non sono esatti. Queste approssimazioni possono addirittura diventare sempre meno accurate a ogni calcolo eseguito.

Per questo motivo, se si computa il valore di due numeri in virgola mobile, sarà molto difficile che questi risultino uguali. Quello che di solito succede è che due valori siano molto simili. Per questo motivo è bene non usare l'operatore == per confrontare i numeri in virgola mobile. Usare l'operatore != può causare lo stesso problema.

Per verificare l'uguaglianza di due numeri in virgola mobile è bene verificare se differiscono di così poco che tale differenza sia dovuta alla loro natura approssimata. Nel momento in cui ci si accorge di questo, i due numeri possono essere considerati uguali.

Partendo da espressioni semplici si possono costruire espressioni booleane più complesse unendo le espressioni semplici con l'**operatore logico** `&&`, chiamato operatore *and*, che è la versione Java della congiunzione, comunemente indicata con la parola "e". Si considerino le seguenti istruzioni:

```
if ((pressione > min) && (pressione < max))
   System.out.println("Pressione OK");
else
   System.out.println("Attenzione: pressione fuori limite.");
```

Se il valore della variabile `pressione` è maggiore di `min` *e* anche minore di `max`, l'output sarà:

```
Pressione OK
```

Altrimenti l'output sarà:

```
Attenzione: pressione fuori limite.
```

Si noti che non si può scrive un'espressione come

```
min < pressione < max    <-- Non corretto!
```

È invece necessario scrivere le disuguaglianze separatamente e connetterle mediante `&&` come segue:

```
(pressione > min) && (pressione < max)
```

Quando si scrive un'espressione booleana complessa connettendo due espressioni semplici con l'operatore `&&`, l'espressione complessa risulta vera solo se entrambe le espressioni semplici sono vere. Se anche una sola delle espressioni semplici è falsa, l'intera espressione risulta falsa. Per esempio, l'espressione composta:

```
(pressione > min) && (pressione < max)
```

è vera solo se `(pressione > min)` e `(pressione < max)` sono entrambe vere, altrimenti risulta falsa.

Usare && come congiunzione (e)

Il simbolo `&&` vuol dire *e* in Java. Si può utilizzare `&&` per costruire un'espressione booleana più complessa a partire da due espressioni semplici.

Sintassi

(sotto_espressione_1) `&&` *(sotto_espressione_2)*

Questa espressione è vera se e solo se entrambe le espressioni *sotto_espressione_1* e *sotto_espressione_2* sono vere.

Esempio

```
if ((pressione > min) && (pressione < max))
   System.out.println("Pressione OK");
else
   System.out.println("Attenzione: pressione fuori limite.");
```

Le espressioni booleane possono essere unite anche con una disgiunzione, cioè con l'operatore *or*. Nella lingua italiana la disgiunzione viene comunemente indicata con la parola "*o*" (*or* in inglese). In Java per esprimere la disgiunzione si usa il simbolo `||`, che viene creato digitando due barre verticali (in alcuni sistemi il simbolo | compare con uno spazio nel mezzo). Il significato dell'operatore `||` è lo stesso del termine italiano *o*. Si considerino, per esempio, le istruzioni seguenti:

```
if ((salario > spese) || (risparmi > spese))
   System.out.println("Solvente.");
else
   System.out.println("Bancarotta.");
```

Se il valore della variabile `salario` è maggiore del valore della variabile `spese` *o* se il valore della variabile `risparmi` è maggiore del valore della variabile `spese`, *oppure se entrambi sono vere*, l'output del programma sarà:

```
Solvente.
```

Altrimenti l'output sarà:

```
Bancarotta.
```

Usare || come disgiunzione (o)

La coppia di simboli || vuol dire *o* (*oppure*) in Java. Si può usare il simbolo || per formare un'espressione booleana più complessa a partire da due espressioni semplici.

Sintassi

(sotto_espressione_1) || *(sotto_espressione_2)*

Questa espressione è vera se *sotto_espressione_1* è vera, oppure se *sotto_espressione_2* è vera, oppure se entrambe sono vere.

Esempio

```
if ((salario > spese) || (risparmi > spese))
    System.out.println("Solvente.");
else
    System.out.println("Bancarotta.");
```

L'espressione booleana in un'istruzione `if-else` deve essere racchiusa tra parentesi. Un'istruzione `if-else` che usa l'operatore `&&` viene normalmente scritta utilizzando le parentesi come segue:

```
if ((pressione > min) && (pressione < max))
```

Le parentesi nell'espressione `(pressione > min)` e in `(pressione < max)` non sono necessarie, tuttavia è bene includerle per rendere l'espressione più leggibile. Le parentesi sono utilizzate allo stesso modo anche nelle espressioni che contengono l'operatore || al posto dell'operatore `&&`.

In Java si può negare un'espressione booleana facendola precedere dal simbolo `!`. Per esempio,

```
if (!(numero >= min))
    System.out.println("Troppo piccolo");
else
    System.out.println("OK");
```

Se la variabile `numero` *non* è maggiore o uguale a `min`, l'output sarà

```
Troppo piccolo
```

Altrimenti l'output sarà

```
OK
```

È bene evitare di usare l'operatore ! quando possibile. Per esempio, l'istruzione `if-else` dell'esempio precedente è equivalente a:

```
if (numero < min)
    System.out.println("Troppo piccolo");
else
    System.out.println("OK");
```

La Figura 3.5 mostra come trasformare un'espressione nella forma

`!(A` *operatore_di_confronto* `B)`

in un'espressione senza l'operatore di negazione. Se possibile, è bene evitare di usare l'operatore ! nei programmi al fine di renderli più comprensibili. Tuttavia l'operatore ! sarà adottato più volte nei prossimi paragrafi per chiarire il significato dell'espressione. In particolare l'operatore ! sarà utilizzato nel contesto delle iterazioni, che saranno introdotte nel prossimo capitolo, oppure per negare il valore di una variabile booleana come mostrato alla fine di questo capitolo.

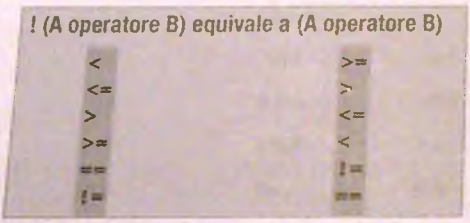


Figura 3.5 Evitare l'operatore di negazione.

Usare ! per la negazione

L'operatore ! rappresenta la negazione in Java. Si può usare ! per negare il valore di un'espressione booleana. Questa operazione è molto comune quando, per esempio, l'espressione è una variabile booleana. Molto spesso un'espressione booleana può essere riscritta in modo da evitare l'uso della negazione.

Sintassi

!espressione_booleana

Il valore di questa espressione è l'opposto del valore dell'espressione booleana *espressione_booleana*: vera se *espressione_booleana* è falsa; falsa se *espressione_booleana* è vera.

Esempio

```
if (!(numero < 0))
    System.out.println("OK");
else
    System.out.println("Negativo!");
```

MyLab

leo 3.1
ivere
ressioni
leane

La Figura 3.6 mostra i tre diversi operatori logici di Java. Java combina i valori *true* (vero) e *false* (falso) secondo le regole presentate nella tabella rappresentata nella Figura 3.7. Per esempio, se l'espressione booleana `A` ha valore `true` e l'espressione booleana `B` ha valore `false`, l'espressione `A && B` avrà valore `false`.

Nome	NotazioneJava	Esempio Java
And logico (congiunzione - *e*)	`&&`	`(somma > min) && (somma < max)`
Or logico (disgiunzione - *o*)	`||`	`(risposta == 's') || (risposta == 'S')`
Not logico (negazione - *non*)	`!`	`!(numero < 0)`

Figura 3.6 *Operatori logici in Java.*

Valore di *A*	Valore di *B*	Valore di *A && B*	Valore di *A || B*	Valore di *!(A)*
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Figura 3.7 *Effetto degli operatori booleani* `&&` (and), `||` (or) e `!` (not).

3.1.3 Istruzioni `if-else` annidate

Un'istruzione di controllo `if-else` può contenere qualsiasi sorta di istruzione. In pa ticolare, si può **annidare** (*nest*) un'istruzione `if-else` all'interno di un'altra istruzio `if-else`, come illustrato nelle seguenti righe:

```
if (saldo >= 0)
    if (TASSO_INTERESSE >= 0)
        saldo = saldo + (TASSO_INTERESSE * saldo) / 12;
    else
        System.out.println("Non si puo' avere un interesse" +
                           " negativo.");
else
    saldo = saldo - PENALITA;
```

Se il valore della variabile `saldo` è maggiore o uguale a 0, viene eseguito tutto il segue blocco `if-else`:

```
if (TASSO_INTERESSE >= 0)
    saldo = saldo + (TASSO_INTERESSE * saldo) / 12;
else
    System.out.println("Non si puo' avere un interesse" +
                       " negativo.");
```

Le istruzioni annidate possono essere rese più chiare aggiungendo delle parentesi, come nel seguente esempio:

```
if (saldo >= 0) {
    if (TASSO_INTERESSE >= 0)
        saldo = saldo + (TASSO_INTERESSE * saldo) / 12;
    else
        System.out.println("Non si puo' avere un interesse" +
                           " negativo.");
} else
    saldo = saldo - PENALITA;
```

In questo caso, le parentesi aiutano a rendere più comprensibile il codice, ma non sono strettamente necessarie. In altri casi, invece, le parentesi sono necessarie. Se omettiamo un'istruzione `else`, per esempio, il codice diventa più complesso da comprendere. Le due istruzioni che seguono differiscono apparentemente solo perché la prima presenta una coppia di parentesi in più, tuttavia esse *non hanno* lo stesso comportamento:

```
//Prima Versione – Con parentesi graffe
if (saldo >= 0) {
    if (TASSO_INTERESSE >= 0)
        saldo = saldo + (TASSO_INTERESSE * saldo) / 12;
} else
    saldo = saldo - PENALITA;

//Seconda Versione – Senza parentesi graffe
if (saldo >= 0)
    if (TASSO_INTERESSE >= 0)
        saldo = saldo + (TASSO_INTERESSE * saldo) / 12;
else
    saldo = saldo - PENALITA;
```

In un blocco `if-else`, ciascun `else` è associato al più vicino `if`. La seconda versione, quella senza parentesi graffe, associa l'`else` al secondo `if`, sebbene l'indentazione del codice sembri dire un'altra cosa. Il significato della seconda versione è equivalente quindi al seguente blocco di istruzioni:

```
//Equivalente alla seconda Versione
if (saldo >= 0) {
    if (TASSO_INTERESSE >= 0)
        saldo = saldo + (TASSO_INTERESSE * saldo) / 12;
    else
        saldo = saldo - PENALITA;
}
```

Per rendere ancora più chiara questa differenza, si consideri quello che accade quando il valore della variabile `saldo` è maggiore o uguale a 0. Nella prima versione, vengono eseguite le seguenti istruzioni:

```
if (TASSO_INTERESSE >= 0)
    saldo = saldo + (TASSO_INTERESSE * saldo) / 12;
```

Sempre nella prima versione, se il valore della variabile `saldo` *non è* maggiore o uguale a 0, viene eseguita la seguente istruzione:

Nella seconda versione invece, se il valore della variabile `saldo` è maggiore o uguale a 0, viene eseguito interamente il seguente blocco `if-else`:

```
if (TASSO_INTERESSE >= 0)
    saldo = saldo + (TASSO_INTERESSE * saldo) / 12;
else
    saldo = saldo - PENALITA;
```

Sempre nella seconda versione, se il valore della variabile `saldo` *non è* maggiore o uguale a 0, non viene eseguita alcuna istruzione.

Corrispondenze tra `else` e `if`

In un blocco di istruzioni `if-else`, ciascuna istruzione `else` viene associata all'istruzione `if` più vicina, sempre che questa non sia associata a un'altra istruzione `else`. Per rendere più chiaro il codice, è bene usare un'indentazione coerente con il significato dell'istruzione. Tuttavia, occorre sempre ricordarsi che il compilatore non tiene conto delle indentazioni. Utilizzare le parentesi graffe risulta quindi molto utile per rendere esplicito il significato di un blocco `if-else`.

3.1.4 Istruzioni `if-else` multi-ramo

Come è possibile creare costrutti di selezione a due rami, è anche possibile crearne con quattro rami. Per realizzarli, basta creare un costrutto `if-else` a due rami (cioè in cui siano presenti sia l'istruzione `if`, sia l'istruzione `else`) e far sì che ciascun ramo includa a sua volta un costrutto `if-else`. Con questa tecnica, è possibile annidare istruzioni `if-else` per realizzare un numero qualsiasi di ramificazioni. I programmatori adottano un approccio standard per realizzare questo tipo di selezioni multi-ramo. Tale approccio si è così consolidato tanto da essere trattato come un nuovo costrutto di selezione, anche se si tratta semplicemente di un insieme di istruzioni `if-else` annidate. Un esempio permetterà di illustrare questi concetti.

Si supponga che la variabile `saldo` contenga il saldo di un conto in banca e che si voglia conoscere se il saldo è positivo, negativo (cioè se il conto sia in rosso) o a 0. Per evitare qualsiasi questione sull'accuratezza dei calcoli, si supponga che la variabile `saldo` contenga il numero di Euro interi presenti nel conto, ignorando i centesimi. Si supponga, cioè, che `saldo` sia di tipo `int`. Per capire se il saldo è positivo, negativo o 0 si potrebbe utilizzare il seguente costrutto `if-else` annidato:

```
if (saldo > 0)
    System.out.println("Saldo positivo");
else
    if (saldo < 0)
        System.out.println("Saldo negativo");
    else
        if (saldo == 0)
            System.out.println("Saldo 0");
```

Un modo più chiaro per scrivere queste stesse istruzioni è il seguente:

```
if (saldo > 0)
    System.out.println("Saldo positivo");
```

```
else if (saldo < 0)
    System.out.println("Saldo negativo");
else if (saldo == 0)
    System.out.println("Saldo 0");
```

Questa forma prende il nome di costrutto `if-else` **multi-ramo** (*multibranch*). Questo costrutto corrisponde a tutti gli effetti a un costrutto ordinario di tipo `if-else` annidato.

Quando viene eseguito un costrutto `if-else` multi-ramo, il computer verifica le espressioni booleane una dopo l'altra, partendo dall'alto. Quando rileva la prima espressione booleana vera, esegue l'istruzione che la segue. Se, per esempio, il valore della variabile `saldo` è maggiore di 0, il codice precedente mostrerà la scritta `Saldo positivo`. Se il valore della variabile `saldo` è minore di 0, verrà visualizzata la scritta `Saldo negativo`. Infine, se il valore della variabile `saldo` è pari a 0 verrà visualizzata la scritta `Saldo 0`. Verrà prodotto soltanto uno dei tre possibili output, a seconda del valore della variabile `saldo`.

Questo primo esempio presentava solo tre possibili casistiche, tuttavia se ne può definire un numero qualsiasi: basta aggiungere nuovi blocchi `else-if`. Inoltre, le casistiche di questo primo esempio erano tra loro **mutuamente esclusive** (ovvero una sola risulta vera in un certo istante). Tuttavia, si possono utilizzare espressioni booleane di qualsiasi natura, anche se non sono mutuamente esclusive. Se è vera più di un'espressione booleana, viene eseguita solo l'azione associata alla prima espressione booleana vera. Un costrutto `if-else` multi-ramo non esegue mai più di un'azione.

Se nessuna delle espressioni booleane risulta vera, non accade nulla. Tuttavia, è buona pratica aggiungere una clausola `else`, priva di qualsiasi `if`, alla fine del blocco multi-ramo. Questa istruzione verrà eseguita qualora nessuna delle espressioni booleane risultasse vera. L'esempio del saldo presentato in precedenza può essere, infatti, riscritto in questo modo. Se `saldo` non è né positivo, né negativo, deve per forza essere uguale a 0. Aggiungere l'ultimo controllo, `if (saldo == 0)`, è quindi inutile. Per questo motivo, il costrutto `if-else` multi-ramo presentato in precedenza risulta essere equivalente al seguente:

```
if (saldo > 0)
    System.out.println("Saldo positivo");
else if (saldo < 0)
    System.out.println("Saldo negativo");
else
    System.out.println("Saldo 0");
```

Istruzioni `if-else` multi-ramo

Sintassi

```
if (espressione_booleana_1)
    azione_1
else if (espressione_booleana_2)
    azione_2
...
else if (espressione_booleana_n)
    azione_n
else
```

Le azioni sono istruzioni Java. Le espressioni booleane vengono verificate una dopo l'altra, iniziando dalla prima. Quando viene individuata un'espressione booleana vera, viene eseguita l'azione che la segue, mentre le istruzioni di controllo seguenti vengono ignorate. Pertanto, l'istruzione *azione_di_default* viene eseguita solo se nessuna di queste espressioni booleane risulta vera. La Figura 3.8 mostra la semantica di questa istruzione `if-else`.

Esempio

```
if (numero < 10)
    System.out.println("numero < 10");
else if (numero < 50)
    System.out.println("numero >= 10 e numero <= 50");
else if (numero < 100)
    System.out.println("numero >= 50 e numero < 100");
else
    System.out.println("numero  > 100");
```

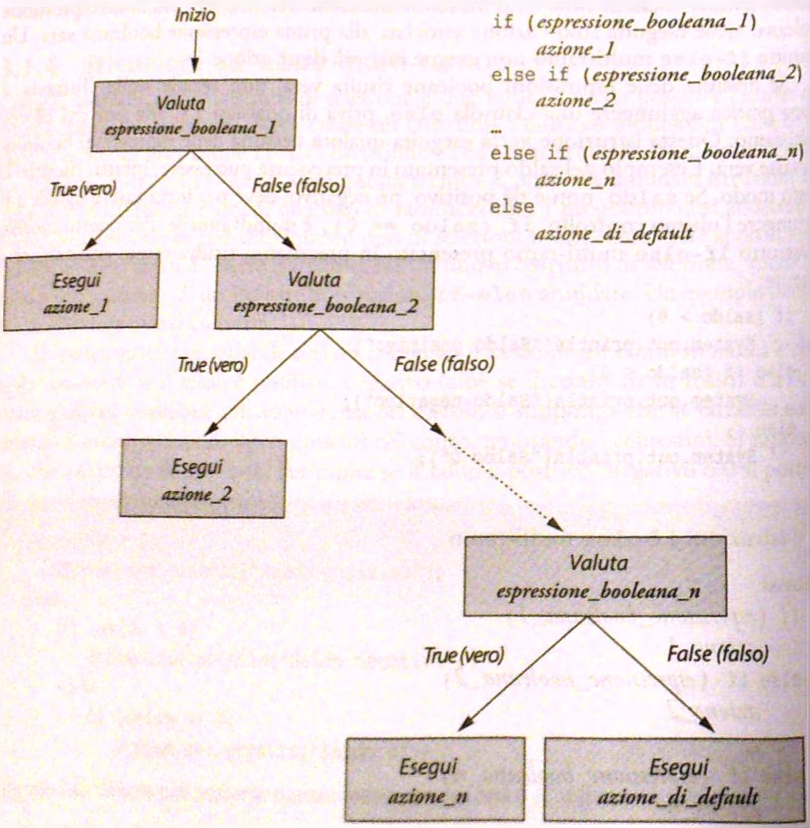


Figura 3.8 La semantica di un'istruzione `if-else` multi-ramo.

LISTATO 3.2 Assegnazione dei voti utilizzando un'istruzione `if-else` multi-ramo.

MyLab

```
import java.util.Scanner;

public class AssegnazioneVoti {

   public static void main(String[] args) {
      int punteggio;
      char voto;

      System.out.println("Inserisci il tuo punteggio:");
      Scanner tastiera = new Scanner(System.in);
      punteggio = tastiera.nextInt();

      if (punteggio >= 90)
         voto = 'A';
      else if (punteggio >= 80)
         voto = 'B';
      else if (punteggio >= 70)
         voto = 'C';
      else if (punteggio >= 60)
         voto = 'D';
      else
         voto = 'F';

      System.out.println("Punteggio = " + punteggio);
      System.out.println("Voto = " + voto);
   }
}
```

Esempio di output

```
Inserisci il tuo punteggio:
85
Punteggio = 85
Voto = B
```

Il Listato 3.2 contiene un programma che assegna un voto in lettere: un punteggio pari o superiore a 90 corrisponde a una A, un punteggio tra 80 e 90 a una B e così via.

Si noti che in un blocco `if-else` multi-ramo, le espressioni booleane vengono valutate in ordine, perciò la seconda espressione non viene mai verificata a meno che la prima non risulti falsa. Per questo motivo, se la seconda condizione viene valutata, allora la prima espressione è falsa e perciò il punteggio è inferiore a 90. Quindi, il blocco `if-else` multi-ramo avrebbe lo stesso significato se sostituissimo la condizione:

```
(punteggio >= 80)
```

con la condizione

```
((punteggio >= 80) && (punteggio < 90))
```

Applicando lo stesso ragionamento a ciascuna espressione booleana, si nota che il blocco multi-ramo del Listato 3.3 è equivalente al seguente:

```
if (punteggio >= 90)
    voto = 'A';
else if ((punteggio >= 80) && (punteggio < 90))
    voto = 'B';
else if ((punteggio >= 70) && (punteggio < 80))
    voto = 'C';
else if ((punteggio >= 60) && (punteggio < 70))
    voto = 'D';
else
    voto = 'F';
```

La maggior parte dei programmatori userebbe la versione nel Listato 3.2, in quanto più efficiente ed elegante. Tuttavia entrambe le versioni sono accettabili.

CASO DI STUDIO
INDICE DI MASSA CORPOREA

L'indice di massa corporea (IMC) è utilizzato per stimare il rischio dell'insorgenza di problemi legati al peso considerando l'altezza e la massa di un soggetto. Fu inventato dal matematico Adolphe Quetelet nel diciannovesimo secolo ed è talvolta chiamato anche indice di Quetelet. L'IMC si calcola secondo la formula

$$IMC = \frac{massa}{altezza^2}$$

In questa formula, la massa deve essere espressa in kilogrammi e l'altezza in metri. La classificazione del rischio per la salute associato a un dato valore di IMC è la seguente:

- sottopeso se IMC < 18.5
- peso normale se IMC ≥ 18.5 e IMC < 25
- sovrappeso su IMC ≥ 25 e IMC < 30
- obeso se IMC ≥ 30

In questo caso di studio, l'utente inserisce il proprio peso in kilogrammi e la propria altezza in metri e centimetri. Il programma, quindi, calcola e stampa l'IMC e la classe di rischio associata. Per fare questo, è necessario convertire un'altezza espressa in metri e centimetri in una espressa in metri.

Algoritmo per il calcolo dell'IMC

1. Leggere il peso in kilogrammi salvandolo nella variabile `kilogrammi`
2. Leggere i metri dell'altezza salvando il valore nella variabile `metri`

3. Leggere i centimetri addizionali dell'altezza salvando il valore nella variabile `centimetri`
4. Impostare la variabile `altezza` al valore `(metri + centimetri * 0.01)`
5. Impostare la variabile `massa` al valore uguale a `kilogrammi`
6. Impostare la variabile `IMC` al valore `massa / (altezza * altezza)`
7. Stampare `IMC`
8. Se `IMC < 18.5`, stampare "Sottopeso."
9. Altrimenti se `IMC ≥ 18.5` e `IMC < 25` stampare "Peso normale."
10. Altrimenti se `IMC ≥ 25` e `IMC < 30` stampare "Sovrappeso."
11. Altrimenti stampare "Obeso."

Questo algoritmo funziona, ma può essere semplificato ulteriormente. Se l'IMC è minore di 18.5, la condizione del punto 8 risulterà soddisfatta e il programma stamperà "Sottopeso.". Il punto 9 sarà eseguito solo se la condizione del punto 8 risulta falsa, il che significa che se il programma raggiunge il punto 9, si sa automaticamente che l'IMC non è minore di 18.5. In altre parole, l'IMC è maggiore o uguale a 18.5. Quindi il controllo al punto 9 che l'IMC sia maggiore o uguale a 18.5 è ridondante. Lo stesso vale per il punto 10. Di conseguenza, i punti 9 e 10 dell'algoritmo possono essere modificati in

9. Altrimenti se `IMC < 25` stampare "Peso normale."
10. Altrimenti se `IMC < 30` stampare "Sovrappeso."

La traduzione in codice di questo algoritmo è riportata nel Listato 3.3. Le variabili `altezza`, `massa` e `IMC` sono state dichiarate di tipo `double` per ottenere una maggiore precisione.

LISTATO 3.3 Un programma per il calcolo dell'Indice di Massa Corporea.

MyLab

```
import java.util.Scanner;

public class IMC {
    public static void main(String[] args) {
        Scanner tastiera = new Scanner(System.in);
        int kilogrammi, metri, centimetri;
        double altezza, massa, IMC;

        System.out.println("Inserire il proprio peso in kilogrammi.");
        kilogrammi = tastiera.nextInt();
        System.out.println("Inserire i metri della propria altezza" +
            " seguiti da uno spazio e dai centimetri rimanenti.");
        metri = tastiera.nextInt();
        centimetri = tastiera.nextInt();

        altezza = metri + centimetri * 0.01;
        massa = kilogrammi;
```

```
        if (IMC < 18.5)
            System.out.println("Sottopoeso.");
        else if (IMC < 25)
            System.out.println("Peso normale.");
        else if (IMC < 30)
            System.out.println("Sovrappeso.");
        else
            System.out.println("Obeso.");
    }
}
```

Esempio di output

```
Inserire il proprio peso in kilogrammi.
70
Inserire i metri della propria altezza seguiti da uno spazio e dai
centimetri rimanenti.
1 70
L'IMC è 24.221453287197232
La categoria di rischio è Peso normale.
```

3.1.5 Confronto tra stringhe

Per verificare se due valori di un tipo primitivo, per esempio due numeri, sono uguali, si utilizza l'**operatore di uguaglianza** `==`. Tuttavia tale operatore ha un significato diverso quando viene applicato agli oggetti. Dato che una stringa è un oggetto di classe `String`, l'operatore `==` non permette di verificare se due stringhe sono uguali; occorre utilizzare il metodo `equals`.

L'espressione booleana `s1.equals(s2)` restituisce `true` se le stringhe `s1` e `s2` hanno lo stesso valore, altrimenti restituisce `false`. Il programma del Listato 3.4 mostra come utilizzare il metodo `equals`. Si noti che le due espressioni:

```
s1.equals(s2)
```

e

```
s2.equals(s1)
```

sono equivalenti. Il Listato 3.4 mostra, inoltre, l'utilizzo del metodo `equalsIgnoreCase`. Questo metodo si comporta come `equals`, ma considera uguali la versione maiuscola e minuscola di una stessa lettera. Per esempio, le stringhe `"Ciao"` e `"ciao"` sono evidentemente differenti, perché i loro caratteri iniziali, `'C'` e `'c'`, sono differenti; il metodo `equalsIgnoreCase` le considera invece uguali. L'istruzione seguente, per esempio, produrrà in output la stringa `Uguali`.

```
if ("Ciao".equalsIgnoreCase("ciao"))
    System.out.println("Uguali");
```

Si noti che è consentito utilizzare la stringa tra apici `"Ciao"` nell'invocazione di `equalsIgnoreCase`. Una stringa tra apici, infatti, è un oggetto di tipo `String` e possiede tutti i metodi di un qualsiasi altro oggetto di tipo `String`.

LISTATO 3.4 Verificare l'uguaglianza tra stringhe.

MyLab

```
import java.util.Scanner;

public class UguaglianzaStringheDemo {

   public static void main(String[] args) {
      String s1, s2;

      System.out.println("Scrivi due righe di testo:");

      Scanner tastiera = new Scanner(System.in);
      s1 = tastiera.nextLine();
      s2 = tastiera.nextLine();

      if (s1.equals(s2)) {
          System.out.println("Le due righe sono uguali.");
      } else {
          System.out.println("Le due righe non sono uguali.");
      }

      if (s2.equals(s1)) {
          System.out.println("Le due righe sono uguali.");
      } else {
          System.out.println("Le due righe non sono uguali.");
      }

      if (s1.equalsIgnoreCase(s2)) {
          System.out.println("Ma sono uguali se si ignorano " +
                             "maiuscole/minuscole.");
      } else {
          System.out.println("Le due righe non sono uguali, " +
                             "nemmeno ignorando maiuscole/minuscole.");
      }
   }
}
```

Queste due invocazioni del metodo `equals` sono equivalenti.

Esempio di output

```
Scrivi due righe di testo:
Mi piace Java
MI PIACE Java
Le due righe non sono uguali
Le due righe non sono uguali
Ma sono uguali se si ignorano maiuscole/minuscole.
```

Usare l'operatore == con le stringhe

Se applicato a due stringhe (o più in generale a una qualsiasi coppia di oggetti), l'operatore == verifica solo se queste sono memorizzate nella stessa area di memoria. Questo concetto sarà ampiamente trattato nel Capitolo 8; per ora è sufficiente sapere che == non verifica che due stringhe contengano lo stesso valore.

Per il tipo di applicazioni presentate in questo capitolo, l'utilizzo di == per verificare l'uguaglianza tra stringhe potrebbe comunque fornire il risultato atteso. Tuttavia, utilizzare == per verificare l'uguaglianza tra stringhe, è una cattiva abitudine di programmazione che può portare anche a compiere errori logici. Bisognerebbe utilizzare sempre il metodo `equals` invece dell'operatore == per confrontare due stringhe. Lo stesso vale per gli altri operatori di confronto, per esempio <, e il metodo `compareTo`, presentato nei prossimi paragrafi.

I metodi `equals` ed `equalsIgnoreCase`

Per verificare l'uguaglianza tra stringhe si possono utilizzare i metodi `equals` ed `equalsIgnoreCase`.

Sintassi

stringa`.equals(`*altra_stringa*`)`

stringa`.equalsIgnoreCase(`*altra_stringa*`)`

Esempio

```
String s1 = tastiera.next(); //tastiera è un oggetto di tipo Scanner
if (s1.equals("Ciao"))
    System.out.println("La stringa e' Ciao.");
else
    System.out.println("La stringa non e' Ciao.");
```

I programmi hanno spesso bisogno di confrontare due stringhe per verificare quale preceda l'altra in ordine alfabetico. Così come l'operatore == non dovrebbe essere utilizzato per verificare l'uguaglianza tra stringhe, anche gli operatori < e > non dovrebbero essere utilizzati per verificare l'ordine alfabetico. Bisognerebbe invece utilizzare il metodo `compareTo` della classe `String`, descritto nella Figura 2.5 del Capitolo 2.

Il metodo `compareTo` confronta due stringhe per verificare il loro **ordine lessicografico**. L'ordine lessicografico è simile all'ordine alfabetico. Nell'ordine lessicografico le lettere e gli altri caratteri sono ordinati secondo la sequenza Unicode.

Se `s1` e `s2` sono due variabili di tipo `String` a cui sono stati assegnati due valori `String`, l'invocazione del metodo:

```
s1.compareTo(s2);
```

confronta l'ordine lessicografico delle due stringhe e restituisce:

- un numero negativo se `s1` precede `s2`;

- *0 se le due stringhe sono uguali;*
- *un numero positivo se* `s1` *segue* `s2`.

L'espressione booleana:

```
s1.compareTo(s2) < 0
```

è dunque vera se `s1` *precede* `s2` *in ordine lessicografico e falsa altrimenti.*
Le seguenti istruzioni, per esempio, generano un output corretto:

```
if (s1.compareTo(s2) < 0)
   System.out.println(s1 + " precede " + s2 +
                      " in ordine lessicografico ");
else
   if (s1.compareTo(s2) > 0)
         System.out.println(s1 + " segue " + s2 +
                            " in ordine lessicografico");
   else //s1.compareTo(s2) == 0
      System.out.println(s1 + " e' uguale a " + s2);
```

Si supponga invece di dover effettuare il controllo in base all'ordine alfabetico e non a quello lessicografico. Se si osserva la tabella Unicode si nota che tutte le lettere maiuscole precedono quelle minuscole. Per esempio `'Z'` *precede* `'a'` *secondo l'ordine lessicografico. Quando si confrontano due stringhe che contengono caratteri minuscoli e maiuscoli, occorre tenere conto del fatto che l'ordine lessicografico e quello alfabetico non sono uguali. Tuttavia in Unicode tutte le lettere minuscole sono in ordine alfabetico tra loro, così come quelle maiuscole. Per qualsiasi coppia di stringhe composte da soli caratteri minuscoli (o solo caratteri maiuscoli), l'ordine lessicografico corrisponde a quello alfabetico. Per confrontare in ordine alfabetico due stringhe è perciò sufficiente convertire le stringhe in modo che siano entrambe composte o solo da caratteri minuscoli o maiuscoli e quindi confrontarle in ordine lessicografico.*

Si supponga di convertire le due stringhe di caratteri, `s1` *e* `s2`*, in stringhe costituite da sole lettere maiuscole. A questo punto è sufficiente usare* `compareTo` *per verificare l'ordinamento della versione a lettere maiuscole di* `s1` *ed* `s2` *secondo l'ordine lessicografico. Le seguenti istruzioni producono l'output desiderato:*

```
String upperS1 = s1.toUpperCase();
String upperS2 = s2.toUpperCase();
if (upperS1.compareTo(upperS2) < 0)
   System.out.println(s1 + " precede " + s2 +
                      " in ordine ALFABETICO");
else
   if (upperS1.compareTo(upperS2) > 0)
      System.out.println(s1 + " segue " + s2 +
                         " in ordine ALFABETICO");
   else //s1.compareTo(s2) == 0
      System.out.println(s1 + " e' uguale a " + s2 +
                         " ignorando maiuscole e minuscole");
```

Ordine alfabetico

Per verificare se due stringhe di lettere sono in ordine alfabetico, occorre assicurarsi che tutte le lettere siano dello stesso tipo (maiuscole o minuscole) prima di utilizzare il metodo `compareTo` per confrontarle. Per raggiungere questo scopo si può utilizzare sia il metodo `toUpperCase`, sia il metodo `toLowerCase`. Senza questo passaggio, l'uso del metodo `compareTo` restituirebbe come risultato il confronto in ordine lessicografico invece che alfabetico.

3.1.6 Operatore condizionale (opzionale)

Al fine di essere compatibile con i vecchi stili di programmazione, Java presenta un operatore che costituisce una variante sintattica rispetto ad alcune forme dell'istruzione `if-else`. Per esempio, il seguente blocco di istruzioni:

```
if (n1 > n2)
    max = n1;
else
    max = n2;
```

può essere espresso nel seguente modo:

```
max = (n1 > n2) ? n1 : n2;
```

La coppia di caratteri `?` e `:` sul lato destro di questa istruzione di assegnamento è nota come **operatore condizionale** (*conditional operator*) o **operatore ternario** (*ternary operator*). L'espressione basata sull'operatore condizionale

```
(n1 > n2) ? n1 : n2;
```

inizia con un'espressione booleana, seguita da un "`?`" e da due espressioni separate dal carattere "`:`". Se l'espressione booleana (la prima) è vera, viene restituita la seconda espressione; altrimenti viene restituita la terza. Il modo più comune per utilizzare un operatore condizionale è quello mostrato in questo paragrafo e cioè utilizzarlo per assegnare a una variabile un valore scelto fra due disponibili, sulla base di una condizione booleana. Un'espressione basata sull'operatore condizionale restituisce sempre un valore e quindi è equivalente solo ad alcuni tipi di istruzioni `if-else`.

Di seguito viene presentato un altro esempio di uso dell'operatore condizionale. Si supponga che il salario settimanale di un impiegato sia pari al numero di ore di lavoro svolte moltiplicato per la paga oraria. Se poi l'impiegato lavora più di 40 ore, il tempo di lavoro eccedente viene pagato 1.5 volte la normale paga oraria. L'istruzione `if-else` che segue effettua questa computazione:

```
if (oreSvolte <= 40)
    paga = oreSvolte * pagaOraria;
else
    paga = 40 * pagaOraria + 1.5 * pagaOraria * (oreSvolte - 40);
```

Questa stessa istruzione può essere espressa usando nel seguente modo l'operatore condizionale:

```
paga = (oreSvolte <= 40) ? (oreSvolte * pagaOraria) :
       (40 * pagaOraria + 1.5 * pagaOraria * (oreSvolte - 40 ));
```

3.1.7 Il metodo `exit`

Alle volte i programmi si trovano in situazioni che rendono insensato il proseguimento dell'esecuzione. In questi casi, l'esecuzione del programma può essere terminata tramite l'invocazione del metodo `exit`:

```
System.exit(0);
```

L'istruzione precedente termina l'esecuzione di un programma Java.

Si consideri per esempio il codice seguente:

```
if (numeroVincitori == 0) {
    System.out.println("Errore: Divisione per zero.");
    System.exit(0);
} else {
    vincitaSingola = vincita / numeroVincitori;
    System.out.println("Ciascun vincitore ricevera' " +
                       vincitaSingola + " Euro");
}
```

Questo codice, normalmente, mostra la somma che riceve ciascun vincitore. Tuttavia, se il numero dei vincitori fosse 0, questo codice causerebbe un errore a run-time, a causa di una divisione per zero. Per evitare la divisione per zero, il programma controlla il numero dei vincitori e, se questo è uguale a 0, invoca il metodo `exit` per terminare l'esecuzione.

`System` è una classe della Java Class Library utilizzabile all'interno dei programmi anche senza essere caricata con un'istruzione di `import`. Il metodo `exit` appartiene alla classe `System`. Il numero 0 passato come argomento al metodo `System.exit` viene restituito al sistema operativo. La maggior parte dei sistemi operativi utilizza il valore 0 per indicare una corretta terminazione del programma e 1 per indicare una terminazione non corretta (dunque l'opposto di quello che ci si aspetterebbe). Se l'istruzione `System.exit` termina il programma in una condizione normale, l'argomento passato dovrebbe essere valore 0. In questo caso il termine *normale* indica che il programma non viola alcun vincolo. Non vuol dire che il programma abbia fatto quello che ci si aspettava. Il valore 0 quindi il valore più utilizzato come parametro nell'invocazione di `System.exit`.

Il metodo `exit`

L'invocazione del metodo `exit` termina l'esecuzione di un programma. La forma normale di un'invocazione al metodo `exit` è:

```
System.exit(0);
```

3.2 Tipo boolean

Il tipo boolean è un tipo primitivo, come i tipi int, double e char. Come per questi altri tipi, anche nel caso del tipo boolean si possono avere: valori di tipo boolean, costanti di tipo boolean, variabili di tipo boolean ed espressioni di tipo boolean. Tuttavia, il tipo boolean presenta solo due valori possibili: true (vero) e false (falso). I valori true e false possono essere utilizzati in un programma nella stessa maniera in cui si usano le costanti numeriche, come 2 o 3.45, e le costanti di tipo carattere, come 'A'.

3.2.1 Variabili booleane

Le variabili booleane, cioè le variabili di tipo boolean, possono essere utilizzate, tra le altre cose, per rendere più comprensibili i programmi. Per esempio, un programma potrebbe contenere un'istruzione come quella che segue, in cui la variabile sistemiPronti è una variabile booleana che è posta a true se i sistemi di lancio sono pronti alla partenza:

```
if (sistemiPronti)
    System.out.println("Iniziare la sequenza di lancio.");
else
    System.out.println("Abortire la sequenza di lancio.");
```

Se non si utilizzasse una variabile booleana, il codice precedente potrebbe diventare molto difficile da decifrare, come nell'esempio che segue:

```
if ((temperatura <= 100) && (propulsione >= 12000)
                         && (pressioneCabina > 30))
    System.out.println("Iniziare la sequenza di lancio.");
else
    System.out.println("Abortire la sequenza di lancio.");
```

Chiaramente la variabile booleana sistemiPronti rende più comprensibile la prima versione. Naturalmente è necessario assegnare in qualche modo il valore alla variabile booleana. Questa è un'operazione semplice, come si vedrà nei prossimi paragrafi.

Un'espressione booleana come numero > 0 restituisce uno dei due valori true o false. Per esempio, se numero > 0 è vera (true), la seguente istruzione mostra la frase Il numero e' positivo

```
if (numero > 0)
    System.out.println("Il numero e' positivo");
else
    System.out.println("Il numero e' negativo o 0");
```

Se invece numero > 0 è false, l'output del programma è la frase Il numero e' negativo o 0. Il significato di un'espressione booleana come numero > 0 è semplice da comprendere all'interno di un contesto come quello di un'istruzione if-else. Tuttavia, quando si programma con le variabili booleane bisogna utilizzare le espressioni booleane come se non avessero un contesto. Un'espressione booleana può generare un valore true o false anche se non compare in un contesto come quello del costrutto if-else.

A una variabile booleana può essere assegnato il valore di un'espressione booleana utilizzando un operatore di assegnamento. L'operazione è analoga a quella utilizzata

assegnare un valore a una variabile di qualsiasi altro tipo. Le seguenti istruzioni, per esempio, assegnano alla variabile `positivo` il valore `false`:

```
int numero = -5;
boolean positivo = (numero > 0);
```

Sebbene le parentesi non siano necessarie, facilitano l'interpretazione dell'istruzione.

Nel momento in cui viene assegnato un valore a una variabile booleana, si può utilizzare la variabile come se si stesse utilizzando un'espressione booleana. Si consideri per esempio il codice seguente:

```
boolean positivo = (numero > 0);
if (positivo)
   System.out.println("Il numero e' positivo");
else
   System.out.println("Il numero e' negativo o 0");
```

Questo codice è equivalente al blocco `if-else` presentato in precedenza.

Chiaramente in questo caso si tratta di un esempio giocattolo, tuttavia si potrebbero utilizzare delle istruzioni di questo tipo per gestire il caso in cui il valore di `numero`, e quindi dell'espressione booleana, potrebbe variare. Questo è di solito quanto succede nel contesto di un'iterazione (*loop*), argomento che sarà presentato nel prossimo capitolo.

Possono essere utilizzate allo stesso modo anche espressioni booleane più complicate. Per esempio, le istruzioni utilizzate all'inizio di questo paragrafo per verificare se i sistemi erano pronti per effettuare un lancio, dovrebbero essere precedute da un'istruzione come la seguente:

```
boolean sistemiPronti = (temperatura <= 100)
                     && (propulsione >= 12000)
                     && (pressioneCabina > 30);
```

Dare un nome alle variabili booleane

Per una variabile booleana è bene scegliere un nome che chiarisca il concetto che risulta vero quando la variabile è uguale a *true*, per esempio `positivo` (o il corrispettivo inglese `isPositive`), `sistemiPronti` (oppure `systemsAreOK` in inglese) e così via. In questo modo si può comprendere il significato della variabile booleana quando viene usata in un blocco `if-else` o in un'altra istruzione di controllo. È bene evitare tutti quei nomi che non descrivono chiaramente il significato della variabile. Non si usino, per esempio, nomi come `segnoDelNumero` oppure `statoSistema` e così via.

FAQ Perché si può scrivere `if (b)...` invece di `if (b == true)...` quando `b` è una variabile booleana?

La variabile booleana `b` è un'espressione booleana, così come l'espressione `b == true`. Se `b` è `true`, entrambe queste espressioni risultano vere. Se `b` è `false`, entrambe queste espressioni risultano false. Per questo motivo si possono utilizzare entrambe queste forme.

3.2 Tipo boolean

Il tipo `boolean` è un tipo primitivo, come i tipi `int`, `double` e `char`. Come per questi altri tipi, anche nel caso del tipo `boolean` si possono avere: valori di tipo `boolean`, costanti di tipo `boolean`, variabili di tipo `boolean` ed espressioni di tipo `boolean`. Tuttavia, il tipo `boolean` presenta solo due valori possibili: `true` (vero) e `false` (falso). I valori `true` e `false` possono essere utilizzati in un programma nella stessa maniera in cui si usano le costanti numeriche, come `2` o `3.45`, e le costanti di tipo carattere, come `'A'`.

3.2.1 Variabili booleane

Le variabili booleane, cioè le variabili di tipo `boolean`, possono essere utilizzate, tra le altre cose, per rendere più comprensibili i programmi. Per esempio, un programma potrebbe contenere un'istruzione come quella che segue, in cui la variabile `sistemiPronti` è una variabile booleana che è posta a `true` se i sistemi di lancio sono pronti alla partenza:

```
if (sistemiPronti)
    System.out.println("Iniziare la sequenza di lancio.");
else
    System.out.println("Abortire la sequenza di lancio.");
```

Se non si utilizzasse una variabile booleana, il codice precedente potrebbe diventare molto difficile da decifrare, come nell'esempio che segue:

```
if ((temperatura <= 100) && (propulsione >= 12000)
                         && (pressioneCabina > 30))
    System.out.println("Iniziare la sequenza di lancio.");
else
    System.out.println("Abortire la sequenza di lancio.");
```

Chiaramente la variabile booleana `sistemiPronti` rende più comprensibile la prima versione. Naturalmente è necessario assegnare in qualche modo il valore alla variabile booleana. Questa è un'operazione semplice, come si vedrà nei prossimi paragrafi.

Un'espressione booleana come `numero > 0` restituisce uno dei due valori `true` o `false`. Per esempio, se `numero > 0` è vera (`true`), la seguente istruzione mostra la frase `Il numero e' positivo`

```
if (numero > 0)
    System.out.println("Il numero e' positivo");
else
    System.out.println("Il numero e' negativo o 0");
```

Se invece `numero > 0` è `false`, l'output del programma è la frase `Il numero e' negativo o 0`. Il significato di un'espressione booleana come `numero > 0` è semplice da comprendere all'interno di un contesto come quello di un'istruzione `if-else`. Tuttavia, quando si programma con le variabili booleane bisogna utilizzare le espressioni booleane come se non avessero un contesto. Un'espressione booleana può generare un valore `true` o `false` anche se non compare in un contesto come quello del costrutto `if-else`.

A una variabile booleana può essere assegnato il valore di un'espressione boo... utilizzando un operatore di assegnamento. L'operazione è analoga a quella utilizzata p...

assegnare un valore a una variabile di qualsiasi altro tipo. Le seguenti istruzioni, per esempio, assegnano alla variabile `positivo` il valore `false`:

```
int numero = -5;
boolean positivo = (numero > 0);
```

Sebbene le parentesi non siano necessarie, facilitano l'interpretazione dell'istruzione.

Nel momento in cui viene assegnato un valore a una variabile booleana, si può utilizzare la variabile come se si stesse utilizzando un'espressione booleana. Si consideri per esempio il codice seguente:

```
boolean positivo = (numero > 0);
if (positivo)
    System.out.println("Il numero e' positivo");
else
    System.out.println("Il numero e' negativo o 0");
```

Questo codice è equivalente al blocco `if-else` presentato in precedenza.

Chiaramente in questo caso si tratta di un esempio giocattolo, tuttavia si potrebbero utilizzare delle istruzioni di questo tipo per gestire il caso in cui il valore di `numero`, e quindi dell'espressione booleana, potrebbe variare. Questo è di solito quanto succede nel contesto di un'iterazione (*loop*), argomento che sarà presentato nel prossimo capitolo.

Possono essere utilizzate allo stesso modo anche espressioni booleane più complicate. Per esempio, le istruzioni utilizzate all'inizio di questo paragrafo per verificare se i sistemi erano pronti per effettuare un lancio, dovrebbero essere precedute da un'istruzione come la seguente:

```
boolean sistemiPronti = (temperatura <= 100)
                        && (propulsione >= 12000)
                        && (pressioneCabina > 30);
```

Dare un nome alle variabili booleane

Per una variabile booleana è bene scegliere un nome che chiarisca il concetto che risulta vero quando la variabile è uguale a *true*, per esempio `positivo` (o il corrispettivo inglese `isPositive`), `sistemiPronti` (oppure `systemsAreOK` in inglese) e così via. In questo modo si può comprendere il significato della variabile booleana quando viene usata in un blocco `if-else` o in un'altra istruzione di controllo. È bene evitare tutti quei nomi che non descrivono chiaramente il significato della variabile. Non si usino, per esempio, nomi come `segnoDelNumero` oppure `statoSistema` e così via.

FAQ Perché si può scrivere `if (b)...` invece di `if (b == true)...` quando `b` è una variabile booleana?

La variabile booleana `b` è un'espressione booleana, così come l'espressione `b == true`. Se `b` è `true`, entrambe queste espressioni risultano vere. Se `b` è `false`, entrambe queste espressioni risultano false. Per questo motivo si possono utilizzare entrambe queste forme.

3.2.2 Regole di precedenza

Java valuta le espressioni booleane adottando la stessa strategia che utilizza per valutare le espressioni aritmetiche. Per esempio, se la variabile intera `punteggio` nella seguente espressione valesse 95:

```
(punteggio >= 80) && (punteggio < 90)
```

la prima espressione `(punteggio >= 80)` sarebbe vera, mentre la seconda espressione `(punteggio < 90)` sarebbe falsa.

L'intera espressione sarebbe quindi equivalente a:

```
true && false
```

Java combina i valori `true` e `false` in base alle regole presentate in Figura 3.7. L'espressione booleana precedente ha quindi valore `false`.

Così come si fa quando si scrivono espressioni aritmetiche, è meglio usare le parentesi per esplicitare l'ordine delle operazioni nelle espressioni booleane. Se si omettono le parentesi, Java effettua le operazioni nell'ordine specificato dalle regole di precedenza mostrate nella Figura 3.9. Questa figura è una versione estesa della Figura 2.2 e mostra la maggior parte degli operatori che si possono utilizzare. Come già indicato nel Capitolo 2, gli operatori in cima all'elenco sono quelli con maggiore precedenza. Quando l'ordine di due operazioni non viene dettato dalle parentesi, Java considera la loro precedenza ed effettua prima l'operazione con la precedenza maggiore e poi l'altra. Alcuni operatori hanno la stessa precedenza, nel qual caso vengono valutati nell'ordine in cui compaiono: gli operatori binari che hanno la stessa precedenza vengono valutati da sinistra a destra, mentre gli operatori unari che hanno la stessa precedenza vengono valutati da destra a sinistra. Si ricorda che un operatore unario presenta un solo operando, cioè si applica a un solo valore. Un operatore binario ha invece due operandi.

Si consideri l'esempio seguente (anche se è scritto con un cattivo stile di programmazione poiché non sono presenti le parentesi, non crea comunque alcun problema al computer):

```
punteggio < min / 2 - 10 || punteggio > 90
```

Di tutti gli operatori presenti nell'espressione, l'operatore di divisione (/) ha la precedenza maggiore, quindi la divisione viene eseguita per prima. Per sottolinearlo sono aggiunte due parentesi come segue:

```
punteggio < (min / 2) - 10 || punteggio > 90
```

Degli operatori rimanenti, la sottrazione (-) ha la precedenza maggiore, quindi al passo successivo viene eseguita la sottrazione:

```
punteggio < ((min / 2) - 10) || punteggio > 90
```

Degli operatori rimanenti, gli operatori di confronto > e < hanno la precedenza maggiore e quindi vengono eseguiti al passo successivo. Dato che > e < hanno la stessa precedenza, vengono eseguiti nell'ordine, da sinistra a destra:

```
(punteggio < ((min / 2) - 10)) || (punteggio > 90)
```

Infine, i risultati dei due confronti vengono combinati con l'operatore ||, che ha la precedenza minore.

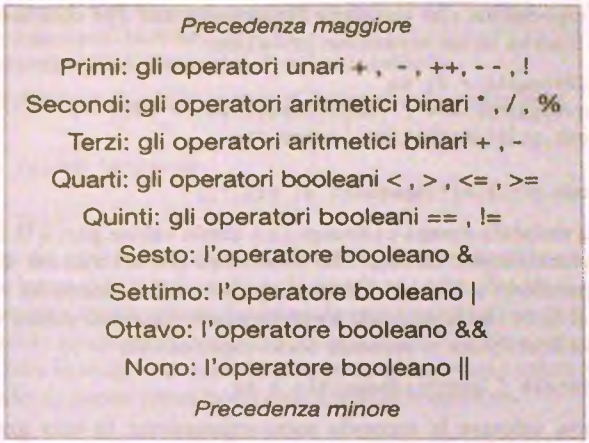


Figura 3.9 Precedenza degli operatori.

È stata quindi ottenuta una versione con parentesi dell'espressione applicando le regole di precedenza. Per il computer questa versione e quella originale senza parentesi sono equivalenti. Per rendere più comprensibili sia le espressioni aritmetiche sia quelle booleane è buona norma includere le parentesi. Tuttavia, spesso è possibile ometterle quando l'espressione contiene una sequenza semplice di operatori `&&` o `||`. L'esempio seguente presenta un buono stile di programmazione, sebbene siano state omesse alcune parentesi:

```
(temperatura > 95) || (pioggiaCaduta > 20) || (umidita >= 60)
```

A partire da quanto è stato descritto finora, si potrebbe concludere che in questo caso i tre confronti tra parentesi si verificano per primi e quindi vengono combinati con l'operatore ||. Tuttavia queste regole sono più complicate di quanto visto fin qui. Si supponga che il valore della variabile `temperatura` sia 99. In questo caso risulta chiaro che l'intera espressione booleana è vera indipendentemente dal valore delle variabili `pioggiaCaduta` e `umidita`. Questo risultato è dato dal fatto che `true || true` viene valutato come `true`, così come `true || false`. Quindi, indipendentemente dal fatto che `pioggiaCaduta > 20` sia vero, il valore di:

```
(temperatura > 95) || (pioggiaCaduta > 20)
```

risulta vero. Per lo stesso motivo, si può anche concludere che

```
true || (umidita >= 60)
```

sia vero indipendentemente dal fatto che `umidita >= 60` sia vero. Java valuta la prima espressione tra parentesi; se questa è sufficiente per conoscere il valore di verità dell'intera espressione booleana, non valuta le espressioni successive.

In questo esempio, quindi, Java non considera le espressioni che coinvolgono `pioggiaCaduta` e `umidita`. Questo modo di valutare un'espressione è detta **valutazione a corto circuito** (*short-circuit evaluation*) oppure **valutazione pigra** (*lazy evaluation*) e corrisponde al comportamento di Java ogni volta che incontra espressioni con `||` o `&&`.

Si osservi ora un'espressione che contiene l'operatore `&&`. Per dare un contesto a questa espressione viene inserita in un'istruzione `if-else`:

```
if ((compitiEseguiti > 0) &&
      ((punteggioTotale / compitiEseguiti) > 60))
   System.out.println("Ottimo lavoro.");
else
   System.out.println("Impegnati di piu.");
```

Si supponga che la variabile `compitiEseguiti` abbia valore pari a 0. La prima espressione viene quindi valutata come `false`. Dal momento che sia `false && true` sia `false && false` corrispondono a `false`, l'intera espressione booleana ha valore `false`, indipendentemente dal fatto che la seconda sotto-espressione abbia valore `true` o `false`. Per questo motivo Java non valuta la seconda sotto-espressione:

```
(punteggioTotale / compitiEseguiti) > 60
```

In questo caso, non valutare la seconda sotto-espressione fa una grande differenza, in quanto essa include una divisione per zero. Se Java avesse provato a valutarla, avrebbe generato un errore a run-time. Utilizzando la valutazione a corto circuito, Java ha prevenuto questo errore.

Java permette, però, di forzare una **valutazione completa**. Quando viene effettuata una valutazione completa, se due espressioni sono unite da un operatore *and* o da un operatore *or*, vengono *sempre valutate* entrambe, applicando poi le tabelle di verità per ottenere il valore finale dell'espressione. Per ottenere una valutazione completa in Java occorre utilizzare l'operatore `&` invece dell'operatore `&&` per il costrutto *and* e l'operatore `|` invece di `||` per il costrutto *or*.

Nella maggior parte delle situazioni, la valutazione a corto circuito e la valutazione completa restituiscono gli stessi risultati, tuttavia, come abbiamo appena visto, ci sono casi in cui una valutazione a corto circuito può evitare alcuni errori a run-time. Ci sono infine alcune situazioni in cui la valutazione completa è preferibile, tuttavia questo testo non presenterà queste tecniche. Nei prossimi capitoli verranno sempre utilizzati gli operatori `&&` e `||` per sfruttare la più veloce valutazione a corto-circuito.

Valutazione a corto-circuito

Se in un'espressione booleana nella forma *espressione_A* `||` *espressione_B*, *espressione_A* è vera, Java conclude che l'intera espressione è vera, senza valutare *espressione_B*. Allo stesso modo, se in un'espressione nella forma *espressione_A* `&&` *espressione_B*, *espressione_A* è falsa, Java conclude che l'intera espressione è falsa, senza valutare *espressione_B*.

3.2.3 Input e output di valori booleani

I valori `true` e `false` del tipo `boolean` possono essere letti come input o scritti in output nello stesso modo dei valori degli altri tipi primitivi, come `int` e `double`. Si consideri, per esempio, il seguente frammento di programma Java:

```
boolean booleanVar = false;
System.out.println(booleanVar);
System.out.println("Inserisci un valore booleano:");
```

```
Scanner tastiera = new Scanner(System.in);
booleanVar = tastiera.nextBoolean();
System.out.println("Hai inserito " + booleanVar);
```

Questo codice potrebbe produrre le seguenti interazioni con l'utente:

```
false
Inserisci un valore booleano:
true
Hai inserito true
```

Come si può vedere da questo esempio, la classe `Scanner` possiede un metodo chiamato `nextBoolean` che legge un singolo valore di tipo `boolean`. Per far sì che un valore inserito venga correttamente interpretato, l'utente deve scrivere `false` o `true`, usando lettere maiuscole o minuscole (o anche una combinazione di maiuscole e minuscole). All'interno di un programma Java invece, le costanti `true` e `false` devono essere scritte in minuscole; dunque il metodo di input `nextBoolean` è meno restrittivo.

3.3 Istruzione switch

Un blocco `if-else` multi-ramo che presenta numerosi percorsi alternativi può diventare molto difficile da leggere. Se la scelta si basa sul valore di un intero, di un singolo carattere o, dalla versione 7 di Java, di una stringa, l'**istruzione switch** può migliorare la comprensibilità del codice.

Un'istruzione switch inizia con la parola chiave `switch` seguita da un'**espressione di controllo** specificata tra parentesi:

```
switch (espressione_di_controllo) {
    ...
}
```

Il corpo dell'istruzione è sempre racchiuso fra parentesi graffe. Il Listato 3.5 mostra un esempio di istruzione `switch`, la cui espressione di controllo è la variabile `numeroNeonati`.

Tra le parentesi graffe c'è un elenco di casi; ciascun caso consiste della parola chiave `case` seguita da una costante, detta **etichetta case** (*case label*), quindi due punti e l'elenco delle istruzioni che costituiscono le azioni per quel caso:

```
case etichetta_case:
    elenco_istruzioni
```

Quando viene eseguita l'istruzione `switch`, viene valutata l'espressione di controllo, in questo esempio `numeroNeonati`. Poi viene controllato l'elenco delle alternative, finché non viene trovata un'etichetta case che corrisponde al valore dell'espressione di controllo. A questo punto viene eseguita l'azione corrispondente. Non si possono specificare etichette case duplicate, in quanto ciò genererebbe una situazione ambigua.

Si noti che l'azione per ciascun caso del Listato 3.5 termina con un'**istruzione break**, che consiste della parola `break` seguita da un punto e virgola. Quando l'esecuzione raggiunge un'istruzione `break`, l'esecuzione dell'istruzione `switch` termina. Se tra le istruzioni appartenenti a un certo caso non viene individuata alcuna istruzione `break`, l'esecuzione procede con il caso successivo, finché non viene incontrata un'istruzione `break` o anche fino alla fine dello `switch`.

Alle volte è necessario predisporre un caso senza un'istruzione `break`. Non si possono avere etichette multiple per un solo caso, però si possono elencare i casi uno di seguito all'altro in modo che generino la stessa azione. Per esempio nel Listato 3.5, sia `case 4`, sia `case 5` generano la stessa azione, in quanto `case 4` non ha alcuna istruzione di `break`; inoltre a `case 4` non è associata alcuna azione.

Se nessun caso corrisponde al valore dell'espressione di controllo, viene eseguito il **caso di default**, che inizia con la parola chiave `default` e il carattere due punti. Il caso di default è opzionale. Se si omette il caso di default e non viene trovata alcuna corrispondenza negli altri casi, non viene intrapresa alcuna azione. Sebbene il caso di default sia opzionale, è bene utilizzarlo sempre. Se si pensa che i casi coprano tutte le possibilità anche senza un caso di default, si può usare il caso di default per inserire un messaggio d'errore. Infatti, non si può essere certi di non aver dimenticato qualche caso.

Di seguito viene presentato un altro esempio di istruzione `switch` che utilizza come espressione di controllo una variabile di tipo `char`:

```
switch (tipoUova) {
    case 'A':
    case 'a':
        System.out.println("Tipo A");
        break;
    case 'C':
    case 'c':
        System.out.println("Tipo C");
        break;
    default:
        System.out.println("Compriamo solo i tipi A e C.");
        break;
}
```

In questo esempio le uova di tipo `A` e `C` possono essere indicate con caratteri maiuscoli o minuscoli. Gli altri valori per la variabile `tipoUova` sono gestiti dal caso di default.

Si noti che le etichette case non devono necessariamente essere né consecutive né ordinate. Infatti, si possono avere le etichette `'A'` e `'C'` senza l'etichetta `'B'` come nell'esempio precedente. In maniera analoga, in un'istruzione `switch` con etichette case di tipo intero, si possono avere gli interi 1 e 3, anche senza il 2. Le etichette case devono essere valori discreti, un'etichetta non può indicare un intervallo o una serie di valori. Se, per esempio, si intende eseguire la stessa azione per i valori da 1 a 4, occorre scrivere un caso per ciascun valore. Quindi, per un intervallo di valori molto ampio, un blocco `if-else` risulta molto più pratico di un'istruzione `switch`.

L'espressione di controllo di un'istruzione `switch` può anche essere più complicata di una singola variabile; per esempio, può includere degli operatori aritmetici. Prima della versione 7 del Java Development Kit (JDK), l'espressione di controllo doveva necessariamente restituire un valore di tipo intero, per esempio di tipo `int`, oppure di tipo `char`. A partire dalla versione 7 del JDK, sono consentite anche espressioni di controllo di tipo `String`. L'esempio seguente funzionerà con la versione 7 del JDK e con le successive, ma produrrà un errore in fase di compilazione con una versione precedente. Si noti che l'espressione di controllo qui è il valore di ritorno di un metodo che restituisce la variabile `risposta` in lettere minuscole.

```
System.out.println("Quale stato degli USA ha il nome composto da " +
                   una sola sillaba?");
```

```
Scanner tastiera = new Scanner(System.in);
String risposta = tastiera.next();
switch (risposta.toLowerCase()) {
    case "maine":
        System.out.println("Esatto!");
        break;
    default:
        System.out.println("Sbagliato, la risposta giusta è Maine.");
    break;
}
```

LISTATO 3.5 Istruzione switch.

```
import java.util.Scanner;
public class NasciteMultiple {
    public static void main(String[] args) {
        int numeroNeonati;
        System.out.print("Inserisci il numero di neonati: ");
        Scanner tastiera = new Scanner(System.in);
        numeroNeonati = tastiera.nextInt();
        switch (numeroNeonati) {   <-- Espressione di controllo
            case 1:   <-- Etichetta case
                System.out.println("Congratulazioni.");
                break;
            case 2:
                System.out.println("Wow. Gemelli.");
                break;   <-- Istruzione break
            case 3:
                System.out.println("Wow. Tre.");
                break;
            case 4:   <-- Istruzione case senza break
            case 5:
                System.out.print("Incredibile: ");
                System.out.println(numeroNeonati + " bambin
                break;
            default:
                System.out.println("Non ci credo!!!");
                break;
        }
    }
}
```

Esempio di output 1

```
Inserisci il numero di neonati: 1
Congratulazioni.
```

Esempio di output 2

```
Inserisci il numero di neonati: 2
Wow. Gemelli.
```

Esempio di output 3

```
Inserisci il numero di neonati: 3
Wow. Tre.
```

Esempio di output 4

```
Inserisci il numero di neonati: 4
Incredibile: 4 bambini!
```

Esempio di output 5

```
Inserisci il numero di neonati: 6
Non ci credo!!!
```

L'istruzione `switch`

Sintassi

```
switch (espressione_di_controllo) {
    case etichetta_case:
        istruzione;
        ...
        istruzione;
        break;
    case etichetta_case:
        istruzione;
        ...
        istruzione;
        break;
    default:
        istruzione;
        ...
        istruzione;
        break;
}
```

espressione_di_controllo deve essere di tipo intero come `int`, `short` o `byte`, di tipo `char` oppure di tipo `String`.

Ciascuna *etichetta_case* è una costante dello stesso tipo di *espressione_di_controllo*. Ciascun caso deve avere una *etichetta_case* diversa.

L'istruzione `break` può anche essere omessa. Senza l'istruzione `break`, l'esecuzione continua fino al prossimo `case`.

Si può utilizzare un numero arbitrario di casi.

Il caso di `default` è opzionale. Tuttavia, se non viene individuata alcuna corrispondenza, senza un caso di `default` non viene eseguita alcuna azione.

Esempio

```
int codicePosizioneSedia;
...
switch (codicePosizioneSedia)    {
    case 1:
        System.out.println("Orchestra.");
        prezzo = 40.00;
        break;
    case 2:
        System.out.println("Prima balconata.");
        prezzo = 30.00;
        break;
    case 3:
        System.out.println("Balconata.");
        prezzo = 15.00;
        break;
```

```
    default:
        System.out.println("Codice sconosciuto.");
        break;
}
```

Omettere un'istruzione `break`

Se, durante il collaudo di un programma che contiene un'istruzione `switch`, il programma stesso esegue due `case` mentre ci si aspettava che ne eseguisse uno solo, probabilmente ci si è dimenticato di inserire un'istruzione `break` dove invece era necessaria.

Omettere il caso `default`

Se i `case` di un'istruzione `switch` assegnano valori a una variabile, l'omissione del caso `default` può causare un errore di sintassi. Il compilatore, infatti, potrebbe pensare che la variabile abbia un valore indefinito dopo l'esecuzione dello `switch`. Per evitare questo errore è bene fornire il caso di `default` o inizializzare la variabile prima dell'esecuzione dello `switch`.

3.3.1 Enumerazioni

Un recensore valuta la qualità dei film catalogandoli come eccellenti, buoni o pessimi. Se si scrivesse un programma Java per organizzare queste recensioni, si potrebbero rappresentare i punteggi con gli interi 3, 2 e 1 oppure con i caratteri E, B e P. Tuttavia, qualora si definisse una variabile di tipo `int` o `char` per contenere questa valutazione, potrebbe accadere che questa finisca per contenere un valore diverso da quelli validi. Per restringere il contenuto di una variabile a un certo insieme di valori, si può definire la variabile come di tipo **enumerazione** (*enumerated data type* o *enumeration*). Un'enumerazione elenca solo i valori legittimi per una certa variabile.

Per esempio l'istruzione seguente definisce `PunteggioFilm` come enumerazione:

```
enum PunteggioFilm {E, B, P}
```

Si noti che la definizione dell'enumerazione non è seguita da alcun punto e virgola. Tuttavia, anche se venisse inserito un punto e virgola, questo non genererebbe un errore di sintassi, ma verrebbe semplicemente ignorato dal compilatore. Si noti, inoltre, che i valori E, B e P non sono racchiusi tra apici, perché non sono valori di tipo `char`.

Un'enumerazione si comporta come un tipo classe. Di conseguenza, è possibile utilizzarla per dichiarare una variabile `punteggio` come segue:

```
PunteggioFilm punteggio;
```

Gli elementi elencati tra le parentesi graffe nella definizione di `PunteggioFilm` rappresentano i valori che è possibile assegnare a `punteggio`. Per assegnare `B` a `punteggio`, si può scrivere, per esempio:

```
punteggio = PunteggioFilm.B;
```

Si noti che alla lettera `B` occorre far precedere il nome dell'enumerazione, seguito da un punto. Assegnare un valore diverso da E, B o P alla variabile `punteggio` causa un errore di sintassi.

Dopo aver assegnato un valore a una variabile di tipo enumerazione, si può usare tale valore all'interno di un'istruzione `switch` per scegliere l'azione da compiere. L'istruzione seguente, per esempio, visualizza un messaggio sulla base del valore di `punteggio`:

```
switch (punteggio) {
    case E: //Eccellente
        System.out.println("Dovete vederlo!");
        break;
    case B: //Buono
        System.out.println("E' accettabile, ma non fantastico.");
        break;
    case P: //Pessimo
        System.out.println("Evitatelo!");
        break;
    default:
        System.out.println("Valore errato.");
```

Dal momento che il tipo dell'espressione nell'istruzione `switch` è un'enumerazione, il compilatore presuppone che le etichette dei `case` appartengano all'enumerazione e quindi non occorre farle precedere dal nome dell'enumerazione. Anzi, scrivere, per esempio, `PunteggioFilm.E` produrrebbe un errore di sintassi.

Ma per far riferimento a uno dei valori dell'enumerazione al di fuori di un'istruzione `switch` occorre far precedere al valore il nome dell'enumerazione. Sebbene nell'esempio sia indicato un caso di `default`, questo in realtà è inutile. La variabile `punteggio`, infatti, non può assumere un valore diverso da quelli definiti dall'enumerazione.

I valori di un'enumerazione si comportano in maniera simile alle costanti. Per questo motivo, di solito si utilizza la stessa convenzione usata per le costanti, cioè scrivendo i valori in lettere maiuscole. I valori di un'enumerazione non devono per forza esser costituiti da singoli caratteri. Per esempio, potremmo definire l'enumerazione precedente come:

```
enum PunteggioFilm {ECCELLENTE, BUONO, PESSIMO}
```

E quindi scrivere un'istruzione di assegnamento come:

```
punteggio = PunteggioFilm.BUONO;
```

oppure un `case` come:

```
case ECCELLENTE:
```

Un'enumerazione è una classe. Pertanto può essere definita all'interno di un'altra classe, ma all'esterno delle definizioni dei metodi. Classi ed enumerazioni saranno presentate in dettaglio nel Capitolo 9.

3.4 Riepilogo

- Una struttura decisionale sceglie un'azione fra un elenco di azioni e la esegue. I costrutti `if-else` e `switch` sono strutture decisionali.

- Un'espressione booleana combina variabili e operatori di confronto, per esempio l'operatore <, per produrre un valore che può essere `true` (vero) o `false` (falso).
- Un'istruzione `if-else` verifica il valore di un'espressione booleana ed effettua un'azione tra le due possibili in base al valore vero o falso dell'espressione.
- Si può omettere il ramo `else` di un'istruzione `if-else`; si otterrà un'istruzione `if` che effettua un'azione se l'espressione booleana è vera e nessuna azione se questa è falsa.
- Un'istruzione composta è una sequenza di istruzioni Java racchiuse tra una coppia di parentesi graffe.
- Le istruzioni `if-else` possono essere annidate. Entrambi i rami dell'istruzione `if-else` possono anche contenere un'altra istruzione `if-else`. Utilizzare le parentesi per creare un'istruzione composta rende più chiaro l'intento. Se non si utilizzano le parentesi, occorre ricordare che ciascun `else` è associato all'istruzione `if` precedente più vicina e non ancora associata.
- Un struttura multi-ramo è una forma speciale di `if-else` annidato: tipicamente si scrive l'istruzione `if` subito dopo l'istruzione `else`. Quindi si scrive `if`, `else if`, `else if`,... `else`. Sebbene un'istruzione `else` finale non sia obbligatoria, di solito è buona norma includerla.
- Le espressioni booleane possono essere combinate con gli operatori logici `&&`, `||` e `!` per costituire espressioni di maggiori dimensioni. In questi casi Java utilizza la valutazione a corto circuito.
- Invocare il metodo `exit` termina l'esecuzione di un programma.
- Il valore di un'espressione booleana può essere memorizzato in una variabile di tipo `boolean`. Questa variabile è essa stessa un'espressione booleana e quindi può essere usata per controllare un'istruzione `if-else`. Una variabile booleana può essere quindi utilizzata in ogni posto in cui è lecito porre un'espressione booleana.
- L'istruzione `switch` fornisce un altro tipo di struttura decisionale multi-ramo. Questa istruzione verifica il valore di un'espressione di tipo intero, di un'espressione di tipo `char` o di un'espressione di tipo `String` ed esegue l'azione specificata nel `case` associato a quel valore. Anche in questo costrutto è buona norma fornire un `case` di default.
- Si può definire un'enumerazione i cui dati siano simili a costanti.

3.5 Esercizi

1. Scrivere un frammento di codice che verifichi che la variabile intera `punteggio` contenga un valore valido. Si supponga che i valori siano validi se sono compresi tra 0 e 100.
2. Scrivere un frammento di codice che cambi il valore intero memorizzato nella variabile x nel seguente modo: se x è pari, deve essere diviso per 2; se è dispari deve essere moltiplicato per 3 e gli deve esser sottratto 1.

3. Scrivere un programma che chieda all'utente di restituire una riposta di tipo si/no. Si supponga che il programma legga la risposta dell'utente e la inserisca nella variabile `String risposta`.
   - Se il valore di `risposta` è `si` o `s`, assegnare alla variabile `accettato` il valore `true`; altrimenti assegnare `false`.
   - Come si può modificare il codice in modo che accetti anche i valori `Si` e `S`?
4. Si consideri il seguente frammento di codice:

```
if (x >= 5)
    System.out.println("A");
else if (x < 10)
    System.out.println("B");
else
    System.out.println("C");
```

Che cosa verrebbe visualizzato nel caso in cui x avesse il valore:

a. 4
b. 5
c. 6
d. 9
e. 10
f. 11

5. Si consideri il seguente frammento di codice:

```
if (x > 5) {
    System.out.println("A");
    if (x < 10)
        System.out.println("B");
} else
    System.out.println("C");
```

Che cosa verrebbe visualizzato nel caso in cui x avesse il valore:

a. 4
b. 5
c. 6
d. 9
e. 10
f. 11

6. Si supponga di dover definire un programma per determinare i costi di servizio legati alla riscossione di assegni. Il costo del servizio dipende dall'ammontare dell'assegno. Se è minore o uguale a 10 Euro, il costo di servizio è di 1 Euro. Se è maggiore di 10 ma minore o uguale a 100 Euro il costo del servizio è pari al 10% dell'importo.

Se l'importo è maggiore di 100 Euro ma minore o uguale a 1000 Euro, il costo del servizio è pari a 5 Euro più il 5% dell'importo. Se il valore dell'importo è superiore a 1000 Euro, il costo del servizio è pari a 40 Euro più l'1% dell'importo. Scrivere un frammento di codice che permetta di computare questa cifra tramite un'istruzione `if-else` multi-ramo.

7. Qual è il valore delle seguenti espressioni booleane se `x` vale 5, `y` vale 10 e `z` vale 15?

   a. `(x < 5 && y > x)`
   b. `(x < 5 || y > x)`
   c. `(x > 3 || y < 10 && z == 15)`
   d. `(! (x > 3) && x != z || x + y == z)`

8. Il frammento di codice seguente non è compilabile. Perché?

```
if !x > x + y
   x = 2 * x;
else
   x = x + 3;
```

9. Si consideri l'espressione booleana `((x > 10) || (x < 100))`. Perché quest'espressione probabilmente non rappresenta quello che il programmatore avrebbe davvero voluto scrivere?

10. Si consideri l'espressione booleana `((2 < 5) && (x < 100))`. Perché quest'espressione probabilmente non rappresenta quello che il programmatore avrebbe davvero voluto scrivere?

11. Scrivere un'istruzione `switch` che converta un voto in lettere nel voto numerico equivalente, su una scala di quattro punti. Si assegni alla variabile `voto` il valore 4.0 per una A, 3.0 per una B, 2.0 per una C, 1.0 per una D e 0.0 per una F. Per qualsiasi altra lettera si assegni il valore 0.0 e si mostri un messaggio di errore.

12. Modificare il programma precedente in modo che consideri anche i "+" e i "-". A+ corrisponde a 4.25, A- a 3.75, B+ a 3.25, B- a 2.75 e così via.

    a. Scrivere un frammento di codice che implementi la conversione con un'istruzione `if-else` multi-ramo.

    b. Scrivere un frammento di codice che implementi la conversione usando un'istruzione `switch`.

13. Si supponga di scrivere un programma che mostra un menu con cinque scelte indicate con le lettere dalla a alla `e`. Si supponga che la scelta dell'utente sia letta e assegnata alla variabile `scelta`. Scrivere un'istruzione `switch` che mostri un messaggio che indica la scelta effettuata. Si mostri un messaggio di errore se l'utente effettua una scelta sbagliata.

14. Ripetere l'esercizio precedente definendo invece un'enumerazione da usare all'interno dell'istruzione `switch`.

15. Ripetere l'Esercizio 13, ma usando un'istruzione `if-else` multi-ramo invece di un'istruzione `switch`.

16. Data la variabile `int` di nome `temp` contenente una temperatura non negativa, scrivere un'istruzione Java che usi l'operatore condizionale per assegnare alla variabile `String etichetta` il valore `"grado"` o `"gradi"`. Lo scopo della variabile etichetta è quello di generare un output grammaticalmente corretto, come per esempio `"0 gradi"`,`"1 grado"`, `"2 gradi"` e così via.

3.6 Progetti

1. Un numero x è **divisibile** per y se il resto della divisione è pari a 0. Scrivere un programma che controlli se un numero è divisibile per un altro. Entrambi i numeri devono essere letti dalla tastiera.
2. Scrivere un programma che legga dalla tastiera tre numeri interi non negativi. Visualizzare, quindi, gli interi in ordine crescente.
3. Scrivere un programma che legga dalla tastiera tre stringhe. Visualizzare, quindi, la stringa che risulta essere la seconda in ordine lessicografico.
4. Scrivere un programma che legga una frase di una riga e quindi mostri la seguente risposta:
   - `Si` se la frase termina con un punto interrogativo (?) e il numero di caratteri è pari;
   - `No` se la frase termina con un punto interrogativo (?), ma il numero di caratteri è dispari;
   - `Wow` se la frase termina con un punto esclamativo (!);
   - le parole `Tu dici sempre` seguite dalla frase inserita racchiusa tra doppi apici in tutti gli altri casi.

   L'output del programma dovrebbe esser contenuto in una sola riga. Si noti che nell'ultimo caso l'output deve comprendere i doppi apici intorno alla stringa. In tutti gli altri casi non devono comparire apici nell'output. Il programma non deve verificare che la frase in input sia sensata.
5. Scrivere un programma che permetta all'utente di convertire una temperatura fornita in gradi da Celsius a Fahreneit e viceversa. Usare le seguenti formule:

   gradi_Celsius = 5 (gradi_Fahreneit - 32) / 9
   gradi_Fahreneit = (9 (gradi_Celsius) / 5) + 32

   Si chieda all'utente di scrivere una temperatura e una lettera. La lettera `C` o `c` indica che il valore è in gradi Celsius, la lettera `F` o `f` in Fahrenheit. Si converta la temperatura in gradi Fahrenheit se si inseriscono i Celsius e viceversa. Se vengono digitate lettere diverse da `C`, `c`, `F` o `f`, si mostri un messaggio di errore e si termini il programma.
6. Ripetere l'esercizio del Progetto 10 del Capitolo 2, ma includendo il controllo dell'input. Si mostri il resto solo se viene inserito un prezzo valido (non meno di 25 centesimi, non più di 100 centesimi e con interi multipli di 5 centesimi). Altrimenti, si mostri un messaggio d'errore diverso per ciascuno dei seguenti input non validi:

- prezzo inferiore a 25 centesimi;
- prezzo che non è un multiplo di 5;
- prezzo superiore a 1 Euro.

7. Si supponga di dover scrivere un programma che fornisce un servizio di messaggistica per i propri utenti. Si vuole dare agli utenti la possibilità di filtrare i messaggi sulla base di determinate parole indesiderate. Si supponga di considerare le parole *console*, *censore* e *magistrato* parole indesiderate. Scrivere un programma che legga una stringa dalla tastiera e verifichi se essa contiene una delle parole indesiderate. Il programma deve esser in grado di stabilire che anche la parola *cEnsore* è indesiderata anche se differisce per una maiuscola. Si estenda il programma in modo che escluda solamente le righe che contengono le parole indesiderate prese come parole a se stanti e non come parti di altre parole. La frase *Sto aspettando l'ascensore*, per esempio, *non* deve essere filtrata.

8. Scrivere un programma che legga una stringa dalla tastiera e che verifichi se questa contiene una data valida. Si mostri la data e un messaggio che indica se questa è valida. Se non è valida si mostri inoltre un messaggio che ne spieghi il motivo. La data in input deve avere il formato *gg/mm/aaaa*. Un valore valido per il mese *mm* deve essere compreso tra 1 e 12 (Gennaio è 1). Un valore valido per il giorno *gg* deve essere incluso tra 1 e un valore *corretto per quel mese*. Si deve quindi tenere in considerazione il fatto che Aprile, Giugno, Settembre e Novembre hanno 30 giorni, mentre Febbraio ne ha 28. Si consideri inoltre l'effetto degli anni bisestili, che sono tutti quelli divisibili per 4, ma non divisibili per 100 a meno che non siano divisibili per 400.

9. Riscrivere il programma per il conteggio delle calorie descritto nel Progetto 13 del Capitolo 2. Questa volta, si chieda all'utente di inserire la stringa "U" se è un uomo e "D" se è una donna. Si utilizzi quindi, per il calcolo delle calorie necessarie, solo la formula per gli uomini nel caso l'utente inserisca "U" e solo quella per le donne nel caso inserisca "D". Si stampi come nel caso precedente il numero di barrette di cioccolato richieste.

10. Ripetere il Progetto 9 chiedendo in aggiunta all'utente se è

    a. Sedentario

    b. Moderatamente attivo (svolge attività fisica occasionalmente)

    c. Attivo (svolge attività fisica 3-4 volte alla settimana)

    d. Molto attivo (svolge attività fisica tutti i giorni)

    Se l'utente risponde "Sedentario", si incrementi il valore di MB del 20 percento. Se risponde "Moderatamente attivo", si incrementi MB del 30 percento. Se risponde "Attivo", si incrementi MB del 40 percento. Infine, se l'utente risponde "Molto attivo" si incrementi MB del 50 percento. Si stampi il numero di barrette di cioccolato richieste in base al nuovo valore di MB.

Capitolo 4

Flusso di controllo: i cicli

OBIETTIVI

- Strutturare un ciclo (*loop*).
- Usare le istruzioni `while`, `do` e `for`.
- Usare l'istruzione `for-each` con le enumerazioni.
- Usare le asserzioni.

Questo capitolo porta avanti la descrizione delle strutture di controllo iniziata nel capitolo precedente. Le strutture di controllo Java che permettono di scegliere tra due o più percorsi alternativi sono estremamente importanti, ma la vera potenza di un computer sta nella sua capacità di ripetere più volte un gruppo di istruzioni. Java fornisce vari costrutti che permettono di sfruttare questa capacità.

Prerequisiti

Gli esempi in questo capitolo utilizzano l'istruzione `if-else`, l'istruzione `switch` e le enumerazioni, argomenti presentati nel Capitolo 3.

4.1 Cicli in Java

Spesso i programmi hanno la necessità di ripetere una o più azioni. Per esempio, un programma di supporto alle valutazioni didattiche potrebbe contenere strutture ramificate per assegnare un voto in lettera a uno studente sulla base dei punteggi conseguiti dallo studente stesso nei compiti e nei test. Per assegnare voti all'intera classe, il programma dovrà ripetere quest'azione per ogni studente della classe. La parte del programma che ripete un'istruzione o un gruppo di istruzioni è chiamata **ciclo** (*loop*). L'istruzione o il gruppo di istruzioni che vengono ripetuti nel ciclo sono chiamati **corpo** (*body*) del ciclo. Ogni ripetizione del corpo del ciclo è chiamata **iterazione** del ciclo.

Quando si definisce un ciclo, occorre determinare l'azione svolta nel corpo del ciclo. È necessario, inoltre, definire un meccanismo che permetta di determinare quando la ripetizione del corpo del ciclo deve terminare. Java mette a disposizione varie istruzioni per realizzare i cicli che forniscono questo meccanismo.

4.1.1 Istruzione `while`

Un modo per definire un ciclo in Java è tramite l'utilizzo dell'**istruzione `while`**, detta anche **ciclo `while`** (*while loop*). Un'istruzione `while` ripete più e più volte l'azione definita nel corpo del ciclo finché un'espressione booleana di controllo rimane vera. Questo è il motivo per cui è chiamato ciclo `while`, letteralmente *ciclo mentre*, perché il ciclo viene ripetuto *mentre* l'espressione booleana di controllo risulta vera. Quando l'espressione diviene falsa, la ripetizione termina. Il Listato 4.1 presenta un esempio giocattolo di un'istruzione `while`. L'istruzione inizia con la parola chiave `while` seguita da un'espressione booleana racchiusa tra parentesi: questa è l'espressione di controllo. Il corpo del ciclo viene ripetuto fintantoché l'espressione booleana di controllo rimane vera. Spesso il corpo del ciclo è costituito da un'istruzione composta, racchiusa tra parentesi graffe `{}`. Normalmente il corpo del ciclo contiene un'azione che può modificare il valore dell'espressione booleana di controllo da vera a falsa, così da terminare il ciclo. Di seguito viene descritta passo dopo passo l'esecuzione del ciclo `while` presentato nel Listato 4.1.

Si consideri la prima esecuzione del ciclo rappresentato nel Listato 4.1. L'utente digita il numero 2, quindi 2 diventa il valore della variabile `numero`. L'espressione booleana di controllo è:

```
conteggio <= numero
```

Dal momento che la variabile `conteggio` vale 1 e la variabile `numero` vale 2, l'espressione booleana è vera e quindi viene eseguito il corpo del ciclo mostrato qui di seguito:

```
{
    System.out.print(conteggio + ", ");
    conteggio++;
}
```

MyLab

LISTATO 4.1 Un esempio di ciclo `while`.

```
import java.util.Scanner;

public class WhileDemo {
    public static void main(String[] args) {
        int conteggio, numero;

        System.out.println("Inserisci un numero");
        Scanner tastiera = new Scanner(System.in);
        numero = tastiera.nextInt();

        conteggio = 1;
        while (conteggio <= numero) {
            System.out.print(conteggio + ", ");
            conteggio++;
        }
        System.out.println();
        System.out.println("Sorpresa!");
    }
}
```

Esempio di output 1

```
Inserisci un numero
2
1, 2,
Sorpresa!
```

Esempio di output 2

```
Inserisci un numero
3
1, 2, 3,
Sorpresa!
```

Esempio di output 3

```
Inserisci un numero
0
```

← Il corpo del ciclo viene eseguito zero volte.

```
Sorpresa!
```

Il corpo del ciclo visualizza il valore della variabile `conteggio`, che è 1, e poi incrementa il valore di `conteggio` di 1 unità, facendole assumere il valore 2.

Dopo un'iterazione del corpo del ciclo, l'espressione booleana di controllo viene nuovamente valutata. Dal momento che la variabile `conteggio` vale 2 e la variabile `numero` vale 2, l'espressione booleana risulta ancora vera. Il corpo del ciclo viene quindi eseguito una seconda volta. Di nuovo viene mostrato sullo schermo il valore della variabile `conteggio`, che è 2, e viene nuovamente incrementato di 1 unità il valore della variabile `conteggio`, facendole assumere il valore 3.

Dopo la seconda iterazione del corpo del ciclo, viene nuovamente valutata l'espressione booleana di controllo. La variabile `conteggio` ora vale 3, mentre il valore della variabile `numero` è sempre 2. Per questo motivo, ora l'espressione booleana di controllo `conteggio <= numero` risulta falsa. Per questo motivo il ciclo `while` termina e il programma prosegue eseguendo le due istruzioni `System.out.println` che seguono il ciclo `while`. La prima di queste due istruzioni termina la riga che contiene la sequenza

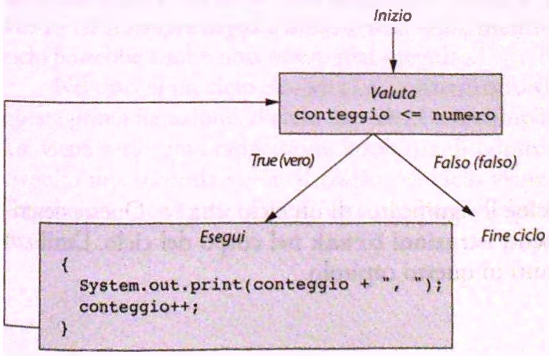


Figura 4.1 Le azioni del ciclo `while` nel Listato 4.1.

di numeri mostrati dal ciclo `while` facendo pertanto iniziare una nuova riga; la seconda mostra il messaggio `Sorpresa!` La Figura 4.1 riassume le azioni svolte in questo ciclo.

Il ciclo `while`

Sintassi

```
while (espressione_booleana)
    corpo
```

Il *corpo* del ciclo può essere una singola istruzione o, come succede più di frequente, un'istruzione composta costituita da un elenco di istruzioni racchiuse tra parentesi graffe `{}`.

Esempio

```
//Estrae il primo numero positivo introdotto alla tastiera
int prossimo = 0;
while (prossimo <= 0)
    prossimo = tastiera.nextInt(); //tastiera e' un oggetto di tipo Scanner
```

Esempio

```
//Somma gli interi positivi letti in input finche'
//non ne viene inserito uno negativo
int totale = 0;
int prossimo = tastiera.nextInt();
while (prossimo >= 0) {
    totale = totale + prossimo;
    prossimo = tastiera.nextInt();
}
```

Tutti i cicli `while` sono strutturati in un modo analogo a quello dell'esempio nel Listato 4.1. Il corpo del ciclo può anche essere costituito da una sola istruzione, anche se di solito è formato da un'istruzione composta, come nel Listato 4.1. La forma più comune di un ciclo `while` è pertanto la seguente:

```
while (espressione_booleana) {
        prima_istruzione
        seconda_istruzione
        ...
        ultima_istruzione
}
```

La Figura 4.2 riassume la semantica, cioè il significato, di un ciclo `while`. Questa descrizione presuppone che non siano presenti istruzioni `break` nel corpo del ciclo. L'utilizzo di tali istruzioni sarà descritto più avanti in questo capitolo.

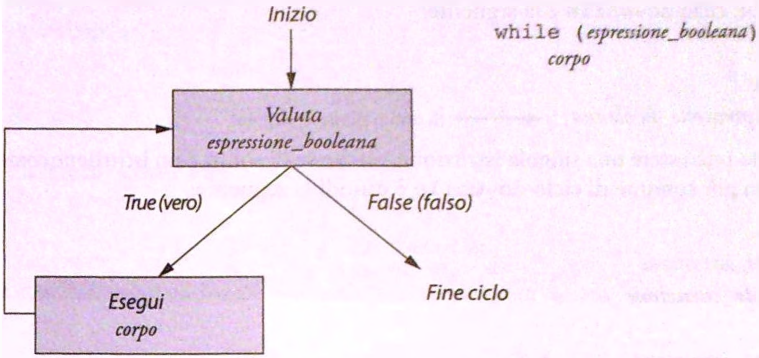


Figura 4.2 La semantica dell'istruzione `while`.

Un ciclo `while` può anche eseguire zero iterazioni

Il corpo di un ciclo `while` può anche essere "eseguito" zero volte. Quando viene eseguito un ciclo `while`, la prima operazione effettuata è la verifica dell'espressione booleana di controllo. Se essa risulta subito falsa, il corpo del ciclo non viene eseguito nemmeno una volta. Sebbene questo comportamento possa apparire strano, in realtà non è così: si potrebbe avere la necessità di impiegare un ciclo `while` da eseguire zero o più volte a seconda dell'input inserito dall'utente. Si consideri, per esempio, il caso di un ciclo che somma tutte le spese effettuate da una persona in una giornata. Se tale persona non ha effettuato alcuna spesa, ci si aspetta che il corpo del ciclo non venga eseguito nemmeno una volta. Il terzo esempio di output del Listato 4.1 mostra proprio il caso di un ciclo `while` che esegue il corpo del ciclo zero volte.

4.1.2 Istruzione `do-while`

L'**istruzione** `do-while`, detta anche **ciclo** `do-while` (*do-while loop*), è molto simile all'istruzione `while`. La differenza principale consiste nel fatto che il corpo di un ciclo `do-while` *viene sempre eseguito almeno una volta,* mentre nel caso del ciclo `while` il corpo del ciclo potrebbe anche non essere mai eseguito.

Nel caso di un ciclo `do-while`, innanzitutto viene eseguito il corpo del ciclo. Dopo questa prima iterazione, il ciclo `do-while` si comporta allo stesso modo di un ciclo `while`: viene verificata l'espressione booleana di controllo; se è vera, il corpo del ciclo viene eseguito una seconda volta. Il corpo del ciclo viene eseguito più e più volte fintantoché l'espressione booleana rimane vera. Appena l'espressione booleana risulta falsa, il ciclo termina.

La sintassi di un ciclo do-while è la seguente:

```
do
    corpo
while (espressione_booleana); ←—— Si noti il punto e virgola!
```

Il *corpo* del ciclo può essere una singola istruzione, anche se di solito è un'istruzione composta. La forma più comune di ciclo do-while è quindi la seguente:

```
do {
    prima_istruzione
    seconda_istruzione
    ...
    ultima_istruzione
} while (espressione_booleana);
```

Si noti il punto e virgola dopo la parentesi tonda chiusa che segue *espressione_booleana*.

Il Listato 4.2 contiene un semplice ciclo do-while che è simile al ciclo while del Listato 4.1, ma che produce un output diverso per gli stessi valori di input. In particolare, nell'*Esempio di output 3* si nota che il corpo del ciclo viene eseguito anche se l'espressione booleana è falsa sin dall'inizio. Questo accade proprio perché il corpo di un ciclo do-while viene sempre eseguito almeno una volta. La Figura 4.3 riassume le azioni svolte in questo ciclo.

LISTATO 4.2 Esempio di ciclo do-while.

```
import java.util.Scanner;

public class DoWhileDemo {
    public static void main(String[] args) {
        int conteggio, numero;

        System.out.println("Inserisci un numero");
        Scanner tastiera = new Scanner(System.in);
        numero = tastiera.nextInt();

        conteggio = 1;
        do {
            System.out.print(conteggio + ", ");
            conteggio++;
        } while (conteggio <= numero);

        System.out.println();
        System.out.println("Sorpresa!");
    }
}
```

Esempio di output 1

```
Inserisci un numero
2
1, 2,
Sorpresa!
```

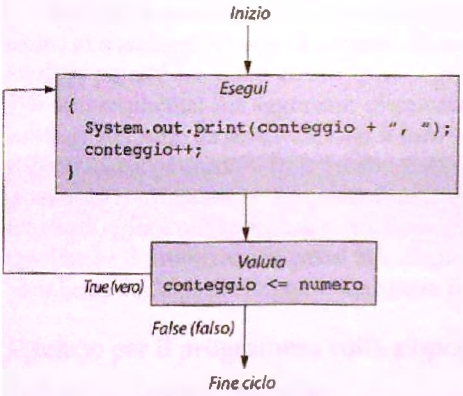
Esempio di output 2

```
Inserisci un numero
3
1, 2, 3,
Sorpresa!
```

Esempio di output 3

```
Inserisci un numero
0
1,
Sorpresa!
```

Il corpo del ciclo viene sempre eseguito almeno una volta.




```
do {
   System.out.print(conteggio + ", ");
   conteggio++;
} while (conteggio <= numero);
```

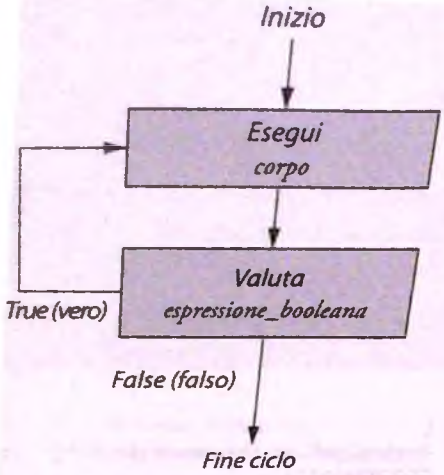
Figura 4.3 Le azioni del ciclo `do-while` nel Listato 4.2.

Il ciclo `do-while` presentato nel Listato 4.2 può essere riscritto come un ciclo `while` equivalente:

```
{
   System.out.print(conteggio + ", ");
   conteggio++;
} while (conteggio <= numero) {
   System.out.print(conteggio + ", ");
   conteggio++;
}
```

Sebbene questo modo di riscrivere i cicli `do-while` non costituisca una buona abitudine di programmazione, questo esempio aiuta a illustrare la differenza esistente tra questi due costrutti iterativi. Con un ciclo `do-while`, il corpo del ciclo viene sempre eseguito almeno una volta; con un ciclo `while` il corpo del ciclo potrebbe non venir mai eseguito.

La Figura 4.4 illustra la semantica di un ciclo `do-while`. Come per la semantica di un ciclo `while` mostrata nella Figura 4.2, anche in questo caso si presuppone che il corpo del ciclo non contenga istruzioni `break`.




```
do {
    corpo
} while (espressione_booleana);
```

Figura 4.4 La semantica dell'istruzione `do-while`.

L'istruzione do-while

Sintassi

```
do {
    corpo
} while (espressione_booleana);
```

Il *corpo* del ciclo può essere una sola istruzione o, come succede più di frequente, un'istruzione composta, costituita da una serie di istruzioni racchiuse tra parentesi graffe `{}`. Il *corpo* del ciclo viene sempre eseguito almeno una volta. Si noti il punto e virgola posto alla fine dell'istruzione.

Esempio

```
//Carica il prossimo numero intero positivo digitato come input
int prossimo;
do
    prossimo = tastiera.nextInt(); //tastiera e' un oggetto di tipo Scanner
while (prossimo <= 0);
```

Esempio

```
//Somma gli interi positivi letti finche' non viene letto un numero negativo
int totale = 0;
int prossimo = 0;
do {
    totale = totale + prossimo;
    prossimo = tastiera.nextInt();
} while (prossimo > 0);
```

ESEMPIO DI PROGRAMMAZIONE
INFESTAZIONE DI SCARAFAGGI

Si supponga che una cittadina sia stata infestata dagli scarafaggi. Chiaramente non è una bella situazione, ma fortunatamente l'azienda *Esperti di disinfestazione* ha il trattamento per eliminare gli scarafaggi dalle case. Purtroppo, però, i cittadini sottovalutano il problema e potrebbero ritrovarsi l'abitazione completamente invasa dagli scarafaggi. Per questo motivo, l'azienda ha collocato un computer presso il supermercato locale con lo scopo di spiegare quanto grave potrebbe diventare il problema. Il programma installato in questo computer calcola il numero di settimane che una popolazione di scarafaggi impiega per riempire completamente una casa dal pavimento al soffitto.

Sebbene la popolazione di scarafaggi cresca in maniera relativamente lenta, trattandosi di scarafaggi il tasso di crescita è comunque critico. Se restano incontrollati, gli scarafaggi potrebbero quasi raddoppiare ogni settimana: il loro tasso di crescita è pari al 95% ogni settimana! Ad aggravare ulteriormente la situazione, le dimensioni di questi scarafaggi non sono da sottovalutare: il loro volume è di circa 0,8 cm^3. Per semplificare il problema, il programma si basa su alcuni presupposti: che la casa non sia arredata e che gli scarafaggi riempiano la casa interamente, senza lasciare alcuno spazio tra loro. Chiaramente la realtà è più complicata, tuttavia questa semplificazione permette di affrontare agevolmente il problema. Si pensi ora ai passi necessari per risolvere il problema. Una prima bozza dell'algoritmo potrebbe essere la seguente:

Algoritmo per il programma sulla popolazione di scarafaggi (prima bozza)

1. Acquisisci il volume della casa.
2. Acquisisci il numero di scarafaggi presenti inizialmente nella casa.
3. Computa il numero di settimane necessarie per riempire la casa di scarafaggi.
4. Mostra i risultati.

Dal momento che si conoscono sia il tasso di crescita della popolazione di scarafaggi, sia le loro dimensioni medie, questi valori non saranno richiesti in input, ma saranno costanti del programma. Sebbene questo algoritmo sembri sufficiente, in realtà non è abbastanza completo per poter iniziare a programmare. Per esempio, il Passo 3, sebbene sia il cuore della soluzione, non presenta alcun suggerimento sul modo in cui effettuare i calcoli. Per questo motivo, è necessario pensare ulteriormente alla risoluzione del problema. Dal momento che occorre calcolare un numero che rappresenta le settimane che passano prima che la casa sia riempita dagli scarafaggi, è possibile utilizzare un contatore. Il valore iniziale di questo contatore è chiaramente 0. Poiché si conoscono il volume dello scarafaggio medio e il numero iniziale di scarafaggi, si può calcolare il loro volume totale semplicemente moltiplicando questi due valori. Questo calcolo permette di ottenere il volume della popolazione di scarafaggi alla settimana 0. Questo risultato, tuttavia, non costituisce il risultato finale, a meno che gli scarafaggi non abbiano già riempito la casa. Se gli scarafaggi non hanno riempito la casa, è perciò necessario aggiungere al risultato finale anche il volume di scarafaggi nati durante la prima settimana. Se gli scarafaggi non hanno riempito la casa, è necessario aggiungere il volume degli scarafaggi nati durante la seconda settimana, il tutto finché la casa non viene riempita. È chiaro, quindi, che è necessario usare un ciclo.

Prima di utilizzare un ciclo, occorre domandarsi quale sia quello più adatto a risolvere il problema. In alcuni casi si potrebbe non essere in grado di scegliere il tipo di iterazione in questa fase di ideazione dell'algoritmo. Tuttavia, in questo caso si è in grado di fare una scelta. In precedenza si è visto che l'infestazione iniziale avrebbe già potuto riempire la casa. In tal caso, non sarebbe necessario effettuare alcun calcolo: non sarebbe quindi necessario eseguire il corpo del ciclo. Pertanto il ciclo `while` risulta essere la scelta ottimale (rispetto al ciclo `do-while`).

A prescindere dal fatto che il tipo di ciclo sia stato identificato o meno, occorre inizialmente individuare alcune costanti e variabili:

`TASSO_CRESCITA` – Tasso di crescita settimanale della popolazione di scarafaggi (una costante, 0.95).

`VOLUME_SCARAFAGGIO` – Volume medio di uno scarafaggio (una costante, 0.76).

`volumeCasa` – Il volume dell'abitazione.

`popolazioneIniziale` – La popolazione iniziale degli scarafaggi.

`conteggioSettimane` – Il contatore della settimana corrente.

`popolazione` – Il numero di scarafaggi attuale.

`volumeTotaleScarafaggi` – Il volume totale degli scarafaggi nella casa.

`nuoviScarafaggi` – Gli scarafaggi nati questa settimana.

`volumeNuoviScarafaggi` – Il volume degli scarafaggi nati questa settimana.

Non si dovrebbe pensare a tutte le variabili necessarie in una volta sola: a mano a mano che lo pseudocodice viene raffinato, si possono aggiungere nuove variabili all'elenco. Mantenere un elenco delle variabili e del loro significato è spesso una scelta molto utile per progettare e implementare un programma.

Prendere note

Prima di iniziare a scrivere la soluzione di un problema in Java, è bene progettare adeguatamente il programma. In particolare, è opportuno scrivere lo pseudocodice, disegnare i grafici e quindi definire l'elenco delle variabili. È necessario, cioè, organizzare i pensieri e scriverli su carta. In questo modo il programma Java finale risulterà ben organizzato.

Si può quindi sostituire il Passo 3 dell'algoritmo con i seguenti passi:

3a. `conteggioSettimane = 0`
3b. Ripeti fino a quando la casa non sarà piena di scarafaggi.

```
{
    nuoviScarafaggi = popolazione * TASSO_CRESCITA
    volumeNuoviScarafaggi = nuoviScarafaggi * VOLUME_SCARAFAGGIO
    popolazione = popolazione + nuoviScarafaggi
    volumeTotaleScarafaggi = volumeTotaleScarafaggi +
                             volumeNuoviScarafaggi
    conteggioSettimane = conteggioSettimane + 1
}
```

Osservando il Passo 3b, si nota che Java non offre un costrutto *ripeti fino a quando* (*repeat until*). Tuttavia, la frase *ripeti fino a quando la casa sarà piena di scarafaggi* può essere parafrasata come *ripeti fintantoché (while) la casa non è piena di scarafaggi* oppure come *ripeti fintantoché (while) il volume degli scarafaggi è minore del volume della casa.*

Quest'ultima considerazione chiarisce che è possibile usare il costrutto:

```
while (volumeTotaleScarafaggi < volumeCasa)
```

L'assemblaggio delle istruzioni definite finora ci permette di ottenere il seguente algoritmo:

Algoritmo per il programma sulla popolazione di scarafaggi

1. Leggi `volumeCasa`
2. Leggi `popolazioneIniziale`
3. `popolazione = popolazioneIniziale`
4. `volumeTotaleScarafaggi = popolazione * VOLUME_SCARAFAGGIO`
5. `conteggioSettimane = 0`
6. `while (volumeTotaleScarafaggi < volumeCasa) {`
   `nuoviScarafaggi = popolazione * TASSO_CRESCITA`
   `volumeNuoviScarafaggi = nuoviScarafaggi * VOLUME_SCARAFAGGIO`
   `popolazione = popolazione + nuoviScarafaggi`
   `volumeTotaleScarafaggi = volumeTotaleScarafaggi +`
   `volumeNuoviScarafaggi`
   `conteggioSettimane = conteggioSettimane + 1`
   `}`
7. Visualizza `popolazioneIniziale`, `volumeCasa`, `conteggioSettimane` e `volumeTotaleScarafaggi`

Il ciclo aggiorna la popolazione di scarafaggi, il loro volume e il contatore delle settimane. Dal momento che il tasso di crescita e il volume di uno scarafaggio sono entrambi numeri interi positivi, il valore di `popolazione` (e, di conseguenza, il valore di `volumeTotaleScarafaggi`) cresce a ogni iterazione. Questo implica che dopo una serie di iterazioni, il valore di `volumeTotaleScarafaggi` sarà maggiore del valore di `volumeCasa` e, di conseguenza, l'espressione booleana di controllo:

```
volumeTotaleScarafaggi < volumeCasa
```

diventerà `false` e farà terminare il ciclo.

La variabile `conteggioSettimane` parte da 0 e viene incrementata di 1 unità a ogni iterazione del ciclo. Per questo motivo, quando il ciclo termina, il valore di `conteggioSettimane` indica il numero di settimane necessarie prima che gli scarafaggi superino il volume della casa.

Il Listato 4.3 mostra questo programma Java e un output di esempio.

LISTATO 4.3 Programma per calcolare la popolazione di scarafaggi.


```
import java.util.Scanner;

/**
  Programma che calcola quanto tempo impiega una popolazione di scarafaggi
  a riempire completamente una casa dal pavimento al soffitto.
*/
```

```
public class Scarafaggi {

    public static final double TASSO_CRESCITA = 0.95; //95% per settimana
    public static final double VOLUME_SCARAFAGGIO = 0.00000076; //in metri cubi

    public static void main(String[] args) {
        System.out.println("Inserisca il volume della sua abitazione");
        System.out.print("in metri cubi: ");
        Scanner tastiera = new Scanner(System.in);
        double volumeCasa = tastiera.nextDouble();

        System.out.println("Inserisca una stima degli");
        System.out.print("scarafaggi nella sua casa: ");
        int popolazioneIniziale = tastiera.nextInt();
        int conteggioSettimane = 0;
        double popolazione = popolazioneIniziale;
        double volumeTotaleScarafaggi = popolazione * VOLUME_SCARAFAGGIO;
        double nuoviScarafaggi, volumeNuoviScarafaggi;

        while (volumeTotaleScarafaggi < volumeCasa) {
            nuoviScarafaggi = popolazione * TASSO_CRESCITA;
            volumeNuoviScarafaggi = nuoviScarafaggi * VOLUME_SCARAFAGGIO;
            popolazione = popolazione + nuoviScarafaggi;
            volumeTotaleScarafaggi = volumeTotaleScarafaggi +
                                     volumeNuoviScarafaggi;
            conteggioSettimane++;
        }

        System.out.println("Con una popolazione iniziale di " +
                           popolazioneIniziale + " scarafaggi");
        System.out.println("e un'abitazione con un volume di " +
                           volumeCasa + " metri cubi,");
        System.out.println("dopo " + conteggioSettimane + " settimane,");
        System.out.println("la sua casa ospiterà una popolazione di " +
                           (int)popolazione + " scarafaggi.");
        System.out.println("Che riempira' un volume di " +
                           (int)volumeTotaleScarafaggi + " metri cubi.");
        System.out.println("E' meglio chiamare gli " +
                           "Esperti della Disinfestazione.");

    }
}
```

Esempio di output

```
Inserisca il volume della sua abitazione
in metri cubi: 240
```

```
Inserisca una stima degli
scarafaggi nella sua casa: 100
Con una popolazione iniziale di 100 scarafaggi
e un'abitazione con un volume di 240.0 metri cubi,
dopo 23 settimane,
la sua casa ospiterà una popolazione di 468593285 scarafaggi.
Che riempira' un volume di 356 metri cubi.
E' meglio chiamare gli Esperti della Disinfestazione.
```

Cicli infiniti

Un difetto molto comune dei programmi è rappresentato dai cicli che non terminano mai, ma che continuano a ripetere all'infinito il proprio corpo. Un ciclo che ripete il proprio corpo senza mai terminare è chiamato **ciclo infinito** (*infinite loop*). Normalmente le istruzioni presenti nel corpo di un ciclo `while` o di un ciclo `do-while` alterano in qualche modo una o più variabili, così che, prima o poi, l'espressione booleana di controllo diventi falsa. Tuttavia, se la o le variabili non cambiano in modo corretto, si potrebbe generare un ciclo infinito.

Si consideri, per esempio, una versione leggermente modificata del programma rappresentato nel Listato 4.3. Si supponga che la cittadina sia stata infestata da rane mangiascarafaggi. Queste rane mangiano così tanti scarafaggi che la popolazione degli scarafaggi decresce; in altre parole, gli scarafaggi hanno un tasso di crescita negativo. Per riflettere questo cambiamento nel programma si può modificare solo la definizione della costante `TASSO_CRESCITA` e ricompilare il programma:

```
public static final double TASSO_CRESCITA = -0.05;
//-5% a settimana
```

Se ci si limita a modificare questa costante e a eseguire il programma, il ciclo `while` diventa infinito se il numero iniziale di scarafaggi è contenuto. Dato che il numero totale di scarafaggi, e quindi il volume degli scarafaggi, *decresce* di continuo, l'espressione booleana di controllo, `volumeTotaleScarafaggi < volumeCasa` rimane sempre vera, facendo sì che il ciclo non termini mai.

Alcuni cicli infiniti non vengono eseguiti veramente all'infinito, ma faranno comunque terminare l'esecuzione del programma dopo aver esaurito qualche risorsa del sistema. Tuttavia, alcuni cicli infiniti potrebbero, in effetti, essere eseguiti per sempre. Per poter terminare un programma che contiene un ciclo infinito occorre imparare a forzare la terminazione di un programma. La tecnica da usare dipende dal sistema operativo. Nella maggior parte dei sistemi operativi (ma non in tutti) un programma può essere terminato con la combinazione di tasti Ctrl + C.

In alcuni casi, il programmatore potrebbe aver scritto appositamente un ciclo infinito. Per esempio, i Bancomat potrebbero essere controllati da un ciclo infinito che gestisce depositi e prelievi. Tuttavia, per gli altri casi di studio trattati finota, i cicli infiniti sono da considerarsi errori.

ESEMPIO DI PROGRAMMAZIONE
CICLI ANNIDATI

Il corpo di un ciclo può contenere qualsiasi tipo di istruzione. In particolare, un ciclo può anche essere contenuto nel corpo di un altro ciclo. Il programma rappresentato nel Listato 4.4, per esempio, usa un'istruzione `while` per calcolare la media di un elenco di voti non negativi. Il programma chiede all'utente di inserire tutti i punteggi, seguiti da un numero negativo che fa da valore sentinella che indica la fine dei dati. L'istruzione `while` è posta all'interno del corpo di un'istruzione `do-while`, in modo che l'utente possa ripetere l'intero processo per un altro esame e poi per un altro ancora e così via, finché l'utente non intende terminare il programma.

MyLab

LISTATO 4.4 Cicli annidati.

```
import java.util.Scanner;

/**
 Calcola la media di un elenco di voti non negativi relativi a un esame.
 Ripete il calcolo per piu' esami fino a quando l'utente non chiede di
 fermarsi.
 */
public class MediaEsami {

    public static void main(String[] args) {
        System.out.println("Questo programma calcola la media");
        System.out.println("di un elenco di voti non negativi.");
        double somma;
        int numeroStudenti;
        double successivo;
        String risposta;
        Scanner tastiera = new Scanner(System.in);

        do {
            System.out.println( );
            System.out.println("Inserisci tutti i voti di cui");
            System.out.println("vuoi calcolare la media.");
            System.out.println("Poi inserisci un numero negativo");
            System.out.println("dopo aver inserito tutti i voti.");
            somma = 0;
            numeroStudenti = 0;
            successivo = tastiera.nextDouble();
            while (successivo >= 0) {
                somma = somma + successivo;
                numeroStudenti++;
                successivo = tastiera.nextDouble();
            }
```

```
            if (numeroStudenti > 0) {
                System.out.println("La media e' " +
                                   (somma / numeroStudenti));
            } else {
                System.out.println("La media non e' calcolabile.");
            }
            System.out.println("Vuoi calcolare la media di un altro " +
                               "esame?");
            System.out.println("Scrivi si o no.");
            risposta = tastiera.next();
        } while (risposta.equalsIgnoreCase("si"));
    }
}
```

Esempio di output

```
Questo programma calcola la media
di un elenco di voti non negativi.

Inserisci tutti i voti di cui
vuoi calcolare la media.
Poi inserisci un numero negativo
dopo aver inserito tutti i voti.
30
25
30
25
-1
La media e' 27.5
Vuoi calcolare la media di un altro esame?
Scrivi si o no.
si

Inserisci tutti i voti di cui
vuoi calcolare la media.
Poi inserisci un numero negativo
dopo aver inserito tutti i voti.
30
27
24
-1
La media e' 27.0
Vuoi calcolare la media di un altro esame?
Scrivi si o no.
no
```

4.1.3 Istruzione `for`

L'**istruzione for**, detta anche **ciclo for**, permette di scrivere facilmente un ciclo controllato da un contatore. Lo pseudocodice che segue, per esempio, definisce un ciclo la cui iterazione si ripete per tre volte e controllato dal contatore `conteggio`:

```
Fai quanto segue per ciascun valore di conteggio da 1 a 3:
    Visualizza conteggio
```

Questo pseudocodice può essere espresso in Java con le seguenti istruzioni:

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

Questa istruzione `for` genera il seguente output:

```
1
2
3
```

Al termine dell'istruzione `for` vengono eseguite le istruzioni definite dopo il corpo del ciclo. In questo semplice esempio di istruzione `for`, il corpo del ciclo è costituito dall'istruzione:

```
System.out.println(conteggio);
```

L'iterazione del corpo del ciclo è controllata dalla riga:

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

La prima delle tre istruzioni tra parentesi, `conteggio = 1`, indica l'operazione da svolgere prima che il corpo del ciclo venga eseguito per la prima volta. La terza espressione, `conteggio++`, viene eseguita dopo ogni iterazione del corpo del ciclo. L'espressione nel mezzo, `conteggio <= 3`, è un'espressione booleana che determina quando il ciclo deve terminare. Si comporta, quindi, allo stesso modo dell'espressione booleana di controllo di un ciclo `while`. Il corpo del ciclo viene perciò eseguito fintantoché il valore della variabile `conteggio` rimane minore o uguale a 3. Per parafrasare quanto detto si può dire che il ciclo `for` seguente

```
for (conteggio = 1; conteggio <= 3; conteggio++)
    corpo
```

è equivalente al seguente ciclo `while`

```
conteggio = 1;
while (conteggio <= 3){
    corpo
    conteggio++;
}
```

La sintassi di un'istruzione `for` è la seguente:

```
for (inizializzazione; espressione_booleana; aggiornamento)
    corpo
```

Il *corpo* del ciclo può essere un'istruzione singola, come nell'esempio precedente, o, come accade di frequente, un'istruzione composta. Perciò, la forma più comune di un'istruzione

for può essere descritta in questo modo:

```
for (inizializzazione; espressione_booleana; aggiornamento) {
    prima_istruzione
    ...
    ultima_istruzione
}
```

Quando viene eseguito, un ciclo `for` si comporta in maniera simile a un'istruzione `while`.
Di conseguenza, il ciclo `for` sopra riportato è equivalente alle seguenti istruzioni:

```
inizializzazione
while (espressione_booleana){
    prima_istruzione
    ...
    ultima_istruzione
    aggiornamento
}
```

Dal momento che un'istruzione `for` rappresenta di fatto un sinonimo di un'istruzione `while`, potrebbe anche non eseguire mai il corpo del suo ciclo. Il Listato 4.5 mostra un esempio di istruzione `for`. Le azioni eseguite dal ciclo sono riassunte nella Figura 4.5. La Figura 4.6 descrive più in generale la semantica di un ciclo `for`.

LISTATO 4.5 Esempio di ciclo for.

```
public class ForDemo {

    public static void main(String[] args) {
        int contoAllaRovescia;

        for (contoAllaRovescia = 3; contoAllaRovescia >= 0; contoAllaRovescia--){
            System.out.println(contoAllaRovescia);
            System.out.println("attendere...");
        }

        System.out.println("Partito!");
    }
}
```

Esempio di output

```
3
attendere...
2
attendere...
1
attendere...
0
attendere...
Partito!
```

```
for (contoAllaRovescia = 3; contoAllaRovescia >= 0; contoAllaRovescia--) {
    System.out.println(contoAllaRovescia);
    System.out.println("attendere...");
}
```

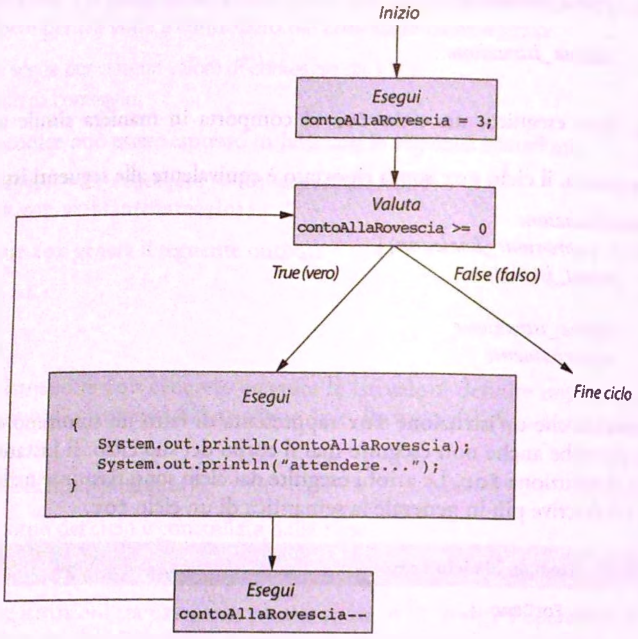


Figura 4.5 Le azioni svolte dal ciclo `for` nel Listato 4.5.

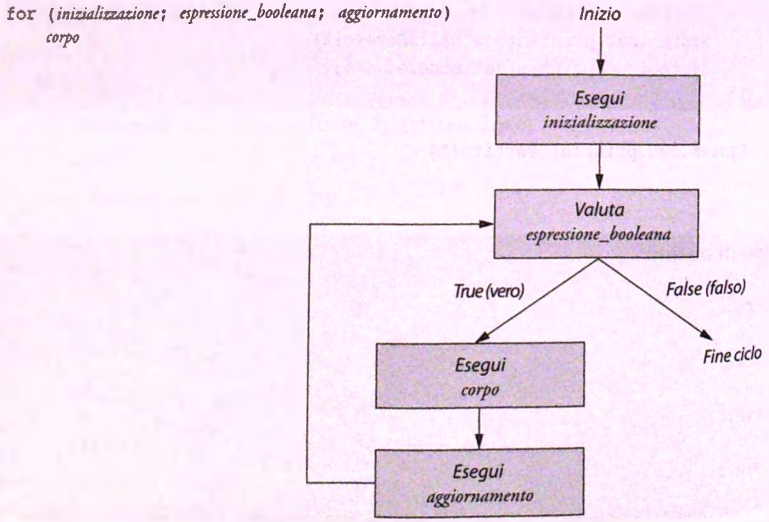


Figura 4.6 La semantica dell'istruzione `for`.

La sintassi dell'istruzione `for`

Sintassi

```
for (inizializzazione; espressione_booleana; aggiornamento)
    corpo
```

Il *corpo* del ciclo può essere un'istruzione singola oppure, come accade più spesso, un'istruzione composta che consiste di un elenco di istruzioni racchiuse tra parentesi graffe { }. Si noti che i tre elementi tra le parentesi tonde sono separati da due punti e virgola, non tre.

Esempio

```
for (prossimo = 0; prossimo <= 10; prossimo = prossimo + 2) {
    somma = somma + prossimo;
    System.out.println("La somma ora corrisponde a " + somma);
}
```

Punto e virgola aggiuntivo in un ciclo

Il codice seguente sembra corretto e viene compilato ed eseguito senza generare alcun errore. Tuttavia contiene un errore.

```
int prodotto = 1;
int numero;
for (numero = 1; numero <= 10; numero++);
    prodotto = prodotto * numero;
System.out.println("Il prodotto di tutti i numeri da 1 a 10 e' " +
                   prodotto);
```

L'output di questo codice (eseguito all'interno di un programma) è:

```
Il prodotto di tutti i numeri da 1 a 10 e' 11
```

Un risultato come questo potrebbe lasciare sconcertati. Chiaramente il problema sta nel ciclo `for`. Si presume che il ciclo `for` assegni alla variabile `prodotto` il valore ottenuto dalla moltiplicazione $1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 \times 8 \times 9 \times 10$, tuttavia assegna il valore 11.

Il problema è dovuto a un banale errore tipografico. L'istruzione `for` presenta un punto e virgola dopo la prima riga:

```
for (numero = 1; numero <= 10; numero++);
    prodotto = prodotto * numero;
```

Che cosa fa un'istruzione `for` di questo tipo? Il punto e virgola alla fine della riga del `for` è di troppo: in pratica segnala la fine di un'istruzione vuota, che però costituisce il corpo del ciclo `for`. Un punto e virgola da solo è considerato come un'istruzione che non fa nulla. È chiamata, infatti, **istruzione vuota** (*empty statement*) o **istruzione nulla** (*null statement*).

Questa istruzione `for` con il punto e virgola aggiuntivo è equivalente al seguente codice:

```
for (numero = 1; numero <= 10; numero++){
    //Non fare niente
}
```

Per questo motivo, il corpo dell'istruzione `for` viene effettivamente eseguito dieci volte; tuttavia, ogni volta che viene eseguito, il ciclo non fa niente, a parte incrementare la variabile `numero` di 1. Questo fa sì che la variabile `numero` valga 11 quando il programma raggiunge l'istruzione:

```
prodotto = prodotto * numero;
```

Si ricordi che la variabile `numero` è stata inizializzata a 1 e viene incrementata di 1 per dieci volte. Questo fa sì che il suo valore diventi 11. Con il valore della variabile `prodotto` uguale a 1 e il valore della variabile `numero` uguale a 11, l'istruzione precedente assegna alla variabile `prodotto` il risultato della moltiplicazione di 1 per 11, cioè 11. Per correggere il problema è sufficiente rimuovere il punto e virgola alla fine della riga dell'istruzione `for`.

Lo stesso problema si può presentare anche nel caso di un ciclo `while`. Il seguente ciclo `while` ha lo stesso difetto del ciclo `for` precedente, ma le conseguenze sono peggiori:

```
int prodotto = 1;
int numero = 1;
while (numero <= 10);{
    prodotto = prodotto * numero;
    numero ++;
}
System.out.println("Il prodotto di tutti i numeri da 1 a 10 e' " +
                   prodotto);
```

Il punto e virgola termina il ciclo `while` e, quindi, il corpo del ciclo `while` è l'istruzione vuota. Dato che il corpo del ciclo è l'istruzione vuota, a ogni iterazione del ciclo non accade niente. Per questo motivo il valore della variabile `numero` non cambia mai e quindi la condizione:

```
numero <= 10
```

è sempre vera. Questo è quindi un ciclo infinito, che non fa nulla e dura per sempre.

4.1.4 Dichiarare variabili all'interno di un'istruzione `for`

Nella parte di inizializzazione di un'istruzione `for` si può anche dichiarare una variabile come nell'esempio seguente:

```
int somma = 0;
for (int n = 1; n <= 10; n++)
    somma = somma + n * n;
```

In questo caso, la variabile n è **locale** (*local*) al ciclo `for`, ovvero non può essere usata al di fuori del ciclo. Per esempio, dopo la fine del ciclo, non si può chiedere di visualizzare il valore della variabile n usando un'istruzione `println`:

```
for (int n = 1; n <= 10; n++)
    somma = somma + n * n;
System.out.println(n); //Non valido
```

La porzione di programma in cui è disponibile una variabile è detto **visibilità** (*scope*) della variabile. Nell'esempio precedente la visibilità della variabile n è l'istruzione `for`, incluso il corpo. Pertanto n non avrà alcun significato al di fuori di questa istruzione `for`.

Quando una variabile viene usata solo per controllare un'istruzione `for`, è bene dichiararla all'interno dell'istruzione di inizializzazione del ciclo invece che prima del ciclo. Per esempio, se si fosse seguita questa indicazione nel box "Punto e virgola aggiuntivo in un ciclo" precedente, avremmo riscritto il codice nel seguente modo:

```
int prodotto = 1;
for (int numero = 1; numero <= 10; numero++);
    prodotto = prodotto * numero; //Non valido
```

Dal momento che `numero` in questo caso è locale rispetto al ciclo `for` e dato che il punto e virgola alla fine del ciclo termina il corpo del ciclo, la variabile `numero` non è visibile nelle righe che seguono la riga che inizia per `for`. L'istruzione così scritta avrebbe pertanto generato un errore di sintassi.

4.1.5 Usare una virgola in un'istruzione `for`

Un'istruzione `for` può anche eseguire più istruzioni di inizializzazione. Per usare più istruzioni di inizializzazione basta separare le istruzioni con una virgola, come nell'esempio seguente:

```
for (int numero = 1, int prodotto = 1; numero <= 10; numero++)
    prodotto = prodotto * numero;
```

Questo ciclo `for` inizializza la variabile `numero` con il valore 1 e in più inizializza la variabile `prodotto` con il valore 1. Si noti che in questo caso per separare due azioni di inizializzazione è stata utilizzata una virgola, non un punto e virgola. La virgola utilizzata in questo contesto viene detta **operatore virgola** (*comma operator*).

In maniera analoga si possono avere più azioni di aggiornamento, basta separarle con una virgola. Questa pratica potrebbe portare a cicli `for` che, sebbene abbiano un corpo vuoto, svolgono operazioni utili. L'esempio seguente mostra proprio come può essere riscritto l'esempio mostrato in precedenza:

```
for (int n = 1, int prodotto = 1; n <= 10; prodotto = prodotto * n, n++);
```

In questo caso, il corpo del ciclo è stato interamente integrato nell'istruzione di aggiornamento. Sebbene sia possibile definire iterazioni senza corpo e sebbene molti programmatori lo ritengano un indice di talento di programmazione, il fatto di definire iterazioni senza corpo riduce la leggibilità del codice; per questo motivo è una pratica sconsigliabile. Come già indicato nel box "Punto e virgola aggiuntivo in un ciclo" precedente, spesso un ciclo `for` senza corpo è il risultato di un errore di programmazione.

Nei cicli `for` non si possono indicare più espressioni booleane che ne determinino la terminazione; si possono però raggruppare più espressioni usando gli operatori `&&` o `||`, in modo da formare un'espressione booleana composta.

Scegliere l'istruzione iterativa più adatta

MyLab

Video 4.2
Scelta del ciclo

Si supponga che il programma che si sta scrivendo richieda un ciclo. Come decidere quale costrutto iterativo è più idoneo? Questo paragrafo fornisce alcune indicazioni utili per la scelta del ciclo più adatto. L'istruzione `while`, per esempio, *non* può essere utilizzata se il corpo del ciclo deve essere eseguito almeno una volta qualsiasi sia l'input del programma. Se si sa che il corpo del ciclo dovrebbe essere iterato almeno una volta, l'istruzione `do-while` risulta certamente la più adatta.

Tuttavia, più spesso di quanto si creda, si ha la necessità di realizzare cicli che potrebbero non dover eseguire il proprio corpo neanche una volta. In questi casi è necessario usare un costrutto `while` o `for`.

Se l'esecuzione del corpo del ciclo modifica una variabile numerica, per esempio un contatore, di una quantità costante a ogni iterazione, è bene usare l'istruzione `for`. Tuttavia nel caso in cui il ciclo `for` renda il codice poco chiaro, è meglio usare l'istruzione `while`.

L'istruzione `while` è sempre una scelta sicura, in quanto permette di realizzare qualsiasi tipo di iterazione. Tuttavia è bene usare anche gli altri costrutti, poiché in certi casi sono più adatti e rendono più chiaro il codice. Per esempio, molti programmatori usano il ciclo `for` per chiarire la logica del loro codice.

4.1.6 Istruzione `for-each`

Nel capitolo precedente si è visto che per restringere i valori possibili di una variabile a un insieme predefinito si usa un'enumerazione. Quando è necessario ripetere una certa azione per ciascun elemento di un'enumerazione, si può usare una qualsiasi delle istruzioni iterative presentate nei paragrafi precedenti. Tuttavia Java fornisce un'altra forma di istruzione `for` da utilizzare proprio quando si deve operare su tutti gli elementi di una collezione di dati, come un'enumerazione: l'**istruzione `for-each`** (letteralmente "per ciascuno").

L'implementazione di un programma per giocare a carte, per esempio, potrebbe richiedere la definizione di un'enumerazione per i quattro semi (cuori, quadri, picche e fiori) come segue:

```
enum Seme {CUORI, QUADRI, PICCHE, FIORI}
```

Per visualizzare questi semi sullo standard output, si può utilizzare un ciclo come quello che segue:

```
for (Seme semeSuccessivo : Seme.values())
    System.out.printl(semeSuccessivo + " ");
System.out.println();
```

L'espressione `Seme.values()` rappresenta tutti i valori dell'enumerazione. La variabile `semeSuccessivo` assume ognuno di questi valori a mano a mano che l'iterazione procede.

Per questo motivo l'output di questo ciclo sarà:

```
CUORI QUADRI PICCHE FIORI
```

Il Capitolo 12 mostrerà come usare l'istruzione `for-each` per collezioni di dati diverse dalle enumerazioni.

4.2 Programmare con i cicli

Un ciclo è costituito tipicamente da tre elementi: le istruzioni di inizializzazione che devono precedere l'iterazione, il corpo del ciclo e il meccanismo per la terminazione del ciclo. In questo paragrafo vengono presentate alcune tecniche utili per progettare correttamente ciascuno di questi componenti. Sebbene l'inizializzazione sia il primo elemento del ciclo, normalmente il *corpo* del ciclo è la prima parte ad essere progettata; per questo motivo, sarà anche il primo argomento trattato in questo paragrafo.

4.2.1 Il corpo del ciclo

Un primo modo per definire il corpo di un ciclo è quello di scrivere la sequenza di azioni che devono essere eseguite dal programma quando viene avviato. Per esempio, si potrebbe scrivere la seguente sequenza di azioni:

1. Mostra le istruzioni all'utente
2. Inizializza le variabili
3. Leggi un numero e assegnalo alla variabile `successivo`
4. `somma = somma + successivo`
5. Mostra `numero` e `somma`
6. Leggi un altro numero e assegnalo alla variabile `successivo`
7. `somma = somma + successivo`
8. Mostra `numero` e `somma`
9. Leggi un altro numero e assegnalo alla variabile `successivo`
10. `somma = somma + successivo`
11. Mostra `numero` e `somma`
12. Leggi un altro numero e assegnalo alla variabile `successivo`
13. ... e così via

Dopo aver scritto la sequenza di azioni, occorre individuare un insieme di azioni da ripetere più volte. In questo caso le azioni ripetute sono:

Leggi un altro numero e assegnalo alla variabile `successivo`

`somma = somma + successivo`

Mostra `numero` e `somma`

Il corpo del ciclo, espresso in pseudocodice, potrebbe essere costituito proprio da queste tre azioni. L'intero pseudocodice del programma potrebbe quindi diventare:

1. Mostra le istruzioni all'utente
2. Inizializza le variabili
3. Ripeti le seguenti operazioni per un numero opportuno di volte
   3.1 Leggi un numero e assegnalo alla variabile `successivo`
   3.1 `somma = somma + successivo`
   3.3 Mostra `numero` e `somma`

4.2.2 Istruzioni di inizializzazione

Si consideri lo pseudocodice definito nel paragrafo precedente. Come si può notare, questo prevede che ogni volta che viene eseguita l'istruzione:

```
somma = somma + successivo
```

all'interno del ciclo, la variabile `somma` contenga già un valore. In particolare, la variabile `somma` deve contenere un valore già dalla prima volta che il ciclo viene eseguito. Questo implica che `somma` sia già stata inizializzata, cioè che le sia già stato assegnato un valore prima che inizi il ciclo. Quando si cerca di individuare il valore corretto con cui inizializzare una variabile, occorre ragionare partendo da quello che si vuole che accada dopo la prima iterazione del ciclo. Nel caso in oggetto, dopo la prima iterazione si vuole che il valore di `somma` sia uguale al primo valore della variabile `successivo`. L'unico modo per far sì che `somma` sia uguale a `successivo` dopo la prima iterazione è quello di porre il valore della variabile `somma` uguale a 0. Perciò una delle operazioni di inizializzazione deve essere:

```
somma = 0
```

L'unica altra variabile utilizzata all'interno del ciclo è `successivo`. La prima istruzione eseguita che coinvolge la variabile `successivo` è:

Leggi un numero e assegnalo alla variabile `successivo`

Questa istruzione assegna un valore alla variabile `successivo`. Per questo motivo, non è necessario assegnarle un valore prima che l'iterazione venga eseguita. Di conseguenza, l'unica variabile che deve essere inizializzata è `somma`. Alla luce di queste considerazioni, lo pseudocodice potrebbe essere riscritto nel seguente modo:

1. Mostra le istruzioni all'utente
2. `somma = 0`
3. Ripeti le seguenti operazioni per un numero opportuno di volte
   3.1 Leggi un numero e assegnalo alla variabile `successivo`
   3.1 `somma = somma + successivo`
   3.1 Mostra `numero` e `somma`

Le variabili non vengono sempre inizializzate a 0 come si può vedere nell'esempio che segue. Si supponga che il ciclo computi il prodotto di n numeri come segue:

```
for (int contatore = 1; contatore <= n; contatore++) {
    Leggi un numero e assegnalo alla variabile successivo
```

```
}
```

Si supponga che, in questo caso, tutte le variabili siano di tipo `int`. Se si inizializzasse la variabile `prodotto` a `0`, indipendentemente da quanti numeri vengano letti e moltiplicati, il valore di prodotto resterebbe uguale a `0`. Il numero `0`, quindi, non è il valore corretto con cui inizializzare il valore della variabile `prodotto`. Il valore corretto è `1`. Per capire che `1` è il valore iniziale corretto occorre notare che la prima volta che il ciclo viene eseguito, è necessario che la variabile `prodotto` sia uguale al primo numero letto. E questo succede se si inizializza la variabile `prodotto` a `1`. Il ciclo e l'operazione di inizializzazione devono quindi essere scritti nel seguente modo:

```
int prodotto = 1;
for (int contatore = 1; contatore <= n; contatore++) {
    Leggi un numero e assegnalo alla variabile successivo
    prodotto = prodotto * successivo;
}
```

4.2.3 Controllare il numero di iterazioni in un ciclo

Questo paragrafo presenta alcune tecniche standard per determinare i criteri di terminazione di un ciclo. In alcuni casi è possibile specificare il numero esatto di iterazioni per cui il ciclo deve essere ripetuto. Si supponga, per esempio, di voler conoscere la media dei voti di un esame di un particolare corso. Per farlo, occorre conoscere il numero degli studenti che hanno sostenuto l'esame. Per questo motivo occorre leggere tale numero e assegnarlo alla variabile `numeroStudenti` che rappresenta proprio il numero di studenti. In questo caso si può usare un ciclo `for` per ripetere il corpo del ciclo un numero di volte pari al numero degli studenti, cioè al valore della variabile `numeroStudenti`. Il codice che segue rappresenta un esempio di questo comportamento:

```
double prossimo, media, somma = 0;
for (int contatore = 1; contatore <= numeroStudenti; contatore ++) {
    prossimo = tastiera.nextDouble();
    somma = somma + prossimo;
}
if (numeroStudenti > 0)
    media = somma / numeroStudenti;
else
    System.out.println("Nessun punteggio disponibile per " +
                       "calcolare la media.");
```

Si noti che il ciclo `for` controlla la ripetizione del corpo del ciclo usando la variabile `contatore` per contare da `1` a `numeroStudenti`. Cicli come questo, in cui si conosce esattamente il numero di iterazioni prima di eseguire il ciclo, sono chiamati **cicli count-controlled** (letteralmente "controllati da un contatore"). In questo particolare esempio, la variabile `contatore` non viene usata nel corpo del ciclo; tuttavia in altri casi potrebbe accadere che il contatore venga usato anche nel corpo del ciclo. I cicli *count-controlled* non devono essere necessariamente implementati con istruzioni `for`, anche se questo è il modo più semplice per farlo. Si noti che se nessuno studente ha sostenuto l'esame, il corpo del ciclo non viene mai eseguito, perciò l'istruzione `if-else` previene una divisione per `0`.

Purtroppo non è sempre possibile conoscere a priori il numero di iterazioni da compiere. Un modo semplice per capire quando terminare un'iterazione consiste nel chiedere all'utente se sia giunto il momento di farlo.

Questa tecnica è detta **ask before iterating** (letteralmente "chiedi prima di iterare"). Il codice seguente, per esempio, aiuta un cliente a determinare il costo dell'acquisto di varie quantità di prodotti:

```
do {
    System.out.println("Inserisci il prezzo:");
    prezzo = tastiera.nextDouble();
    System.out.print("Inserisci il numero di prodotti:");
    quantita = tastiera.nextInt();
    System.out.println(quantita + " prodotti a " + prezzo);
    System.out.println("Costo totale: " + prezzo * quantita);
    System.out.println("Vuoi fare un altro acquisto?");
    System.out.println("Scrivi si o no.");
    risposta = tastiera.next();
} while (risposta.equalsIgnoreCase("si"));
```

Nel caso in cui si sappia anticipatamente che ciascun utente effettua sempre almeno un acquisto, questa istruzione `do-while` sarebbe appropriata. In altri casi un'istruzione `while` potrebbe essere la soluzione migliore.

Questo codice si comporta in maniera adeguata se ciascun cliente effettua solo pochi acquisti. Tuttavia il programma potrebbe diventare tedioso qualora il numero totale di iterazioni non fosse ridotto. Per lunghi elenchi di valori di input è preferibile usare un **valore sentinella** (*sentinel value*) per indicare la fine dell'input. Un valore sentinella deve essere diverso da tutti i valori veri e propri che il programma utilizza per effettuare le computazioni richieste. Si supponga, per esempio, di voler individuare il punteggio più alto e quello più basso ottenuti in un esame. Sapendo in anticipo che almeno una persona ha svolto un esame e nessuno ha preso un voto negativo, si potrebbe chiedere all'utente di specificare un numero negativo qualsiasi dopo aver inserito l'ultimo valore. In questo caso il numero negativo sarebbe il valore sentinella e indicherebbe la fine dei dati di input.

Il codice per individuare il voto più alto e più basso potrebbe essere il seguente:

```
System.out.println("Scrivi i voti di tutti gli studenti.");
System.out.println("Scrivi un numero negativo dopo");
System.out.println("che hai scritto tutti i voti.");
Scanner tastiera = new Scanner(System.in);
double max = tastiera.nextDouble();
double min = max; //max e min coincidono con il primo valore.
double prossimo = tastiera.nextDouble();
while (prossimo >= 0) {
    if (prossimo > max)
        max = prossimo;
    else if (prossimo < min)
        min = prossimo;
    prossimo = tastiera.nextDouble();
}
System.out.println("Il voto piu' alto e' " + max);
System.out.println("Il voto piu' basso e' " + min);
```

Se l'utente scrivesse i voti:

```
100
90
10
-1
```

l'output sarebbe:

```
Il voto piu' alto e' 100
Il voto piu' basso e' 10
```

Il valore sentinella, in questo caso `-1`, non viene usato per la computazione del voto, il che implica, giustamente, che il punteggio più basso sia `10` e non `-1`. Il valore `-1` segnala solo la fine dell'elenco di numeri.

Le tre tecniche presentate nei paragrafi precedenti (usare un contatore, chiedere all'utente o scegliere un valore sentinella) permettono di coprire la maggior parte dei casi che si possono incontrare. Il prossimo caso di studio utilizza un valore sentinella e sfrutta una variabile booleana per terminare l'iterazione del ciclo.

CASO DI STUDIO
USARE UNA VARIABILE BOOLEANA PER TERMINARE UN CICLO

Questo caso di studio non riguarda la risoluzione completa di un problema, ma la progettazione di un ciclo utile in vari contesti. Questo caso di studio permette di acquisire familiarità con uno dei modi più comuni per usare le variabili booleane.

Il ciclo in questo esempio legge una serie di numeri e ne calcola la somma. I numeri letti sono tutti positivi. Questi numeri potrebbero per esempio rappresentare il numero di ore di lavoro svolte da tutte le persone di un gruppo di lavoro. Dato che nessuno può lavorare un numero negativo di ore, si può presupporre che tutti i numeri non siano negativi. Proprio per questo motivo si può usare un numero negativo come valore sentinella per indicare la fine dell'elenco. Per il caso specifico affrontato in questo esempio, si presuppone che siano tutti numeri interi, ma lo stesso approccio può essere adottato per altri tipi di numeri o anche per valori non numerici.

Il seguente pseudocodice permette di comprendere meglio il problema e la possibile soluzione.

```
int somma = 0;
Per ciascun numero nella lista si facciano le seguenti operazioni:
   if (numero negativo)
      Questa è l'ultima iterazione.
   else
      somma = somma + il numero corrente
```

Dato che si sa che un numero negativo indica la fine dell'elenco, lo pseudocodice può essere modificato come segue:

```
int prossimo;
int somma = 0;
while (ci sono altri numeri da leggere) {
   prossimo = tastiera.nextInt();
```

```
    if (prossimo < 0)
        Questa è l'ultima iterazione.
    else
        somma = somma + prossimo
}
```

Questo pseudocodice può essere trascritto in codice Java in vari modi. Un primo modo consiste nell'utilizzare una variabile booleana. L'aspetto positivo legato all'uso di una variabile booleana è che rende il codice simile a una frase in lingua italiana. La variabile booleana `ciSonoAltriNumeriDaLeggere` si può usare come condizione del ciclo `while` al posto della frase "ci sono altri numeri da leggere". Questo permette di ottenere il seguente codice:

```
int prossimo;
int somma = 0;
while (ciSonoaltriNumeriDaLeggere) {
    prossimo = tastiera.nextInt();
    if (prossimo < 0)
        Questa è l'ultima iterazione.
    else
        somma = somma + prossimo
}
```

Completare la conversione dello pseudocodice in codice Java è semplice. La frase "Questa è l'ultima iterazione." può essere trascritta osservando che il ciclo termina quando la variabile booleana `ciSonoAltriNumeriDaLeggere` è uguale a `false`. Perciò "Questa è l'ultima iterazione." può essere tradotta come:

```
ciSonoAltriNumeriDaLeggere = false;
```

Ciò che rimane da fare è determinare il valore iniziale per la variabile `ciSonoAltriNumeriDaLeggere`. Dal momento che, anche se l'elenco di numeri da leggere è vuoto, occorre leggere almeno il valore sentinella, il corpo del ciclo deve essere eseguito almeno una volta. Per eseguire il ciclo almeno una volta, la variabile `ciSonoAltriNumeriDaLeggere` deve essere vera; questo vuol dire che la variabile `ciSonoAltriNumeriDaLeggere` deve essere inizializzata a `true`. Perciò il codice risultante è il seguente:

```
int prossimo;
int somma = 0;
boolean ciSonoAltriNumeriDaLeggere = true;
while (ciSonoAltriNumeriDaLeggere) {
    prossimo = tastiera.nextInt();
    if (prossimo < 0)
        ciSonoAltriNumeriDaLeggere = false;
    else
        somma = somma + prossimo;
}
```

Quando il ciclo termina, la variabile `somma` contiene la somma di numeri dati in input, escluso il valore sentinella. Questo ciclo può essere utilizzato in un programma. Poiché il nome `ciSonoAltriNumeriDaLeggere` è piuttosto lungo e scomodo, è opportuno abbreviarlo con `altri`. Il Listato 4.6 mostra un programma che utilizza il ciclo definito in questo paragrafo.

LISTATO 4.6 Usare una variabile booleana per terminare un ciclo.

MyLab

```
import java.util.Scanner;

/**
Mostra l'uso di una variabile booleana per terminare un ciclo.
*/
public class BooleanDemo {

   public static void main(String[] args) {

      System.out.println("Inserisci dei numeri non negativi.");
      System.out.println("Scrivi un numero negativo");
      System.out.println("per indicare la fine dell'elenco.");

      int somma = 0;
      boolean altri = true;
      Scanner tastiera = new Scanner(System.in);
      while (altri) {
         int prossimo = tastiera.nextInt();
         if (prossimo < 0)
            altri = false;
         else
            somma = somma + prossimo;
      }
      System.out.println("La somma dei numeri e' " + somma);
   }
}
```

Esempio di output

```
Inserisci dei numeri non negativi.
Scrivi un numero negativo
per indicare la fine dell'elenco.
1 2 3 -1
La somma dei numeri e' 6
```

ESEMPIO DI PROGRAMMAZIONE
SPESE FOLLI

Si immagini che Paola abbia vinto un buono omaggio da 100 Euro a una gara. Il buono deve essere speso in un certo negozio, ma non si possono comprare più di tre prodotti. Il computer del negozio tiene traccia di quanto rimane da spendere e del numero di prodotti acquistati. Ogni volta che Paola sceglie un prodotto, il computer dice se può essere acquistato. Sebbene in questo esempio si considerino cifre piccole, si vuole scrivere un programma in cui sia il numero di Euro a disposizione, sia il numero di prodotti acquistabili possano essere modificati facilmente.

Chiaramente questo è un processo ripetitivo: Paola continua ad acquistare finché ha a disposizione ancora soldi e finché ha acquistato meno di tre prodotti. L'espressione

di controllo del ciclo deve basarsi su questi due criteri, come mostrato nello pseudocodice seguente:

Algoritmo per il programma del negozio

```
1. rimastiDaSpendere = ammontare del buono omaggio
2. totaleSpeso = 0
3. numeroProdotti = 1
4. while (ci sono ancora soldi da spendere e
          (numeroProdotti <= numero massimo di prodotti)) {
       Mostra i soldi rimanenti e il numero di oggetti che possono essere acquistati
       Leggi il prezzo dell'acquisto proposto
       if (possiamo affrontare l'acquisto) {
          Mostra un messaggio
          totaleSpeso = totaleSpeso + prezzo del prodotto
          Aggiorna rimastiDaSpendere
          if (rimastiDaSpendere > 0) {
             Mostra l'ammontare rimanente
             numeroProdotti++;
          } else {
             Mostra il messaggio "Hai finito i soldi."
             Questa è l'ultima iterazione del ciclo
          }
       } else {
          Mostra un messaggio "Prodotto troppo costoso."
       }
   }
   Mostra la somma spesa e un messaggio di saluto
```

Questo esempio si focalizza sull'implementazione del criterio per terminare il ciclo. Così come nel caso di studio precedente, occorre utilizzare una variabile booleana per indicare se restano ancora soldi da spendere. A questa variabile può essere assegnato il nome `possiediSoldi`. Prima del ciclo, questa variabile deve avere valore `true`, mentre per uscire dal ciclo il valore deve diventare `false`. Il Listato 4.7 mostra il programma completo e un output di esempio. Si noti l'utilizzo di costanti per indicare l'ammontare del buono omaggio e del numero massimo di prodotti che si possono acquistare. Per semplicità si supponga che l'ammontare di soldi sia sempre esprimibile con numeri interi. Il paragrafo successivo mostra invece come utilizzare l'istruzione `break` per terminare il ciclo. Si noti che a differenza delle variabili booleane, l'utilizzo dell'istruzione `break` può generare soluzioni meno chiare.

yLab

LISTATO 4.7 Programma per le spese folli.

```
import java.util.Scanner;

public class SpeseFolli {

   public static final int AMMONTARE_BUONO = 100;
   public static final int PRODOTTI_MAX = 3;
```

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

    boolean possiediSoldi = true;
    int rimastiDaSpendere = AMMONTARE_BUONO;
    int totaleSpeso = 0;
    int numeroProdotti = 1;
    Scanner tastiera = new Scanner(System.in);

    while (possiediSoldi && (numeroProdotti <= PRODOTTI_MAX)) {
        System.out.println("Puoi comprare fino a " +
                           (PRODOTTI_MAX - numeroProdotti + 1) +
                           " prodotti");
        System.out.println("che costano meno di " +
                           rimastiDaSpendere + " Euro.");
        System.out.print("Inserisci il prezzo del prodotto # " +
                         numeroProdotti + ": ");
        int costoProdotto = tastiera.nextInt();
        if (costoProdotto <= rimastiDaSpendere) {
            System.out.println("Puoi comprare questo prodotto.");
            totaleSpeso = totaleSpeso + costoProdotto;
            System.out.println("Fino a questo momento hai speso " +
                               totaleSpeso + " Euro.");
            rimastiDaSpendere = AMMONTARE_BUONO - totaleSpeso;
            if (rimastiDaSpendere > 0){
                numeroProdotti++;
            } else {
                System.out.println("Hai finito i soldi.");
                possiediSoldi = false;
            }
        } else {
            System.out.println("Prodotto troppo costoso.");
        }
    }
    System.out.println("Hai speso " + totaleSpeso +
                       " e hai finito le spese folli!");
  }
}
```

Esempio di output

```
Puoi comprare fino a 3 prodotti
che costano meno di 100 Euro.
Inserisci il costo dell'articolo # 1: 80
Puoi comprare questo prodotto.
Fino a questo momento hai speso 80 Euro.
Puoi comprare fino a 2 prodotti
che costano meno di 20 Euro.
Inserisci il prezzo del prodotto # 2: 20
Puoi comprare questo prodotto.
Fino a questo momento hai speso 100 Euro.
Hai finito i soldi.
Hai speso 100 e hai finito le spese folli!
```

4.2.4 Istruzioni `break` e `continue` nei cicli (opzionale)

Negli esempi presentati nei paragrafi precedenti, i cicli `while`, `do-while` e `for` terminano quando l'espressione booleana di controllo assume il valore `false`. Nel programma presentato nel Listato 4.7, l'espressione di controllo coinvolge la variabile booleana `possiediSoldi`: quando questa variabile diventa `false`, l'espressione di controllo assume il valore `false` e il ciclo termina.

I cicli possono terminare anche quando incontrano un'istruzione `break`. Quando viene eseguita un'istruzione `break`, il ciclo che la contiene termina immediatamente, senza eseguire l'eventuale parte rimanente del corpo del ciclo. Java permette di utilizzare l'istruzione `break` con i cicli `while`, `do-while` e `for`. Questa istruzione corrisponde all'istruzione `break` usata nel costrutto `switch`.

La variabile booleana `possiediSoldi` compare per tre volte nel programma nel Listato 4.7. Se rimuoviamo l'uso di questa variabile nelle prime due posizioni in cui compare e se sostituiamo l'istruzione:

```
possiediSoldi = false;
```

con

```
break;
```

il ciclo risultante avrebbe l'aspetto rappresentato nel Listato 4.8. Quando la variabile `possiediSoldi` non è più positiva, viene eseguito il metodo `println`, seguito dall'istruzione `break`, che termina l'iterazione. L'istruzione che verrà eseguita successivamente sarà quindi l'istruzione che segue il corpo del ciclo `while`.

Se il ciclo che contiene l'istruzione `break` è, a sua volta, contenuto in un ciclo più esterno, l'istruzione `break` esce solo dal ciclo più interno. Analogamente, se l'istruzione `break` è in un'istruzione `switch` contenuta all'interno del corpo di un ciclo, l'istruzione `break` esce solo dall'istruzione `switch`, ma non dal ciclo. L'istruzione `break` termina solamente l'istruzione iterativa o l'istruzione `switch` *più interna* che la contiene.

Un ciclo senza un'istruzione `break` presenta una struttura semplice e facile da comprendere: per decidere se terminare il ciclo viene verificata un'espressione booleana. Aggiungendo un'istruzione `break`, il ciclo può terminare per due motivi: perché l'espressione booleana di controllo ha assunto il valore `false` o perché è stata eseguita l'istruzione `break`. L'utilizzo di un'istruzione `break` può pertanto complicare l'interpretazione del ciclo.

Per esempio, il ciclo presente nel Listato 4.8 inizia con:

```
while (numeroProdotti <= PRODOTTI_MAX)
```

Questo ciclo sembra avere solamente una condizione che ne determina la terminazione. Per scoprire che esiste anche un'altra condizione di terminazione, è necessario ispezionare il corpo del ciclo, individuando così l'istruzione `break`. Al contrario, il ciclo nel Listato 4.7 inizia con:

```
while (possiediSoldi && (numeroProdotti <= PRODOTTI_MAX))
```

Questa istruzione permette di capire velocemente che il ciclo termina quando finiscono i soldi o quando si raggiunge il numero massimo di prodotti acquistabili.

```
hile (numeroProdotti <= PRODOTTI_MAX) {
   . . .
   if (costoProdotto <= rimastiDaSpendere) {
   . . .
      if (rimastiDaSpendere > 0){
         numeroProdotti++;
      } else {
         System.out.println("Hai finito i soldi.");
         break;
      }
   } else {
   . . .
   }
}
System.out.println(. . .)
```

Evitare le istruzioni `break` e `continue` nei cicli

Le istruzioni `break` e `continue` all'interno dei cicli dovrebbero essere evitate a causa delle complicazioni che generano. Un ciclo che include una di queste istruzioni può, infatti, essere riscritto in modo da non farne uso.

4.2.5 Cicli difettosi

I programmi che contengono cicli sono soggetti a difetti con maggiore probabilità dei programmi più semplici, trattati nei capitoli precedenti. Fortunatamente gli errori tipici dei cicli sono riconducibili a poche tipologie, elencate di seguito. Questo paragrafo presenta inoltre alcune tecniche standard utili per individuare e correggere i difetti delle istruzioni iterative.

I due errori più comuni che affliggono i cicli sono i seguenti.

- Cicli infiniti indesiderati.
- Errori di una unità.

I cicli infiniti sono già stati introdotti in precedenza, tuttavia occorre enfatizzarne un aspetto critico: un ciclo potrebbe, in effetti, terminare per alcuni valori, mentre per altri potrebbe trasformarsi in un ciclo infinito. Si consideri, per esempio, il caso di un conto in banca in rosso. La banca applica una penalità per ogni mese in cui il saldo è negativo. Si

supponga di voler generare un programma che permetta di capire quanto tempo occorre per riportare il conto in attivo qualora venga depositata una somma fissa ogni mese. Si consideri il seguente codice:

```
conteggio = 0;
while (saldo < 0) {
    saldo = saldo – penalita;
    saldo = saldo + deposito;
    conteggio++;
}
System.out.println("Otterrai un saldo non negativo in " +
                   conteggio + " mesi.");
```

Si supponga di inserire questo frammento di codice in un programma completo e di collaudarlo usando dati concreti, per esempio 15 Euro per la `penalita` e 50 Euro per l'ammontare del `deposito`. Questi dati fanno sì che il programma sia eseguito correttamente. Tuttavia, può accadere che questo stesso programma, fatto girare con dati concreti diversi, finisca in un ciclo infinito. Questo potrebbe succedere, per esempio, se l'ammontare del `deposito` è troppo contenuto, per esempio di soli 10 Euro al mese. Dato che, in caso di saldo negativo, la banca prevede una penalità di 15 Euro al mese, il conto diventerà ogni mese più scoperto, anche se l'utente continua a depositare denaro.

Sebbene questa situazione possa sembrare paradossale, non è tanto lontana dalla realtà, proprio perché il comportamento delle persone è a volte imprevedibile. Proprio per questo motivo, quando si programma, occorre fare attenzione a ogni minimo dettaglio. Un modo per correggere il programma precedente consiste nel modificare il codice in modo che determini in anticipo se l'iterazione diventa infinita. Per esempio si potrebbe modificare il codice nel modo seguente:

```
if (deposito <= penalita) {
    System.out.println("Deposito insufficiente");
} else {
    conteggio = 0;
    while (saldo < 0) {
        saldo = saldo – penalita;
        saldo = saldo + deposito;
        conteggio++;
    }
    System.out.println("Otterrai un saldo non negativo in " +
                       conteggio + " mesi.");
}
```

Un altro errore comune nei cicli è l'**errore di una unità** (*off-by-one*). Questo errore fa sì che il ciclo ripeta il corpo una volta di troppo o una volta di meno. Gli errori di questo tipo spesso derivano da una definizione non corretta dell'espressione booleana di controllo. Se, per esempio, nell'espressione booleana di controllo si usasse l'operatore < invece dell'operatore <=, il ciclo potrebbe ripetere l'iterazione per un numero di volte non corretto.

Un altro problema comune delle espressioni booleane di controllo è legato all'uso dell'operatore == per controllare l'uguaglianza. L'operatore di uguaglianza, ==, opera correttamente nel caso di numeri interi e caratteri, ma non è affidabile nel caso di numeri in virgola mobile, che corrispondono a quantità approssimate. L'operatore ==, infatti, con-

trolla se due valori sono uguali, perciò genera un risultato non prevedibile per i numeri in virgola mobile. Quando si confrontano numeri in virgola mobile è bene usare espressioni che impiegano gli operatori minore o maggiore, per esempio l'operatore <=. Usare gli operatori == o != per confrontare due numeri in virgola mobile può generare un errore di una unità, un ciclo infinito e anche altri tipi di errori.

Gli errori di una unità possono essere difficili da individuare, poiché, a prima vista, il risultato potrebbe sembrare plausibile. Tuttavia, un risultato errato potrebbe creare altri problemi. Occorre sempre fare un controllo specifico degli errori di una unità usando valori di input per i quali già si conosce l'output previsto.

Ripetere sempre i test (collaudi)

Ogni volta che si trova un difetto in un programma e lo si corregge (*fix*), è opportuno ripetere sempre l'esecuzione del programma, in quanto si potrebbe manifestare un altro difetto oppure la correzione stessa potrebbe aver introdotto nuovi errori.

4.2.6 Tracciare le variabili

Se un programma non si comporta come si vorrebbe, ma non si riesce a capire che cosa ci sia di sbagliato al suo interno, il tracciamento di alcune variabili chiave potrebbe aiutare a risolvere il problema.

Tracciare le variabili vuol dire osservare il valore che assumono le variabili mentre il programma è in esecuzione. Un programma non mostra il valore di una variabile ogni volta che viene modificato, ma osservare questo cambiamento durante l'esecuzione del programma può essere molto utile per il debugging del programma stesso.

Diversi sistemi di sviluppo presentano strumenti che permettono di tracciare agevolmente le variabili del programma senza intervenire su di esso. Questi strumenti variano da sistema a sistema. Se si possiedono questi strumenti, è bene imparare a usarli. In caso contrario, si possono tracciare le variabili semplicemente inserendo nel programma alcune istruzioni `println` temporanee.

Si supponga, per esempio, di voler tracciare le variabili del codice seguente che contiene un errore:

```
conteggio = 0;
while (saldo < 0) {
    saldo = saldo + penalita;
    saldo = saldo - deposito;
    conteggio++;
}
System.out.println("Otterrai un saldo non negativo in " +
                   conteggio + " mesi.");
```

Per tracciare le variabili si possono introdurre alcune istruzioni `println` nel seguente modo:

```
conteggio = 0;
System.out.println("conteggio == " + conteggio);          //*
System.out.println("saldo == " + saldo);                  //*
System.out.println("penalita == " + penalita);            //*
```

```
System.out.println("deposito == " + deposito);                    //*
while (saldo < 0) {
    saldo = saldo + penalita;
    System.out.println("saldo + penalita == " + saldo);          //*
    saldo = saldo - deposito;
    System.out.println("saldo - deposito == " + saldo);          //*
    conteggio++;
    System.out.println("conteggio == " + conteggio);             //*
}
System.out.println("Otterrai un saldo non negativo in " +
                   conteggio + " mesi.");
```

Lab

o 4.3
ugging
cicli

Dopo aver scoperto l'errore e aver corretto il codice si possono rimuovere le istruzioni di tracciamento. Etichettare queste istruzioni con un commento, come è stato fatto in questo esempio, ne facilita l'individuazione. Inserire le istruzioni di tracciamento sembra un'attività noiosa, tuttavia non comporta molto lavoro. Se si preferisce, si possono tracciare solo alcune delle variabili e vedere se queste forniscono informazioni sufficienti per identificare il problema. Tuttavia, di solito si fa più in fretta a tracciare tutte le variabili fin dall'inizio.

Usare un'etichetta di debug quando si tracciano le variabili

Alle volte, durante il debugging di un programma, si vuole evitare temporaneamente di eseguire le istruzioni di tracciamento del valore delle variabili. Questo può essere fatto definendo una costante booleana che può chiamarsi `DEBUG`, come nell'esempio che segue:

```
public static final boolean DEBUG = true;
. . .
if (DEBUG) {
    <Istruzioni che mostrano il valore di certe variabili>
}
```

In questo esempio, le istruzioni di debug inserite nell'istruzione `if` vengono eseguite. Per evitare di eseguirle in futuro, basta assegnare il valore `false` alla variabile `DEBUG`.

4.2.7 Controllo delle asserzioni

Un'**asserzione** è un'istruzione che specifica un'ipotesi sullo stato del programma. Un'asserzione può essere vera o falsa, ma deve risultare vera se non ci sono errori nel programma. Per esempio, tutti i commenti presenti nel codice seguente sono asserzioni:

```
//n == 1
while (n < limite) {
    n = 2 * n;
}
//n >= limite
//n e' la piu piccola potenza di 2 >= limite
```

Si noti che ognuna di queste asserzioni può essere vera o falsa, a seconda dei valori di `n` e `limite`; tuttavia, se il programma sta operando correttamente devono essere tutte vere. In pratica un'asserzione "asserisce" che un'affermazione sul programma è vera in quel preciso punto dell'esecuzione. Anche se questo esempio riguarda un ciclo, si possono scrivere asserzioni anche per altre situazioni.

In Java si può verificare in modo automatico se un'asserzione è vera o falsa e terminare il programma con un messaggio d'errore qualora questa non sia vera. In Java un **controllo di asserzione** (*assertion check*) ha il seguente aspetto:

```
assert espressione_booleana;
```

Se si esegue il programma in un certo modo e se *espressione_booleana* è falsa, il programma termina dopo aver mostrato un messaggio di errore che indica che l'asserzione è fallita. Se invece *espressione_booleana* è vera, non succede niente di particolare e l'esecuzione procede normalmente.

Il codice precedente, per esempio, può essere scritto come segue, dove due commenti sono sostituiti da altrettanti controlli di asserzione:

```
assert n == 1;
while (n < limite) {
    n = 2 * n;
}
assert n >= limite;
//n e' la piu piccola potenza di 2 >= limite
```

Si noti che solo due dei commenti sono stati tradotti in controlli di asserzione. Non tutti i commenti, infatti, possono diventare controlli di asserzione. Per esempio, l'ultimo commento è in effetti un'asserzione, che può essere vera o falsa; tuttavia non esiste un modo semplice per convertire questo commento in un'espressione booleana. Farlo non è impossibile, tuttavia costringerebbe a scrivere codice molto più complicato di quello che si intende controllare. La scelta su quali commenti trasformare in un'asserzione dipende quindi dal caso specifico.

Il controllo delle asserzioni può essere attivato e disabilitato a piacere. Per esempio può essere attivato durante il debugging dell'applicazione, in modo che un'asserzione falsa faccia terminare l'applicazione. Una volta terminato il processo di debugging si può però disattivare il controllo delle asserzioni, facendo sì che il codice del programma venga eseguito in maniera più efficiente. In questo modo, si possono anche lasciare le asserzioni nel codice, ignorandole in modo automatico una volta che il programma è stato controllato e corretto.

Il comando comunemente utilizzato per eseguire i programmi lascia disattivato il controllo delle asserzioni. Per eseguire il programma attivando il controllo delle asserzioni occorre usare il seguente comando:

```
java –enableassertions Programma
```

Qualora si utilizzi un ambiente di sviluppo integrato (IDE), si faccia riferimento alla sua documentazione per scoprire in quale modo è possibile gestire il controllo delle asserzioni.

Controllo delle asserzioni

Sintassi

```
assert espressione_booleana;
```

Le asserzioni possono essere poste in qualsiasi porzione di codice. Se il controllo delle asserzioni è attivato e l'espressione booleana *espressione_booleana* è falsa, il programma mostra un messaggio d'errore opportuno e termina l'esecuzione. Se il controllo delle asserzioni non è attivato, l'asserzione viene trattata come un commento.

Esempio

```
assert n >= limite;
```

4.3 Riepilogo

- Un ciclo è un costrutto che ripete un'azione per un certo numero di volte. La parte ripetuta è detta corpo del ciclo. Ogni ripetizione è detta iterazione del ciclo.
- Java offre tre costrutti iterativi: `while`, `do-while` e `for`.
- Sia l'istruzione `while`, sia l'istruzione `do-while` ripetono il corpo del ciclo finché un'espressione booleana risulta vera. L'istruzione `do-while` esegue il corpo del ciclo almeno una volta, mentre l'istruzione `while` potrebbe non eseguirlo mai.
- La logica di un'istruzione `for` è identica a quella di un'istruzione `while`. Le operazioni di inizializzazione, verifica e aggiornamento sono però raggruppate insieme e non disperse all'interno del ciclo. L'istruzione `for` è utile nel caso di cicli controllati con un contatore.
- L'istruzione `for-each` è una variante dell'istruzione `for` che svolge l'iterazione sugli elementi di una collezione di valori, per esempio un'enumerazione.
- Un modo per terminare un ciclo che richiede un input è quello di impiegare un valore sentinella alla fine dei dati e far sì che il ciclo controlli questo valore sentinella.
- Per controllare un ciclo può essere utilizzata una variabile di tipo `boolean`.
- I difetti più comuni dei cicli sono le iterazioni infinite e gli errori di una unità.
- L'espressione "tracciare una variabile" indica l'attività di mostrare il valore di una variabile in punti ben precisi del programma. L'operazione di tracciamento può essere effettuata utilizzando alcuni strumenti di debugging specifici o inserendo istruzioni di output temporanee.
- Un'asserzione è un'istruzione che indica, in un certo punto del programma, una proprietà che deve essere vera se il programma è corretto. Java fornisce un meccanismo di controllo delle asserzioni per verificare se un'asserzione sia realmente vera.

4.4 Esercizi

1. Si scriva un frammento di codice che legge le parole digitate sulla tastiera finché non viene digitata la parola fine. Per ciascuna parola (tranne la parola fine) si riporti se il primo carattere è uguale all'ultimo. Per implementare il ciclo richiesto si utilizzino i seguenti costrutti:
   a. l'istruzione `while`;
   b. l'istruzione `do-while`.
2. Si scriva un algoritmo che calcoli mese per mese il saldo del conto corrente. Si supponga di poter eseguire una transazione al mese, deposito o prelievo. Gli interessi vengono accreditati sul conto all'inizio di ogni mese. Il tasso di interesse mensile corrisponde a quello annuo diviso per 12.
3. Si scriva un algoritmo per un semplice gioco che chieda di indovinare un codice numerico di cinque cifre. Quando l'utente scrive la risposta, il programma restituisce due valori: il numero di cifre al posto giusto e la somma di queste cifre. Per esempio, se il codice segreto è 53840 e l'utente ipotizza 83241, le cifre 3 e 4 sono al posto giusto. Il programma perciò restituirebbe in output i numeri 2 (cifre corrette) e 7 (somma). Si permetta all'utente di provare per un numero prefissato di volte.
4. Si scriva un frammento di codice che computi la somma dei primi n numeri interi positivi dispari. Per esempio, se n fosse uguale a 9, il programma dovrebbe calcolare $1 + 3 + 5 + 7 + 9$.
5. Si modifichi il seguente codice in modo che utilizzi dei costrutti `while` annidati al posto dei `for`.

```
int s = 0;
int t = 1;
for (int i = 0; i < 10; i++) {
    s = s + i;
    for (int j = i; j > 0; j--) {
        t = t * (j - i);
    }
    s = s * t;
    s = System.out.println("T vale " + t);
}
System.out.println("S vale " + s);
```

6. Si scriva un'istruzione `for` che calcoli la somma $1 + 2^2 + 3^2 + 4^2 + 5^2 + \ldots + n^2$
7. (*Opzionale*) Si svolga l'esercizio precedente usando l'operatore virgola e omettendo il corpo del ciclo `for`.
8. Si scriva un ciclo che conti il numero di caratteri di spaziatura in una stringa data.

9. Si scriva un ciclo che crei una nuova stringa che corrisponde all'inversione di una stringa data.

10. Si scriva un programma che computi la statistica per otto lanci di una moneta. Per ciascuno dei lanci effettuati, l'utente scrive `t` se è uscito testa e `c` se è uscito croce. Il programma mostrerà il numero totale e la percentuale di teste e croci. Si usi l'operatore di incremento per contare ciascuna `t` e `c` inserita. Un possibile dialogo tra il programma e l'utente potrebbe essere il seguente:

```
Per ciascun lancio della moneta inserisci t se è uscito testa
e c se e' uscito croce.
Primo lancio: t
Secondo lancio: c
Terzo lancio: t
Quarto lancio: t
Quinto lancio: c
Sesto lancio: c
Settimo lancio: t
Ottavo lancio: t
Numero di teste: 5
Numero di croci: 3
Percentuale di teste: 62.5
Percentuale di croci: 37.5
```

11. Si supponga di partecipare a una festa, per socializzare si stringono le mani a tutti i partecipanti. Si scriva un frammento di codice usando un'istruzione `for` che calcoli il numero totale di strette di mano. *Suggerimento*: quando una persona arriva, stringe la mano a chi è già presente. Si usi il ciclo per individuare il numero di strette di mano effettuate ogni volta che arriva una nuova persona.

12. Si definisca un'enumerazione che includa i mesi dell'anno. Si usi un'istruzione `for-each` per mostrare ciascun mese.

13. Si scriva un frammento di codice che calcoli il punteggio finale di una partita a baseball. Si usi un ciclo per leggere il numero di punti effettuati da ciascuna delle due squadre durante ciascuno dei 9 tempi. Si mostri il punteggio finale.

14. Si supponga di lavorare per un'azienda che produce bibite. L'azienda vuole conoscere il costo ottimale per un contenitore cilindrico che contiene un certo volume di bibita. Si scriva un frammento di codice che utilizza un ciclo del tipo *chiedi-prima-di-iterare*. Durante ogni iterazione del ciclo, il codice deve chiedere all'utente di inserire il volume e il raggio del cilindro; quindi deve calcolare e visualizzare l'altezza e il prezzo del contenitore. Si utilizzi la formula seguente, in cui V rappresenta il volume, r il raggio, h l'altezza e C il prezzo.

$$h = V / (\pi r^2)$$
$$C = 2 \pi r(r + h)$$

15. Si supponga di voler computare la media geometrica di un elenco di valori positivi. Per calcolare la media geometrica di k valori, occorre moltiplicarli tra loro e quindi calcolare la radice *k-esima* del valore. Per esempio, la media geometrica di 2, 5 e 7 è la radice cubica del prodotto dei tre valori. Si usi un ciclo con una sentinella per permettere all'utente di inserire un numero arbitrario di valori. Si calcoli e si mostri

la media geometrica di tutti i valori esclusa la sentinella. *Suggerimento*: il metodo `Math.pow(x, 1.0/k)` calcola la `k`-*esima* radice di `x`.

16. Si immagini un programma che comprime i file all'80% e li salva su un supporto di memorizzazione. Prima che il file compresso venga memorizzato, deve essere diviso in blocchi da 512 byte ciascuno. Si sviluppi un algoritmo per questo programma che prima legge il numero di blocchi disponibili sul supporto di memorizzazione, quindi legge, in un ciclo, la dimensione non compressa del file, per determinare se il file compresso può essere inserito nello spazio rimanente nel supporto di memorizzazione. In caso affermativo, il programma comprime e memorizza il file. Il processo continua finché non incontra un file che supera lo spazio disponibile.

    Si supponga, per esempio, che il supporto possa contenere 1000 blocchi. Un file di dimensione 1100 byte viene compresso in un file di 880 byte e richiede 2 blocchi. Lo spazio disponibile è di 998 blocchi. Un file di 20.000 byte viene compresso in un file di 16.000 byte e richiede 32 blocchi. Lo spazio disponibile ora è di 966 blocchi.

17. Che cosa visualizza il seguente frammento di codice? Quali erano le azioni che il programmatore aveva in mente di codificare. Come si dovrebbero realizzare?

```
int prodotto = 1;
int max = 20;
for (int i = 0; i <= max; i++)
   prodotto = prodotto * i;
System.out.println("Il prodotto e' " + prodotto);
```

18. Che cosa viene visualizzato dalla seguente porzione di codice? Quali erano le azioni che il programmatore aveva in mente di codificare. Come si dovrebbero realizzare?

```
int somma = 0;
int prodotto = 1;
int max = 20;
for (int i = 1; i <= max; i++)
   somma = somma + i;
   prodotto = prodotto * i;
System.out.println("La somma e' " + somma + " e il prodotto e' " +
                   prodotto);
```

4.5 Progetti

1. Si ripeta il Progetto 4 del Capitolo 3, ma usando un ciclo che legge ed elabora le frasi finché l'utente non chiede di terminare il programma.
2. Si scriva un programma che implementi l'algoritmo dell'Esercizio 2.
3. Si ripeta il Progetto 5 del Capitolo 3, ma usando un ciclo in modo che l'utente possa convertire più temperature. Se l'utente inserisce una lettera diversa da `C` o `F` (maiuscola o minuscola), si mostri un messaggio d'errore e si chieda all'utente di inserire un valore valido. Non si chieda, tuttavia, di re-inserire la parte numerica della temperatura. Dopo ciascuna conversione si chieda all'utente di scrivere `E` o `e`

per terminare il programma o di premere un qualsiasi altro tasto per ripetere il ciclo ed effettuare un'altra conversione.

4. Si scriva un programma che legge un elenco di numeri non negativi e mostri l'intero più grande, quello più piccolo e la media di tutti gli interi. L'utente indica la fine dell'input inserendo un valore negativo come sentinella che non deve essere considerato quando si individuano il valore più grande, più piccolo e medio. La media dei numeri inseriti deve essere un numero di tipo `double`.

5. Si scriva un programma che legge un elenco di voti ottenuti durante un esame come interi da 0 a 30. Si mostri il numero totale di voti e il numero di voti in ciascuna delle seguente categorie: Ottimo (voti 29 e 30), Distinto (voti da 26 a 28), Buono (voti da 23 a 25), Discreto (voti da 20 a 22), Sufficiente (voti 18 e 19), Insufficiente (voti da 0 a 17). Si usi un numero negativo per indicare la fine dell'inserimento.

   Se l'input fosse:

   ```
   30 29 14 26 23 28 19 23 25 12 27
   ```

   L'output potrebbe essere:

   ```
   Numero totale di voti: 11
   Numero di ottimi: 2
   Numero di distinti: 3
   Numero di buoni: 3
   Numero di discreti: 0
   Numero di sufficienti: 1
   Numero di insufficienti: 2
   ```

6. Si combino i programmi dei Progetti 4 e 5 per leggere punteggi tra 0 e 30 e mostrare le seguenti statistiche:
   - numero totale di voti;
   - numero totale di ciascun giudizio in lettere;
   - percentuale rispetto al totale per ciascun giudizio in lettere;
   - range dei voti: voto più piccolo e più grande;
   - media dei voti.

   Come nel caso precedente, si inserisca un valore numerico negativo per indicare la fine dei dati.

7. Si scriva un programma che implementi l'algoritmo dell'Esercizio 3.

8. Si scriva un programma che individua se una parola è palindroma, cioè se scritta nel verso opposto è uguale alla parola di partenza. Anna, per esempio, è una parola palindroma. Il programma deve terminare quando viene scritta la parola *uscita*.

9. Si scriva un programma che legge il saldo di un conto in banca e un tasso di interesse e mostra il valore del conto in banca tra dieci anni. L'output deve mostrare il valore del conto per tre diversi metodi di calcolo dell'interesse: annuale, mensile e giornaliero. Quando l'interesse viene calcolato annualmente, il tasso di interesse viene calcolato una volta sola per tutto l'anno. Quando viene calcolato mensilmente,

l'interesse viene aggiunto 12 volte l'anno. Quando viene calcolato giornalmente, l'interesse viene aggiunto 365 volte l'anno. Non si consideri il caso particolare degli anni bisestili, si supponga che tutti gli anni siano di 365 giorni. Per l'interesse annuo, si può presupporre che l'interesse venga calcolato esattamente un anno dopo il giorno del deposito. Analogamente, l'interesse mensile viene calcolato esattamente un mese dopo il deposito. Il saldo dovrebbe essere maggiore quando l'interesse viene calcolato giornalmente, in quanto il conto accumula interessi sugli interessi stessi. Ci si assicuri di correggere il tasso d'interesse a seconda della frequenza con cui viene applicato. Se il tasso è del 5% si usi 5/12 per calcolare l'interesse mensile e 5/365 per calcolare l'interesse giornaliero. Si effettui questo calcolo usando un ciclo che aggiunge gli interessi per ogni periodo di riferimento; non usare una formula matematica che calcoli tutto in un'operazione sola. Il calcolo deve essere ripetuto fino a quando l'utente non chiede di terminare il programma.

10. Si modifichi il Progetto 10 del Capitolo 2 affinché verifichi la validità dei dati. Un input valido non è inferiore a 25 centesimi né superiore a 100 e gli interi devono essere multipli di 5 centesimi. Si calcoli il resto solo se viene inserito un prezzo valido. Altrimenti si scriva un messaggio d'errore differente per ciascuno dei seguenti input non validi: un valore inferiore ai 25 centesimi, un valore che non è multiplo di 5, un valore superiore a 1 Euro.

11. Si scriva un programma che chieda all'utente di inserire le dimensioni di un triangolo (un intero compreso tra 1 e 50). Si visualizzi il triangolo mostrando righe di asterischi. La prima riga avrà un asterisco solo, la seconda due e così via; ciascuna riga avrà un asterisco in più della precedente fino a raggiungere il numero di righe indicato dall'utente. Per le righe successive, il numero di asterischi per riga deve decrescere di uno per ogni nuova riga. *Suggerimento*: si usino dei cicli annidati; il ciclo più esterno deve controllare il numero di righe da scrivere, mentre il ciclo interno deve controllare il numero di asterischi da scrivere in una riga. Per esempio, se l'utente scrive 3 l'output sarà:

MyLab

Video 4.4 Cicli for annidati

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

12. Si scriva un programma che simuli una palla che rimbalza calcolando la sua altezza da terra in cm per ogni secondo a mano a mano che il tempo passa su un orologio simulato. Al tempo zero la palla comincia ad altezza zero e ha una velocità iniziale data dall'utente (una velocità iniziale di 300 cm al secondo è una buona scelta). Dopo ogni secondo si cambi l'altezza aggiungendo la velocità corrente; quindi si sottragga 96 dalla velocità. Se la nuova altezza è inferiore a 0, si moltiplichi altezza e velocità per -0.5 per simulare il rimbalzo. Ci si fermi al quinto rimbalzo. L'output del programma deve avere il formato seguente:

```
Inserisci la velocita' iniziale della palla: 300
Tempo: 0 Altezza: 0.0
Tempo: 1 Altezza: 300.0
Tempo: 2 Altezza: 504.0
```

```
Tempo: 3 Altezza: 612.0
Tempo: 4 Altezza: 624.0
Tempo: 5 Altezza: 540.0
Tempo: 6 Altezza: 360.0
Tempo: 7 Altezza:  84.0
Rimbalzo!
Tempo: 8 Altezza: 144.0
. . .
```

13. Si hanno a disposizione tre premi identici da assegnare in un gruppo di dieci finalisti, ai quali sono stati associati i numeri da 1 a 10. Scrivere un programma che scelga in modo casuale i numeri dei tre finalisti che riceveranno un premio. Si faccia attenzione a non sorteggiare lo stesso numero più volte. Per esempio, l'estrazione dei finalisti 3, 6, 2 sarebbe valida, ma quella di 3, 3, 11 no perché il finalista numero 3 compare due volte e inoltre 11 non è un numero valido per un finalista. Si può semplicemente utilizzare la seguente riga di codice per generare un numero casuale tra 1 e 10:

```
int num = (int) (Math.random() * 10) + 1;
```

14. Si supponga di poter comprare barrette di cioccolato da una rivenditrice automatica per 1 € l'una. All'interno di ogni barretta c'è un buono. Con sei buoni si ha diritto a una barretta gratis. Quindi, dopo aver iniziato a comprare barrette, si hanno sempre dei buoni avanzati. Si vuole determinare quante barrette si possono avere se si parte con N euro. Per esempio, con 6 euro si possono ottenere 7 barrette, comprandone 6 (ottenendo così 6 buoni) e scambiando i 6 buoni con un'ulteriore barretta. Alla fine, avanzerebbe il singolo buono contenuto nella settima barretta. Partendo con 11 euro, si potrebbero ottenere 13 barrette avanzando ancora un buono. Con 12 euro si potrebbero ottenere 14 barrette rimanendo con due buoni avanzati.

    Scrivere un programma che riceva in ingresso un valore per N e stampi quante barrette si possono mangiare e quanti buoni avanzano alla fine. Si utilizzi un ciclo che continua a scambiare buoni con barrette ogni volta che ce ne sono abbastanza per almeno una barretta.

I metodi: concetti base

OBIETTIVI

- Definire metodi che eseguono istruzioni senza restituire un valore.
- Definire metodi che eseguono istruzioni e restituiscono un valore.
- Invocare metodi.
- Descrivere il ruolo dei parametri in una definizione di metodo.
- Passare argomenti a un metodo.
- Utilizzare i metodi della classe `Math`.
- Utilizzare *stub*, *driver* e altre tecniche per collaudare i metodi realizzati.

Nei capitoli precedenti si sono già incontrati i metodi. Basti pensare all'utilizzo che si è fatto dei metodi della classe `Scanner` per catturare i dati inseriti dall'utente tramite tastiera e al metodo `println` per stampare a video. In questo capitolo si vedrà come definire i propri metodi e come utilizzarli. Un metodo raggruppa un insieme di istruzioni assegnando loro un nome. L'insieme di istruzioni di un metodo può essere eseguito ogni volta che è necessario semplicemente riferendosi ad esso attraverso il nome. Nei linguaggi orientati a oggetti esistono due tipi di metodi: **metodi di istanza** e **metodi di classe** (o **statici**). In questo capitolo saranno introdotti i metodi di classe, mentre nei Capitoli 8 e 9 saranno presentati quelli di istanza approfondendo la differenza tra i due tipi.

Prerequisiti

Occorre aver letto il materiale presentato nei primi quattro capitoli per poter comprendere gli argomenti trattati in questo capitolo.

5.1 Definizione e invocazioni di metodi

Alcune sequenze di istruzioni possono dover essere ripetute più volte all'interno di un programma. Risulta quindi comodo poter scrivere tali sequenze una volta sola e poter far riferimento ad esse all'interno del programma tutte le volte che la loro esecuzione risulta

necessaria. I **metodi** costituiscono lo strumento di programmazione che realizza quanto sopra. Un metodo raggruppa una sequenza di istruzioni che realizzano una funzionalità del programma e assegna loro un **nome**. Ogni qualvolta è necessario eseguire quella funzionalità, è sufficiente richiamarla attraverso il nome. Un metodo è quindi una porzione di codice riusabile.

Quando si usa un metodo, si dice che si **invoca** o **chiama** il metodo. Negli esempi dei capitoli precedenti, sono già stati invocati diversi metodi. Per esempio, è stato spesso invocato il metodo `nextInt` definito nella classe `Scanner`. Inoltre, è stato spesso invocato il metodo `println` definito in `System.out`.

Si consideri, per esempio, la seguente istruzione che stampa a video un saluto:

```
System.out.println("Ciao!");
```

Invece di doverla ripetere in tutti i punti del programma in cui occorre porgere un saluto, è possibile definire un metodo che la racchiude e far riferimento a quel metodo attraverso il suo nome ogni volta che si ha bisogno di salutare.

Il Listato 5.1 riporta la definizione della classe `Saluta` che include il metodo `saluta`. Tale metodo racchiude l'istruzione che stampa a video "`Ciao!`".

MyLab

LISTATO 5.1 Definizione del metodo `saluta` e sua invocazione.

```
public class Saluta {

    public static void saluta() {

        System.out.println("Ciao!");
    }

    public static void main(String[] args) {
        System.out.println("Prima dell'esecuzione...");
        saluta();
        System.out.println("...Dopo l'esecuzione");
    }
}
```

Intestazione

Corpo

Invocazione

Esempio di output

```
Prima dell'esecuzione...
Ciao!
...Dopo l'esecuzione
```

La classe `Saluta` definita nel Listato 5.1 include anche la definizione del metodo `main` in cui è invocato il metodo `saluta` attraverso l'istruzione

```
saluta();
```

Il comportamento del `main` è equivalente al comportamento di un qualsiasi programma che esegue direttamente l'istruzione:

```
System.out.println("Ciao!");
```

Una volta terminata l'esecuzione del metodo `saluta`, il metodo `main` prosegue a eseguire le istruzioni successive a quella dell'invocazione del metodo, nell'esempio la stampa a video di "`...Dopo l'esecuzione`".

Il metodo `saluta` definito della classe `Saluta` è definito come metodo che esegue istruzioni (nel caso specifico una sola istruzione) e non restituisce alcun valore.

Java ha due tipi di metodi:

- metodi che restituiscono un valore;
- metodi che eseguono alcune istruzioni, ma non restituiscono alcun valore.

Il metodo `nextInt` della classe `Scanner` è un esempio di metodo che restituisce un singolo valore: un valore di tipo `int`. I metodi `println` e `saluta` sono esempi di metodi che eseguono delle istruzioni, ma non restituiscono alcun valore. I metodi che eseguono istruzioni, ma non restituiscono un valore sono detti **metodi `void`**.

Due tipi di metodi

Java ha due tipi di metodi: quelli che restituiscono un valore, e quelli che eseguono delle istruzioni ma non restituiscono alcun valore. Questi due metodi sono usati in modi diversi.

Come utilizzare i metodi

In primo luogo occorre **definire il metodo**:

- scrivendo la sequenza di istruzioni;
- assegnando alla sequenza un **nome**.

La definizione viene fatta una sola volta e all'interno di una classe.

Una volta definito il metodo, è possibile **invocare il metodo** usando il nome del metodo.

Quando viene invocato un metodo, vengono eseguite le istruzioni definite al suo interno.

Quando **tutte** le istruzioni del metodo sono state eseguite, l'esecuzione viene ripristinata nella posizione in cui era stata eseguita la chiamata al metodo.

5.1.1 Definire e invocare metodi `void`

Si consideri ora la definizione del metodo `saluta` per capire come sono scritte le definizioni dei metodi. La definizione è data nel Listato 5.1 e viene ripetuta di seguito:

```
public static void saluta(){
    System.out.println("Ciao!");
}
```

Un metodo è definito all'interno di una classe. Pertanto si dice che un metodo appartiene alla classe in cui è stato definito. Per esempio, se si osserva il Listato 5.1, si può notare che la definizione di metodo sopra riportata è contenuta nella definizione della classe `Saluta`. Il metodo `saluta` appartiene quindi alla classe `Saluta`.

La parola chiave `public` viene definita **modificatore d'accesso** e sarà trattata nel Capitolo 8. Per il momento basta sapere che un metodo definito `public` può essere invocato in ogni parte di un programma, anche in classi diverse da quelle in cui è stato definito. In altre parole, il termine `public` indica che non ci sono particolari restrizioni sull'uso del metodo.

La parola chiave `static` è un altro modificatore che regola il modo con cui il metodo può essere invocato. Come introdotto, esistono due tipi di metodi in Java: **metodi di classe** (o **statici**) e **metodi di istanza**. I primi hanno il modificatore `static` nell'intestazione e i secondi non lo hanno. La distinzione sarà chiarita nel Capitolo 9. Per il momento basta sapere che in questo capitolo saranno trattati solo i metodi di classe e che quindi in tutti sarà posto il modificatore `static`. In questo capitolo i termini metodo, metodo di classe e metodo statico sono pertanto intercambiabili.

Per definire un metodo come `saluta` che non restituisce alcun valore, a sinistra del nome del metodo viene specificata la parola chiave `void`. Questa parola chiave indica che il metodo non restituisce alcun valore, ovvero è un metodo `void`. Dopo il termine `void`, il nome del metodo è seguito da una coppia di parentesi. Tra parentesi è indicata la specifica degli argomenti di cui il metodo ha bisogno per poter eseguire le istruzioni in esso definite. In questo caso non è richiesta alcuna informazione, quindi tra le due parentesi non c'è nulla. Nei prossimi paragrafi si vedrà cosa è possibile specificare tra le parentesi. Questa prima parte della definizione di un metodo è detta **intestazione** (*heading*) del metodo. L'intestazione di solito viene scritta su una sola riga, ma, se è troppo lunga, può essere spezzata su due o più righe.

Dopo l'intestazione viene riportata la parte rimanente della definizione di un metodo: il **corpo** (*body*) del metodo. Le istruzioni contenute nel corpo del metodo sono racchiuse tra parentesi graffe `{}`. Nel corpo di un metodo può comparire qualsiasi istruzione che può comparire nel corpo di un programma. Le variabili utilizzate per la definizione di un metodo devono essere dichiarate all'interno della definizione del metodo stesso. Queste variabili sono dette **variabili locali** e saranno discusse più avanti.

Se si considera il Listato 5.1, si nota che la classe `Saluta` definisce oltre al metodo `saluta` anche il metodo `main`. In quest'ultimo compare l'istruzione

```
saluta();
```

L'invocazione di un metodo `void` avviene semplicemente scrivendo un'istruzione che include il nome del metodo seguito da una coppia di parentesi e da un punto e virgola. Tra le parentesi è indicato il valore degli argomenti di cui il metodo ha bisogno per eseguire le istruzioni in esso definite. In questo caso, l'intestazione del metodo non specifica alcun argomento, quindi tra le due parentesi non è riportato nulla.

Come detto, un metodo appartiene alla classe in cui è definito. Come tale, esso può essere invocato all'interno di altri metodi definiti nella sua stessa classe. Questo è il caso della classe `Saluta` del Listato 5.1 in cui il metodo `main` invoca il metodo `saluta`.

Un metodo però può essere invocato anche al di fuori della classe in cui è stato definito. In questo caso, però, occorre far precedere il nome del metodo dal nome della classe in cui è definito seguito da un punto. Il Listato 5.2 propone la classe `SalutaDemo` il cui metodo `main` invoca il metodo `saluta` definito nella classe `Saluta` del Listato 5.1 attraverso l'istruzione:

```
Saluta.saluta();
```

Se l'invocazione non fosse preceduta da `Saluta.`, la compilazione della classe `SalutaDemo` fallirebbe con un errore del tipo: metodo `saluta` non trovato nella definizione classe `SalutaDemo`. L'errore è dovuto al fatto che la definizione del metodo `saluta` viene cercata all'interno della classe in cui tale metodo viene invocato (e quindi nella classe `SalutaDemo`). Essendo definito nella classe `Saluta`, la compilazione fallisce.

In considerazione di quanto sopra, l'istruzione

```
saluta();
```

definita all'interno del metodo `main` della classe `Saluta` del Listato 5.1 può essere sostituita da

```
Saluta.saluta();
```

In questo caso, però, il nome della classe non è obbligatorio.

LISTATO 5.2 Definizione del metodo `saluta` in una classe.

MyLab

```
public class SalutaDemo {

    public static void main(String[] args) {
        Saluta.saluta();
    }
}
```

Invocazione

Esempio di output

```
Ciao!
```

Per il momento si può pensare che quando viene invocato un metodo `void`, è come se l'invocazione venisse sostituita dal corpo del metodo invocato; pertanto verranno eseguite le istruzioni definite nel corpo del metodo. In realtà il meccanismo di invocazione di metodo è più complesso e sarà presentato più avanti.

`main` è un metodo `void`

È sicuramente vero che il metodo `main` è un metodo `void`. Per ora non si consideri il significato di ciò che è riportato nelle parentesi tonde presenti nell'intestazione del metodo `main`. Ci si limiti a scriverlo; più avanti verrà spiegato il significato. Soltanto le classi che vengono eseguite come programmi devono avere un metodo `main`, tuttavia ogni classe può averne uno. Se una classe ha un metodo `main`, ma non viene eseguita come un programma, il metodo `main` viene semplicemente ignorato.

5.1.2 Definire metodi che restituiscono un valore

I metodi che restituiscono un valore vengono definiti in maniera simile ai metodi `void` con l'aggiunta della specifica del tipo di valore che restituiscono.

Come la definizione di un metodo `void`, anche la definizione di un metodo che restituisce un valore può essere divisa in due parti: l'intestazione e il corpo.

Si consideri la classe `CerchioPrimaVersione` nel Listato 5.3, la riga seguente mostra l'intestazione del metodo `areaCerchioRaggio2` in essa definito:

```
public static double areaCerchioRaggio2()
```

L'intestazione di un metodo che restituisce un valore è simile a quella di un metodo `void`. L'unica differenza è che un metodo che restituisce un valore indica, al posto della parola chiave `void`, il nome del tipo di ritorno. L'intestazione inizia con la parola chiave `public`, seguita dal modificatore `static` e quindi dal tipo del valore restituito dal metodo, seguito a sua volta dal nome del metodo e da una coppia di parentesi. Così come per i metodi `void`, le parentesi racchiudono la specifica degli argomenti di cui il metodo ha bisogno per poter essere eseguito. In questo esempio le parentesi sono vuote.

Il corpo della definizione di un metodo che restituisce un valore è come il corpo di un metodo `void`, tranne per il fatto che deve contenere almeno un'istruzione `return` al suo interno:

```
return espressione;
```

Un'istruzione `return` indica che il valore restituito dal metodo è il valore di *espressione*, la quale può essere una qualsiasi espressione che produce un valore del tipo specificato nell'intestazione del metodo. Per esempio, la definizione del metodo `areaCerchioRaggio2` contiene l'istruzione:

```
return 3.14159 * 2 * 2;
```

Così come per i metodi `void`, quando viene invocato un metodo che restituisce un valore, il sistema esegue le istruzioni contenute nel suo corpo.

Un metodo che restituisce un valore, lo si può invocare in qualsiasi punto del codice in cui si potrebbe usare un elemento dello stesso tipo di ritorno del metodo. Il metodo `areaCerchioRaggio2` restituisce un valore di tipo `double` e, quindi, si può usare l'espressione:

```
areaCerchioRaggio2()
```

o, se in una classe diversa da `CerchioPrimaVersione`, l'espressione:

```
CerchioPrimaVersione.areaCerchioRaggio2()
```

in qualsiasi punto in cui sia possibile usare un valore di tipo `double`.

Questa espressione si comporta come se fosse sostituita dal valore restituito. Per esempio, dato che il seguente assegnamento è valido:

```
double area = 12.56636;
```

allora, anche la seguente istruzione lo è:

```
double area = areaCerchioRaggio2();
```

dal momento che il valore restituito dal metodo è di tipo `double`.

L'istruzione sopra invoca il metodo `areaCerchioRaggio2` e assegna alla variabile `area` il risultato dell'istruzione definita nel corpo del metodo.

Allo stesso modo, se, per esempio, la seguente istruzione è valida:

```
System.out.println("Area del cerchio di raggio 2: " + 12.56636);
```

allora, anche la seguente lo è:

```
System.out.println("Area del cerchio di raggio 2: " + areaCerchioRaggio2());
```

Un metodo che restituisce un valore potrebbe eseguire ulteriori istruzioni oltre a ritornare un valore, tuttavia deve sempre terminare restituendo un valore.

LISTATO 5.3 Definizione del metodo `areaCerchio` e sua invocazione.

MyLab

```
public class CerchioPrimaVersione {

    public static double areaCerchioRaggio2() {

        return 3.14159 * 2 * 2;
    }

    public static void main(String[] args) {
        double area = areaCerchioRaggio2();
            System.out.println("Area del cerchio di raggio 2: "
                    + area);
            System.out.println("Area del cerchio di raggio 2: "
                    + areaCerchioRaggio2());
    }
}
```

Intestazione

Corpo

Invocazione

Esempio di output

```
Area del cerchio di raggio 2: 12.56636
Area del cerchio di raggio 2: 12.56636
```

Invocare (o chiamare) un metodo

Si invoca un metodo scrivendo il nome del metodo se la sua definizione risiede nella stessa classe in cui viene invocato, oppure scrivendo il nome della classe in cui è definito, seguito da un punto e dal nome del metodo. Seguono una coppia di parentesi che possono contenere gli argomenti, i quali passano informazioni al metodo.

Invocare un metodo che restituisce un valore

Se un metodo restituisce un valore, lo si può invocare in qualsiasi punto del programma in cui si potrebbe usare un valore dello stesso tipo di ritorno del metodo. Per esempio, l'istruzione seguente include un'invocazione del metodo `calcolaArea` e il valore restituito viene assegnato alla variabile `area` di tipo `double`:

```
double area = areaCerchioRaggio2();
```

Invocare un metodo `void`

Se un metodo esegue delle istruzioni e non restituisce un valore, si scrive semplicemente la sua invocazione seguita da un punto e virgola. L'istruzione Java corrispondente all'invocazione eseguirà le istruzioni definite nel metodo. Per esempio, quella che segue è un'invocazione al metodo `saluta`

```
saluta();
```

Questa invocazione di metodo visualizza sullo schermo la riga "`Ciao!`".

Variabili locali

Una variabile dichiarata all'interno della definizione di un metodo è una variabile locale di tale metodo. Le variabili locali possono essere usate esclusivamente all'interno del metodo che le ha definite. Inoltre, anche se due metodi hanno variabili locali con lo stesso nome, queste sono a tutti gli effetti due variabili distinte.

FAQ Cosa sono le variabili globali?

Fino a questo punto si è parlato di variabili locali, il cui significato è limitato alla definizione di un metodo. Alcuni linguaggi di programmazione hanno un altro tipo di variabile, le **variabili globali**, il cui significato riguarda l'intero programma, cioè non ha alcuna limitazione. Java non ha variabili globali. Tuttavia, come viene descritto nel Capitolo 9, Java ha un tipo di variabili, le variabili statiche, che, in un certo senso, possono essere considerate come variabili globali.

5.1.4 Blocchi

Nel Capitolo 3 si è visto che un'istruzione composta è costituita da un gruppo di istruzioni Java racchiuse tra graffe { }. Il termine blocco indica la stessa cosa. Tuttavia i due termini vengono solitamente usati in contesti differenti. Quando si dichiara una variabile all'interno di un'istruzione composta, solitamente l'istruzione composta è detta blocco.

Se si dichiara una variabile all'interno di un blocco, cioè all'interno di un'istruzione composta, la variabile è locale al blocco, cioè ha significato solamente all'interno del blocco. Nel Capitolo 4 si è detto che la porzione di programma in cui una variabile ha significato è detta visibilità della variabile (*scope*). La visibilità di una variabile locale si estende dal punto della sua dichiarazione alla fine del blocco che contiene tale dichiarazione. Questo vuol dire che al termine del blocco tutte le variabili dichiarate al suo interno scompaiono. Pertanto, è possibile usare lo stesso nome, successivamente, per definire un'altra variabile al di fuori del blocco. Se poi si dichiara precedentemente una variabile al di fuori di un blocco, la si può usare sia all'interno sia all'esterno del blocco e avrà lo stesso significato in entrambi i posti.

Blocchi

Un blocco corrisponde a un'istruzione composta, cioè, una lista di istruzioni racchiuse tra parentesi graffe. Tuttavia il termine blocco viene usato quando la dichiarazione di una variabile si trova tra parentesi graffe. Le variabili dichiarate all'interno di un blocco sono locali al blocco e, quindi, scompaiono dopo l'esecuzione del blocco.

Variabili dichiarate in un blocco

Quando una variabile viene dichiarata all'interno di un blocco, non si può usare quella stessa variabile all'esterno del blocco. Dichiarare precedentemente una variabile all'esterno del blocco permette invece di usare la variabile sia all'interno, sia all'esterno del blocco.

5.1.5 Parametri di tipo primitivo

Si consideri il metodo `areaCerchioRaggio2` della classe `CerchioPrimaVersione` definita nel Listato 5.3. Questo metodo restituisce l'area di un cerchio di raggio fissato pari a 2. E se si volesse l'area di un cerchio di raggio 10 o 3.9? Il metodo sarebbe molto più utile se potesse calcolare l'area di un cerchio con un qualsiasi raggio. Per fare ciò, è necessario lasciare degli "spazi vuoti" nel metodo e riempirli a ogni invocazione con valori diversi a seconda del raggio per cui si desidera calcolare l'area. Questi spazi vuoti vengono realizzati mediante i **parametri formali**, detti più semplicemente **parametri**. Il funzionamento dei parametri è piuttosto semplice.

Per fare un esempio di utilizzo dei parametri formali, si riformulerà la classe `CerchioPrimaVersione`. La classe `CerchioTerzaVersione` definita nel Listato 5.5 include il metodo `areaCerchio` che ha il parametro formale `raggio`. Quando si invoca questo metodo, si fornisce il valore che si vuole assegnare al parametro `raggio`. Per esempio, si può usare il metodo `areaCerchio` per calcolare l'area di un cerchio di raggio 4.5 come segue:

```
double risultato = areaCerchio(4.5);
```

Il programma presentato nel Listato 5.6 usa la classe `CerchioTerzaVersione` e il suo metodo `areaCerchio`. Come si può notare, con questa nuova versione della classe, si può calcolare l'area di un cerchio di raggio qualsiasi. Si potrebbe, inoltre, usare una variabile per il raggio come segue:

```
System.out.print("Si inserisca il raggio del cerchio: ");
Scanner tastiera = new Scanner(System.in);
double raggio = tastiera.nextDouble();
double area = CerchioTerzaVersione.areaCerchio(raggio);

System.out.println("Area del cerchio di raggio " +
        raggio + ": " + area);
```

LISTATO 5.5 Un metodo che ha un parametro.

MyLab

```
public class CerchioTerzaVersione {

    public static double areaCerchio(double raggio){
        return 3.14159 * raggio * raggio;
    }
}
```

yLab

LISTATO 5.6 Invocazione di un metodo con parametro.

```
import java.util.Scanner;

public class CerchioTerzaVersioneDemo {

    public static void main(String[] args) {
        System.out.println("Area del cerchio di raggio 2: " +
                CerchioTerzaVersione.areaCerchio(2));

        System.out.print("Si inserisca il raggio del cerchio: ");
        Scanner tastiera = new Scanner(System.in);
        double valore = tastiera.nextDouble();
        double area = CerchioTerzaVersione.areaCerchio(valore);

        System.out.println("Area del cerchio di raggio " +
                valore + ": " + area);
    }
}
```

Esempio di output

```
Area del cerchio di raggio 2: 12.56636
Risultato vale ancora: 30.8
```

Si osservi la definizione del metodo `areaCerchio`. L'intestazione del metodo, riportata di seguito, introduce una novità:

```
public static double areaCerchio(double raggio)
```

La parola `raggio` è un parametro formale. Questo parametro rappresenta una sorta di "sostituto temporaneo" di un valore effettivo che sarà disponibile solo al momento dell'invocazione del metodo. Il valore effettivo è chiamato **argomento**, come anticipato nei capitoli precedenti. In altri testi, gli argomenti sono anche detti **parametri attuali**.

Per esempio, nell'invocazione che segue, il valore `4.5` è un argomento:

```
double risultato = areaCerchio(4.5);
```

A seguito dell'istruzione sopra riportata, il valore del parametro formale `raggio` risulta pari a `4.5` (il valore dell'argomento) in tutte le sue occorrenze all'interno del metodo `areaCerchio` (sempre che il metodo, in qualche punto, non ne cambi il valore). Si noti che al parametro formale viene assegnato il valore dell'argomento. Se l'argomento di un'invocazione di metodo è una variabile, gli viene assegnato il valore della variabile e non il nome. Per esempio, si consideri l'istruzione seguente eseguita da un programma che usa la classe `CerchioTerzaVersione`:

```
double valore = 7.8;
double area = CerchioTerzaVersione.areaCerchio(valore);
```

In questo caso, è il valore `7.8`, non la variabile `valore`, che viene assegnato al parametro formale `raggio` nella definizione del metodo `areaCerchio`. Dato che viene usato il valore dell'argomento, questo meccanismo di assegnamento degli argomenti ai parametri formali è chiamato **passaggio per valore** o **chiamata per valore** (*call-by-value*). In Java,

questo è l'unico meccanismo usato per passare argomenti di tipo primitivo, come `int`, `double`, `boolean` e `char`. Come si vedrà nel Capitolo 8, quando si introducono le classi e gli oggetti, i parametri di tipo classe, invece, usano un meccanismo di assegnamento leggermente diverso.

I dettagli esatti del meccanismo di assegnamento dei parametri sono più complicati della descrizione data finora. Di solito non occorre sapere i dettagli di come avviene l'assegnamento; tuttavia, occasionalmente, potrebbe essere utile sapere alcune cose. Il parametro formale che compare nella definizione di un metodo è una variabile locale che viene inizializzata al valore dell'argomento fornito nell'invocazione del metodo. Per esempio, il parametro `raggio` del metodo `areaCerchio` del Listato 5.5, è una variabile locale di quel metodo e

```
double raggio
```

è la sua dichiarazione. Data la seguente invocazione di metodo

```
double area = CerchioTerzaVersione.areaCerchio(valore);
```

alla variabile locale raggio viene assegnato il valore dell'argomento valore.

In altre parole, è come se fosse eseguita una semplice istruzione di assegnamento come quella che segue:

```
raggio = valore;
```

A un metodo è inoltre possibile passare come argomento un'espressione il cui risultato sia dello stesso tipo del parametro formale.

Per esempio, è possibile invocare il metodo `areaCerchio` nel seguente modo:

```
double area = CerchioTerzaVersione.areaCerchio(valore * 2);
```

In questo caso, alla variabile locale `raggio` viene assegnato il risultato del calcolo dell'espressione `valore * 2`.

Si noti infine che se come argomento nell'invocazione di un metodo si usa una variabile di un tipo primitivo, l'invocazione del metodo non può cambiare il valore dell'argomento proprio perché il passaggio dei parametri avviene per valore. Il metodo `incrementa` del Listato 5.7, quando invocato nel metodo `main`, modifica il valore passato in ingresso (nello specifico 3), ma non la variabile `dato` che continua a valere 3. Il metodo `incrementaRitorna`, oltre a incrementare di una unità il valore passato in ingresso, restituisce il valore così calcolato. Assegnare tale valore alla variabile `dato`, permette di modificarne il valore che fino a quel momento era ancora 3.

LISTATO 5.7 Passaggio per valore.

MyLab

```
public class ChiamataPerValore {

    public static void incrementa(int valore) {
        valore = valore + 1;
    }

    public static int incrementaRitorna(int valore) {
        return valore + 1;
    }
```

```
    public static void main(String[] args) {
        int dato = 3;
        incrementa(dato);
        System.out.println("dato vale: " + dato);

        dato = incrementaRitorna(dato);
        System.out.println("dato vale: " + dato);
    }
}
```

Esempio di output

```
dato vale: 3
dato vale: 4
```

I nomi dei parametri sono locali del metodo

In Java si può scegliere il nome dei parametri formali di un metodo senza preoccuparsi che il nome sia già stato usato in qualche altro metodo. I parametri formali sono in realtà variabili locali e, per questo motivo, il loro significato è confinato nelle definizioni dei rispettivi metodi.

Nell'intestazione di un metodo, il tipo di dato di un parametro formale deve essere scritto prima del nome del parametro. Il metodo `areaCerchio` nel Listato 5.5 specifica che il tipo del parametro `raggio` è `double`. Ciascun parametro formale ha un tipo. Di conseguenza, il tipo dell'argomento che fornirà un valore al parametro nell'invocazione di un metodo deve corrispondere a quello del parametro. Perciò, quando si invoca il metodo `areaCerchio`, l'argomento che viene passato tra parentesi deve essere di tipo `double`.

In realtà questa regola non è così restrittiva. Infatti, in molti casi Java effettua automaticamente una **conversione di tipo** (*type cast*), qualora nell'invocazione di metodo, venga usato un argomento il cui tipo non corrisponde a quello del parametro formale. Per esempio, se il tipo dell'argomento in un'invocazione di metodo è `int` e il tipo del parametro formale è `double`, Java convertirà il valore `int` nel corrispondente valore `double`. L'elenco seguente mostra le conversioni di tipo che vengono effettuate in modo automatico da Java. Nell'invocazione di un metodo, un argomento di uno qualsiasi di questi tipi viene convertito in modo automatico in uno qualsiasi dei tipi che compare alla sua destra (nel caso in cui sia necessario ottenere una corrispondenza con il parametro formale)[1]:

```
byte → short → int → long → float → double
```

Si noti che questo elenco corrisponde a quello usato per la conversione automatica di tipo delle variabili, discussa nel Capitolo 2. Si può, quindi, descrivere la conversione automatica di tipo per le variabili e per gli argomenti con la stessa regola: si può usare un valore qualsiasi tra quelli riportati in questo elenco in una qualsiasi posizione in cui Java si aspetta un valore di un tipo più a destra nell'elenco.

Fino a questo punto, la discussione sui parametri ha coinvolto metodi che restituiscono un valore, tuttavia tutto quello che è stato detto riguardo i parametri formali e

[1] Un argomento di tipo `char` viene convertito in un tipo intero se il parametro formale è di tipo `int` o di un qualsiasi tipo a destra di `int` nella lista di tipi. Questa caratteristica non sarà usata in questo testo.

gli argomenti si applica allo stesso modo anche ai metodi `void`: i metodi `void` possono avere parametri formali, che sono gestiti esattamente allo stesso modo dei metodi che restituiscono un valore.

Parametri di tipo primitivo

Nella definizione di un metodo, un parametro formale viene descritto nell'intestazione, tra parentesi e dopo il nome del metodo. Un parametro formale di tipo primitivo, come `int`, `double`, `boolean` o `char`, è una variabile locale.

Quando viene invocato un metodo, ogni suo parametro viene inizializzato con il valore dell'argomento corrispondente nell'invocazione del metodo. Questo meccanismo è noto come *chiamata per valore*. L'argomento nell'invocazione di un metodo può essere una costante, come 2 o 'A', una variabile oppure una qualsiasi espressione che restituisce un valore del tipo appropriato.

Si noti che se come argomento nell'invocazione di un metodo si usa una variabile di un tipo primitivo, l'invocazione del metodo non può cambiare il valore dell'argomento.

È possibile, anzi è molto comune, avere più di un parametro formale nella definizione di un metodo. In questo caso, ciascun parametro formale è elencato nell'intestazione del metodo e ciascun parametro è preceduto da un tipo. La seguente riga di codice, per esempio, rappresenta l'intestazione di un metodo:

```
public static void faiQualcosa(int n1, int n2, double costo, char codice)
```

Anche se più parametri hanno lo stesso tipo, ognuno di essi deve essere preceduto dal nome di un tipo.

Il numero di argomenti passati in un'invocazione di metodo deve corrispondere con il numero di parametri formali nell'intestazione del metodo. La riga seguente, per esempio, mostra una possibile invocazione del metodo `faiQualcosa`:

```
faiQualcosa(42, 100, 9.99, 'Z');
```

Come suggerito da questo esempio, ci deve essere una corrispondenza esatta di ordine e tipo. Il primo argomento nell'invocazione al metodo fornisce un valore al primo parametro nell'intestazione del metodo, il secondo argomento fornisce un valore al secondo parametro e così via. Ciascun argomento deve corrispondere al relativo parametro in termini di tipo, a parte nel caso in cui si conti sulla conversione automatica di tipo descritta in precedenza.

Uso dei termini parametro e argomento

L'uso dei termini parametro e argomento adottato in questo testo è coerente con l'uso comune. Tuttavia, spesso questi termini vengono usati l'uno al posto dell'altro. Alcuni usano il termine parametro per indicare sia il parametro formale sia l'argomento. Altri usano invece il termine argomento con gli stessi significati. Quando si trovano i termini parametro e argomento in altri testi è bene sforzarsi di capirne il vero significato dal contesto.

Corrispondenza tra parametri formali e argomenti

Nella definizione di un metodo, i parametri formali sono indicati tra parentesi dopo il nome del metodo. Nell'invocazione di un metodo, gli argomenti sono indicati tra parentesi dopo il nome del metodo. Gli argomenti devono corrispondere ai parametri formali dell'intestazione del metodo in termini di numero, ordine e tipo. Gli argomenti forniscono un valore ai parametri formali. Il primo argomento passato nell'invocazione del metodo fornisce un valore al primo parametro, il secondo argomento al secondo parametro e così via. Gli argomenti devono essere dello stesso tipo dei parametri formali corrispondenti, sebbene in alcuni casi Java effettui una conversione di tipo automatica quando questi non corrispondono.

Intestazioni di metodo

Le intestazioni di metodo presentate finora sono tutte nella forma:

`public static` *tipo_valore_restituito_o_void nome_metodo*(*lista_parametri*)

Il termine *lista_parametri* indica una lista di nomi di parametri formali, ciascuno preceduto da un tipo. Se la lista ha più parametri, ciascun parametro è separato dal successivo mediante una virgola. Un metodo può anche non avere parametri; in questo caso all'interno delle parentesi non c'è nulla.

Esempi

```
public static int calcolaMassimo(int primo, int secondo)
public static void salutaPiuVolte(int numeroVolte)
public static void saluta()
public static double areaCerchio(double raggio)
```

5.1.6 Ancora sull'istruzione `return`

Si consideri il seguente metodo che, dati in ingresso due interi, restituisce il maggiore:

```
public static int maggiore(int primo, int secondo) {
    int risultato;
    if (primo >= secondo){
        risultato = primo;
    } else {
        risultato = secondo;
    }
    return risultato;
}
```

In questo esempio è stata usata un'unica istruzione `return` nella definizione del metodo `maggiore`. È stata inoltre definita la variabile `risultato`; le è stato assegnato il valore calcolato da restituire e quindi è stata scritta l'istruzione

```
return risultato;
```

come ultima istruzione del corpo del metodo. Un altro modo per scrivere il metodo è quello di omettere la definizione della variabile `risultato` e definire il metodo come segue:

```
public static int maggiore(int primo, int secondo){
    if (primo >= secondo){
        return primo;
    } else {
        return secondo;
    }
}
```

Alcuni programmatori ritengono che questo modo di scrivere sia chiaro, in quanto tutte le istruzioni `return` sono vicino alla fine del corpo del metodo. Tuttavia, se la logica del metodo fosse più complicata e un'istruzione `return` fosse posizionata lontano dalla fine del corpo del metodo, questo modo di scrivere il programma avrebbe complicato la comprensibilità del metodo. Per questo motivo è buona norma usare una sola istruzione `return`.

Usare un'istruzione `return`

Sebbene si possano utilizzare più istruzioni `return` all'interno del corpo di un metodo che restituisce un valore, è preferibile usarne solamente una. Inserire un'unica istruzione `return` vicino alla fine del corpo del metodo lo rende più semplice da leggere.

FAQ Può un metodo `void` contenere un'istruzione `return`?

Dato che un metodo `void` non restituisce alcun valore, tipicamente non contiene alcuna istruzione `return`. Tuttavia, anche all'interno di un metodo `void` si possono scrivere istruzioni `return` senza farle seguire da alcuna espressione:

```
return;
```

Questa istruzione si comporta come una qualsiasi istruzione `return`, con l'unica differenza che non viene indicata alcuna espressione riguardante il valore da restituire. L'esecuzione di questa istruzione `return` non fa altro che terminare l'esecuzione del metodo `void`.

Alcuni programmatori usano questa istruzione per terminare in anticipo l'invocazione del metodo, per esempio quando viene riscontrato qualche problema durante l'esecuzione. Si consideri il seguente metodo che effettua la divisione tra due numeri interi:

```
public static void divisione(int dividendo, int divisore) {
    if (divisore == 0) {
        System.out.println("Divisore uguale a 0");
        return;
    }
```

```
        int risultato = dividendo / divisore;
        System.out.println(risultato);
    }
```

L'esecuzione del metodo termina con l'esecuzione dell'istruzione `return` in modo da evitare che la parte rimanente del metodo possa incorrere in una divisione per zero. Alcune volte usare un'istruzione `return` all'interno di un metodo `void` porta a del codice chiaro e semplice da leggere, ma alle volte è possibile adottare un approccio migliore. Aggiungere un ramo `else` all'istruzione `if`, come nell'esempio che segue, chiarisce la logica del metodo:

```
public static void divisione(int dividendo, int divisore) {

    if (divisore == 0) {
        System.out.println("Divisore uguale a 0");
    } else {
        int risultato = dividendo / divisore;
        System.out.println(risultato);
    }
}
```

Definizione di metodi

Ciascun metodo appartiene alla classe in cui è definito. Ciascun metodo restituisce un singolo valore oppure non restituisce alcun valore (metodo `void`).

Sintassi

```
public static tipo_valore_restituito nome_metodo(lista_parametri) {
    istruzioni
}
```

Dove *tipo_valore_restituito* assume il valore `void` o il nome del tipo di dato restituito dal metodo. Nel secondo caso, il metodo deve contenere almeno un'istruzione nella forma:

```
return espressione;
```

Un'istruzione `return` specifica il valore restituito dal metodo e termina l'esecuzione del metodo. Un metodo `void` non deve necessariamente includere un'istruzione `return`, ma può averne una se è necessario terminarne l'esecuzione prima della fine del corpo. In tal caso, l'istruzione `return` non è seguita da alcuna espressione. Non è molto comune usare un'istruzione `return` all'interno di un metodo `void`.

Infine, *lista_parametri* indica una lista di nomi di parametri formali, ciascuno preceduto da un tipo. Se la lista ha più parametri, ciascun parametro è separato dal successivo mediante una virgola. Un metodo può anche non avere parametri; in questo caso all'interno delle parentesi non c'è nulla.

Esempi

```
public static void saluta() {
    System.out.println("Ciao");
}

public static int raddoppia(int valore) {
    return valore * 2;
}
```

5.2 La classe `Math`

La classe predefinita `Math` fornisce una serie di metodi matematici standard. Questa classe viene fornita automaticamente dal linguaggio Java, pertanto non è necessaria alcuna dichiarazione d'importazione. Alcuni dei metodi della classe `Math` sono descritti nella Figura 5.1. Tutti questi metodi sono statici. Di conseguenza, possono essere invocati utilizzando il nome della classe `Math`. L'esempio seguente visualizza il massimo tra i due numeri 2 e 3.

```
int risposta = Math.max(2, 3);
System.out.println(risposta);
```

È anche possibile omettere del tutto la variabile risposta e scrivere semplicemente:

```
System.out.println(Math.max(2, 3));
```

La classe `Math` definisce anche le due costanti `PI` e `E`. La costante `PI`, spesso scritta come π nelle formule matematiche, è utilizzata nei calcoli che coinvolgono cerchi, sfere e altre figure geometriche basate su cerchi. Il valore di `PI` è di circa `3,14159`. La costante `E` è la base dei logaritmi naturali, spesso scritta *e* nelle formule matematiche; vale circa `2,71828`.

Poiché le costanti sono definite all'interno della classe `Math`, si fa riferimento a esse come `Math.PI` e `Math.E`. Per esempio, il seguente codice calcola l'area di un cerchio dato il suo raggio:

```
area = Math.PI * raggio * raggio;
```

Si dovrebbero sempre usare le costanti `Math.PI` e `Math.E` nella definizione delle proprie classi invece di definirne le proprie versioni.

Se si osservano i metodi della tabella riportata nella Figura 5.1, si troveranno tre metodi simili, ma non identici: `round`, `floor` e `ceil`. Sebbene alcuni di questi metodi restituiscano un valore di tipo `double`, tutti restituiscono un valore che è intuitivamente un numero intero, molto vicino al valore del loro argomento. Il metodo `round` arrotonda un numero al suo valore intero più vicino. Se l'argomento è un valore di tipo `double`, il metodo restituisce un valore di tipo `long`. Se si desidera che il numero sia di tipo `int`, è necessario utilizzare una conversione di tipo (*type cast*), come nell'esempio seguente:

```
double numeroDiPartenza = 3.56;
int numeroArrotondato = (int)Math.round(numeroDiPartenza);
```

In questo esempio, a `numeroArrotondato` viene assegnato il valore `4`.

Nome	Descrizione	Tipo di argomento	Tipo restituito	Esempio	Valore restituito
pow	Potenza	double	double	Math.pow(2.0, 3.0)	8.0
abs	Valore assoluto	int, long, float o double	Lo stesso del tipo degli argomenti	Math.abs(-7) Math.abs(7) Math.abs(-3.5)	7 7 3.5
max	Massimo	int, long, float o double	Lo stesso del tipo degli argomenti	Math.max(5, 6) Math.max(5.5, 5.3)	6 5.5
min	Minimo	int, long, float, o double	Lo stesso del tipo degli argomenti	Math.min(5, 6) Math.min(5.5, 5.3)	5 5.3
round	Arroton-damento	float o double	int o long, rispettivamente	Math.round(6.2) Math.round(6.8)	6 7
ceil	Intero maggiore	double	double	Math.ceil(3.2) Math.ceil(3.9)	4.0 4.0
floor	Intero inferiore	double	double	Math.floor(3.2) Math.floor(3.9)	3.0 3.0
sqrt	Radice quadrata	double	double	sqrt(4.0)	2.0

Figura 5.1 Metodi della classe Math.

I metodi floor e ceil sono simili al metodo round, ma con piccole differenze. Restituiscono un numero intero come valore di tipo double e non di tipo int o long. Il metodo floor restituisce il numero intero più vicino minore o uguale del suo argomento. Questa azione è chiamata **arrotondamento per difetto**. Così Math.floor(3.9) restituisce il valore 3.0 (non il valore 4.0) e anche Math.floor(3.3) restituisce il valore 3.0.

Il metodo ceil, abbreviazione di *ceiling* (letteralmente "punto più alto"), restituisce il numero intero più vicino maggiore o uguale al suo argomento. Questa azione è chiamata **arrotondamento per eccesso**. Così Math.ceil(3.1) restituisce 4.0 (non 3.0); Math.ceil(3.9) restituisce il valore 4.0.

Se si vuole memorizzare il valore restituito dai metodi floor e ceil in una variabile di tipo int, bisogna usare una conversione di tipo, come mostrato nell'esempio seguente:

```
double numeroDiPartenza = 3.56;
int piuPiccolo = (int)Math.floor(numeroDiPartenza);
int piuGrande = (int)Math.ceil(numeroDiPartenza);
```

In questo esempio, Math.floor(numeroDiPartenza) restituisce un numero double il cui valore è 3.0 e la variabile piuPiccolo riceve un int di valore 3. Math.ceil(numeroDiPartenza) restituisce un numero double di valore 4.0 e la variabile piuGrande riceve un int di valore 4.

5.3 Cosa accade realmente quando si invoca un metodo?

Nel Paragrafo 5.1 si è introdotto il meccanismo di invocazione come un'operazione che intuitivamente sostituisce l'invocazione a metodo con le istruzioni presenti nel metodo invocato. In realtà, il meccanismo è un po' più complesso. Quando si invoca un metodo, l'esecuzione passa al corpo del metodo invocato e viene creata in memoria una struttura dati, chiamata **record di attivazione**, che contiene tutte le informazioni necessarie per gestire correttamente l'esecuzione del metodo invocato.

Un record di attivazione contiene dati relativi al metodo invocato e informazioni per la gestione dei metodi da esso eventualmente invocati. Le informazioni relative al metodo invocato includono **parametri formali** del metodo con gli argomenti attuali e le **variabili locali** al metodo con i valori man mano assunti durante l'esecuzione del metodo. Le informazioni per la gestione dei metodi da esso invocati includono l'**indirizzo di rientro** e il **risultato**. Il primo specifica quale istruzione del metodo deve essere eseguita nel momento in cui il metodo invocato termina la sua esecuzione. Questa informazione è necessaria per gestire correttamente il rientro dei metodi. Quando un metodo termina, infatti, il controllo torna al chiamante, che deve riprendere la sua esecuzione dall'istruzione successiva alla chiamata del metodo. Se il metodo chiamato restituisce un valore, tale valore viene copiato nel campo risultato del chiamate.

La Figura 5.2 illustra un record di attivazione per un generico metodo.

Il record di attivazione viene quindi creato dinamicamente nel momento in cui il metodo viene chiamato e viene posto in cima a un'area di memoria denominata **stack** (**pila**). Rimane nello stack per tutto il tempo in cui il metodo è in esecuzione e viene rimosso al termine dell'esecuzione. Metodi che invocano altri metodi danno luogo a una

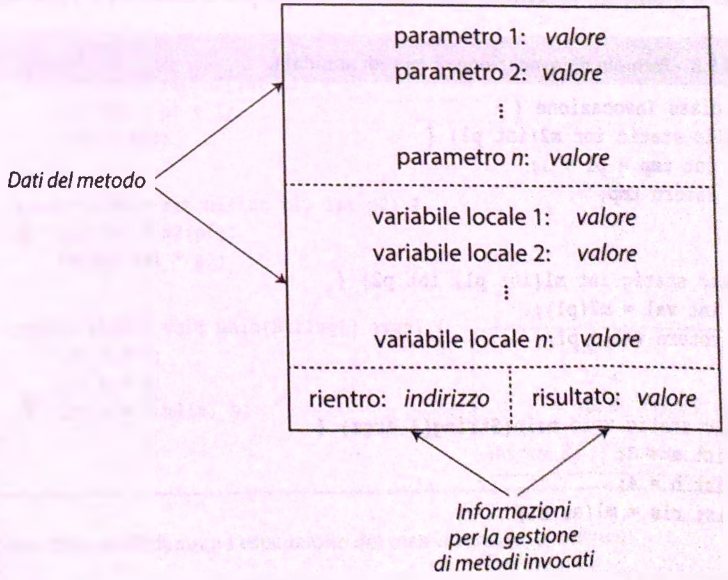


Figura 5.2 Record di attivazione per un generico metodo.

sequenza di record di attivazione gestiti secondo la politica **LIFO** (*Last In First Out*, ultimo dentro primo fuori) allocati secondo l'ordine delle chiamate e deallocati in ordine inverso.

Si consideri il Listato 5.8. La classe `Invocazione` definisce un metodo `main` che invoca il metodo `m1`, il quale, a sua volta, invoca il metodo `m2`. La Figura 5.3 illustra lo stato dello stack durante l'esecuzione del metodo `main`. Come si può notare, l'istruzione

```
int ris = m1(a, b);
```

presente all'interno del metodo `main` è stata etichettata A, mentre l'istruzione

```
int val = m2(p1);
```

presente interno del metodo `m1` ha etichetta **B**. Tali etichette possono essere considerare come gli indirizzi di memoria in cui risiedono quelle istruzioni e che saranno utilizzati per rientrare correttamente dopo l'esecuzione dei metodi.

Osservando la Figura 5.3 è possibile notare che prima dell'esecuzione dell'istruzione

```
int ris = m1(a, b);
```

nello stack risiede il record di attivazione del metodo `main`. Per semplicità, tale record non mostra i parametri formali del metodo, ma solo le variabili locali che risultano tutte inizializzate a eccezione di `ris`. Inoltre, come tutti i record di attivazione, contiene il campo per il rientro (che è indefinito) e il campo per il risultato (anch'esso indefinito).

Quando viene eseguita istruzione relativa all'invocazione del metodo `m1`, il campo `rientro` del record di attivazione del `main` viene inizializzato con l'etichetta **A**. Questo permette al metodo `main` di continuare la sua esecuzione esattamente nel punto in cui è stato interrotto una volta che l'esecuzione del metodo `m1` è terminata. Viene quindi creato un nuovo record di attivazione relativo al metodo `m1` e vengono inizializzati i suoi parametri formali `p1` e `p2` con rispettivamente i valori `3` e `4`. Il valore della variabile locale `val` è per il momento indefinito così come il campo per il rientro e il risultato.

MyLab

LISTATO 5.8 Esempio di invocazione di metodi annidata.

```
public class Invocazione {
    public static int m2(int p1) {
        int tmp = p1 + 1;
        return tmp;
    }

    public static int m1(int p1, int p2) {
        int val = m2(p1);
        return val * p1;
    }

    public static void main(String[] args) {
        int a = 3;
        int b = 4;
        int ris = m1(a, b);
    }
}
```

La prima istruzione all'interno del metodo `m1` include l'invocazione del metodo `m2` con valore 3. Ciò porta ad aggiornare il campo `rientro` del record di attivazione del metodo `m1` con l'etichetta B. Viene creato quindi un nuovo record di attivazione relativo al metodo `m2`. Il parametro viene inizializzato a 3, mentre la variabile locale `tmp` e i campi `rientro` e `risultato` hanno valori indefiniti. L'esecuzione dell'istruzione

```
int tmp = p1 + 1;
```

comporta l'aggiornamento del valore della variabile locale `tmp` all'intero del record di attivazione così da valere 4. L'ultima istruzione del metodo

```
return tmp;
```

restituisce al chiamante (il metodo `m1`) il valore della variabile `tmp` e termina l'esecuzione. In questa fase, il valore della variabile `tmp` (4) viene copiato all'interno del campo `risultato` del record di attivazione del metodo `m1`, viene eliminato il record di attivazione relativo al metodo `m2` e viene ripresa l'esecuzione del metodo `m1` a partire dall'istruzione con etichetta B (valore del campo rientro). Questa istruzione assegna il valore 4 alla variabile locale `valore`. Infine, l'ultima istruzione del metodo

```
return val * p1;
```

restituisce al chiamante (il metodo `main`) il risultato della moltiplicazione del valore di `val` per il valore di `p1` e termina l'esecuzione. In questa fase, il risultato 12 viene copiato all'interno del campo `risultato` del record di attivazione del metodo `main`, viene eliminato il record di attivazione relativo al metodo `m1` e viene ripresa l'esecuzione del metodo `main` a partire dall'istruzione con etichetta A (valore del campo `rientro`). Questa istruzione assegna alla variabile locale `ris` il valore 12. Il metodo `main` quindi termina concludendo l'esecuzione del programma che comporta la rimozione del record di attivazione del metodo `main`.

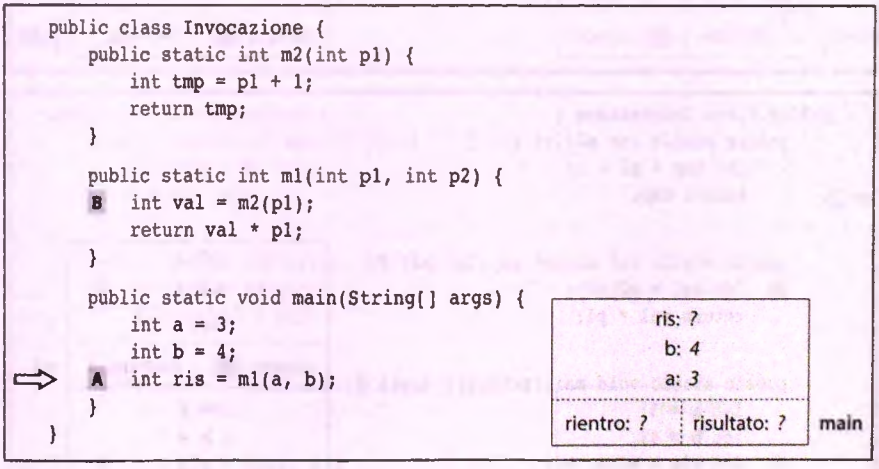


Figura 5.3 Stato dello stack durante l'esecuzione del metodo `main`. *(continua)*

```
public class Invocazione {
    public static int m2(int p1) {
        int tmp = p1 + 1;
        return tmp;
    }
⇨   public static int m1(int p1, int p2) {
    B   int val = m2(p1);
        return val * p1;
    }
    public static void main(String[] args) {
        int a = 3;
        int b = 4;
    A   int ris = m1(a, b);
    }
}
```

p1: *3*	
p2: *4*	
val: *?*	
rientro: *?*	risultato: *?*
ris: *?*	m1
b: *4*	
a: *3*	
rientro: **A**	risultato: *?*
	main

```
public class Invocazione {
⇨   public static int m2(int p1) {
        int tmp = p1 + 1;
        return tmp;
    }
    public static int m1(int p1, int p2) {
    B   int val = m2(p1);
        return val * p1;
    }
    public static void main(String[] args) {
        int a = 3;
        int b = 4;
    A   int ris = m1(a, b);
    }
}
```

p1: *3*	
tmp: *?*	
rientro: *?*	risultato: *?*
	m2
p1: *3*	
p2: *4*	
val: *?*	
rientro: **B**	risultato: *?*
	m1
ris: *?*	
b: *4*	
a: *3*	
rientro: **A**	risultato: *?*
	main

```
public class Invocazione {
    public static int m2(int p1) {
        int tmp = p1 + 1;
⇨       return tmp;
    }
    public static int m1(int p1, int p2) {
    B   int val = m2(p1);
        return val * p1;
    }
    public static void main(String[] args) {
        int a = 3;
        int b = 4;
    A   int ris = m1(a, b);
    }
}
```

p1: *3*	
p2: *4*	
val: *?*	
rientro: **B**	risultato: *4*
	m1
ris: *?*	
b: *4*	
a: *3*	
rientro: **A**	risultato: *?*
	main

Figura 5.3 Stato dello stack durante l'esecuzione del metodo `main`. (*segue*)

```
public class Invocazione {
    public static int m2(int p1) {
        int tmp = p1 + 1;
        return tmp;
    }

    public static int m1(int p1, int p2) {
B       int val = m2(p1);
        return val * p1;
    }

    public static void main(String[] args) {
        int a = 3;
        int b = 4;
A       int ris = m1(a, b);
    }
}
```

p1: *3*	
p2: *4*	
val: *4*	
rientro: B	risultato: *4*
ris: *?*	m1
b: *4*	
a: *3*	
rientro: A	risultato: *?*
	main

```
public class Invocazione {
    public static int m2(int p1) {
        int tmp = p1 + 1;
        return tmp;
    }

    public static int m1(int p1, int p2) {
B       int val = m2(p1);
        return val * p1;
    }

    public static void main(String[] args) {
        int a = 3;
        int b = 4;
A       int ris = m1(a, b);
    }
}
```

ris: *?*	
b: *4*	
a: *3*	
rientro: A	risultato: *12*
	main

```
public class Invocazione {
    public static int m2(int p1) {
        int tmp = p1 + 1;
        return tmp;
    }

    public static int m1(int p1, int p2) {
B       int val = m2(p1);
        return val * p1;
    }

    public static void main(String[] args) {
        int a = 3;
        int b = 4;
A       int ris = m1(a, b);
    }
}
```

ris: *12*	
b: *4*	
a: *3*	
rientro: A	risultato: *12*
	main

Figura 5.3 Stato dello stack durante l'esecuzione del metodo `main`. (*segue*)

5.4 Come scrivere i metodi

Questo paragrafo presenta alcune tecniche di base che aiutano a progettare, codificare e collaudare i metodi. Si partirà con un caso di studio.

CASO DI STUDIO
FORMATTAZIONE DELL'OUTPUT

Quando si usa una variabile di tipo `double` per memorizzare, per esempio, una certa somma di denaro, ci si aspetta che il programma visualizzi l'importo in un formato appropriato. Tuttavia, è probabile che si ottenga un output simile al seguente:

```
Il costo, IVA inclusa, e' di E19.98123576432
```

Ovviamente si preferirebbe avere un output come il seguente:

```
Il costo, IVA inclusa, e' di E19.98
```

Sebbene come separatore dei centesimi si usi la virgola, per rappresentare somme di denaro viene di seguito utilizzato il punto poiché si fa uso di numeri in virgola mobile. Si ricorda che i numeri in virgola mobile utilizzano come separatore per le cifre decimali il punto (.) e non la virgola.

yLab

leo 5.1
finire e
ocare
etodi
tici

In questo caso di studio si definisce la classe `FormattaEuro` contenente i due metodi statici `scrivi` e `scriviRiga` che possono essere utilizzati per produrre questo tipo di output ben formattato. Per esempio, se la somma di denaro è memorizzata nella variabile `double somma`, è possibile scrivere il seguente codice per ottenere l'output desiderato:

```
System.out.print("Il costo, IVA inclusa, e' di ");
FormattaEuro.scriviRiga(somma);
```

Si noti che i metodi `scrivi` e `scriviRiga` devono indicare che il denaro è espresso in Euro e devono sempre mostrare esattamente due cifre dopo la virgola. Quindi, questi metodi devono mostrare E2.10 e non E2.1.

Quando l'importo ha più di due cifre dopo la virgola, bisogna decidere cosa fare con le cifre in eccesso. Si supponga di voler arrotondare il numero. Quindi, se l'importo è 9,128, si otterrebbe in output E9.13.

La differenza tra `scrivi` e `scriviRiga` sarà la stessa che c'è tra `print` e `println`. Dopo aver prodotto l'output, il metodo `scrivi` non passa alla riga successiva, mentre il metodo `scriviRiga` passerà alla riga successiva. Il metodo `scriviRiga` può semplicemente invocare il metodo `scrivi` e quindi utilizzare `println` per andare alla riga successiva. Una volta decise le funzionalità di questi metodi, occorre implementarli. Per visualizzare un importo come 9.87, non si ha molta scelta: bisogna dividerlo in due parti, 9 e 87 e quindi visualizzare ogni parte separatamente. Il seguente pseudocodice illustra cosa dovrebbe fare il metodo `scrivi`.

Algoritmo per mostrare una certa somma di denaro in Euro e centesimi

(La variabile `somma` contiene la somma di denaro da visualizzare)

1. `euro` = il numero di Euro nella variabile `somma`.

2. `centesimi` = il numero di centesimi nella variabile `somma`. Si arrotondi se ci sono più di due cifre nella parte decimale.
3. Visualizza il simbolo dell'Euro, la variabile `euro` e un punto decimale.
4. Visualizza la variabile `centesimi` come un intero di due cifre decimali.

Una volta definito lo pseudocodice occorre convertirlo in codice Java. Si supponga che la variabile `somma` contenga un valore di tipo `double`. Per ottenere gli Euro e i centesimi come valori distinti e di tipo `int`, occorre liberarsi del punto decimale. Un modo per farlo è quello di convertire l'intero importo in centesimi. Per esempio, per convertire 10,95 Euro in centesimi, si moltiplica per 100 per ottenere 1095.0. Se vi è una frazione di un centesimo, come per esempio quando si convertono 10.9567 Euro in 1095.67 centesimi di Euro, è possibile usare il metodo round: `Math.round(1095.67)` che restituisce 1096 come un valore di tipo `long`.

Si noti che quello che si intende fare è memorizzare il risultato dell'arrotondamento in una variabile di tipo `int` che contiene la somma complessiva espressa in centesimi. Tuttavia, il metodo `Math.round` restituisce un valore di tipo `long` e non si può memorizzare un valore di tipo `long` in una variabile di tipo `int`, anche se si tratta di un intero di piccole dimensioni. È pertanto necessario effettuare un'operazione di conversione di tipo fra il valore `long` e il valore `int`, come nell'esempio che segue:

```
(int)(Math.round(1095.67))
```

Questa operazione restituisce il valore 1096 come un valore di tipo `int`. Perciò il codice dovrà iniziare con un'operazione come quella seguente:

```
int centesimiTotali = (int)(Math.round(somma * 100));
```

A questo punto è necessario convertire la variabile `centesimiTotali` in due valori differenti: la parte intera e la parte decimale del valore. Per esempio, dato che 1096 centesimi corrispondono a 10.96 Euro, è necessario estrarre dal numero 1096 gli interi 10 e 96. Dato che un Euro corrisponde a 100 centesimi, si può usare la divisione tra interi per ottenere il numero di Euro:

```
int euro = centesimiTotali / 100;
```

Il resto di questa divisione rappresenta il numero di centesimi di Euro. Per calcolarli basta usare l'operatore %:

```
int centesimi = cetesimiTotali % 100;
```

Le prime due azioni dello pseudocodice indicato in precedenza possono quindi essere tradotte in questo modo in linguaggio Java:

```
int centesimiTotali = (int)(Math.round(somma * 100));
int euro = centesimiTotali / 100;
int centesimi = centesimiTotali % 100;
```

Il terzo passo dello pseudocodice può essere espresso in linguaggio Java con le seguenti operazioni:

```
System.out.print('E');
System.out.print(euro);
System.out.print('.');
```

Infine, è necessario effettuare l'ultima operazione, cioè tradurre in Java l'ultima operazione dello pseudocodice:

4. Visualizza la variabile `centesimi` come un intero di due cifre decimali.

Questa operazione sembra semplice. Si supponga di effettuare la seguente operazione:

```
System.out.println(centesimi);
```

Se si prova a eseguire il codice scritto in precedenza assegnando alla variabile `somma` il valore 10.9567, si otterrà il valore:

```
E10.96
```

che sembra corretto (si noti che un modo semplice per collaudare il codice scritto è quello di assegnare un valore iniziale alla variabile `somma`). Tuttavia, quando si prova a eseguire questa operazione per altri valori si possono incontrare dei problemi. Per esempio, se `somma` è 7.05, si ottiene:

```
E7.5
```

invece di:

```
E7.05
```

Questo semplice test ci permette di capire che è necessario mostrare uno zero prima della parte decimale ogni volta che questa è inferiore al valore 10. L'istruzione seguente permette di correggere il problema:

```
if (centesimi < 10) {
    System.out.print('0');
    System.out.print(centesimi);
} else
    System.out.print(centesimi);
```

Il Listato 5.9 mostra la classe completa. Ora che la classe è stata definita in maniera completa, è tempo di effettuare alcuni test. Il Listato 5.10 mostra un programma che verifica il funzionamento del metodo `scrivi`. Questi programmi sono spesso chiamati **programmi driver** o semplicemente **driver**, perché non fanno nulla al di fuori di esercitare o "guidare" (*drive* in inglese) il metodo. Qualsiasi metodo può venir collaudato in un programma *driver*. Il test porta a risultati corretti finché non viene usato un numero negativo. Una somma di denaro negativa è comunque possibile, per esempio nel caso di debiti o di conti in rosso. Tuttavia la somma -1.20 produce l'output `E-1.0-20`. Il che indica che il metodo non gestisce correttamente valori negativi.

Se si osserva il risultato ottenuto quando `somma` è negativa, si vede che sia la parte intera sia la parte decimale contengono valori negativi. Un valore negativo per la parte intera è corretto, `E-1` nel caso in oggetto. Ma per la parte decimale bisogna visualizzare la cifra `20`, non `-20`. Questo errore può essere corretto in molti modi, tuttavia esiste una soluzione semplice e "pulita". Dato che il codice viene eseguito correttamente per valori positivi, è sufficiente convertire tutti i numeri negativi in numeri positivi e quindi mostrare il segno meno prima del numero positivo.

Il Listato 5.11 mostra la versione riveduta di questa classe. Si noti che il metodo `scriviPositivo` è molto simile al vecchio metodo `scrivi`. L'unica differenza è che `scriviPositivo` non visualizza il simbolo dell'Euro che viene invece visualizzato dalla

nuova versione del metodo `scrivi`. Ora si dovrebbe collaudare il funzionamento della classe `FormattaEuro` con un programma simile a quello usato per collaudare la classe `FormattaEuroPrimaProva`.

LISTATO 5.9 La classe `FormattaEuroPrimaProva`.

MyLab

```
public class FormattaEuroPrimaProva {

    /**
    Mostra l'ammontare in Euro e centesimi.
    Arrotonda dopo due decimali.
    Non inserisce una nuova riga dopo la fine dell'output.
    */
    public static void scrivi(double somma) {
        int centesimiTotali = (int)(Math.round(somma * 100));
        int euro = centesimiTotali / 100;
        int centesimi = centesimiTotali % 100;

        System.out.print('E');
        System.out.print(euro);
        System.out.print('.');

        if (centesimi < 10) {
            System.out.print('0');
            System.out.print(centesimi);
        } else
            System.out.print(centesimi);
    }

    /**
    Mostra l'ammontare in Euro e centesimi.
    Arrotonda dopo due decimali.
    Inserisce una nuova riga dopo la fine dell'output.
    */
    public static void scriviRiga(double somma) {
        scrivi(somma);
        System.out.println();
    }
}
```

LISTATO 5.10 Un programma *driver* che collauda la classe `FormattaEuroPrimaProva`.

MyLa

```
public class FormattaEuroPrimaProvaDriver {
    public static void main (String args[]) {
        double somma;
        String risposta;
        Scanner tastiera = new Scanner(System.in);

        System.out.println("Test FormattaEuroPrimaProva.scrivi:");
```

```
        do {
            System.out.println("Inserisci un valore di tipo double:");
            somma = tastiera.nextDouble();
            FormattaEuroPrimaProva.scrivi(somma);
            System.out.println();
            System.out.println("Testare ancora?");
            risposta = tastiera.next();
        } while (risposta.equalsIgnoreCase("si"));

        System.out.println("Fine del test.");
    }
}
```

Esempio di output

```
Test FormattaEuroPrimaProva.scrivi:
Inserisci un valore di tipo double:
1.2324
E1.23
Testare ancora?
si
Inserisci un valore di tipo double:
1.235
E1.24
Testare ancora?
si
Inserisci un valore di tipo double:
9.02
E9.02
Testare ancora?
si
Inserisci un valore di tipo double:
-1.20
E-1.0-20 <------ Oops! Qui c'è un problema.
Testare ancora?
no
Fine del test.
```

MyLab

LISTATO 5.11 La classe FormattaEuro corretta.

```
public class FormattaEuro {

    /**
    Mostra l'ammontare in Euro e centesimi.
    Arrotonda dopo due decimali.
    Non inserisce una nuova riga dopo la fine dell'output.
    */
    public static void scrivi(double somma) {
        if (somma >= 0) {
            System.out.print('E');
            scriviPositivo(somma);
        } else {
```

```
    /**
    Precondizione: somma > 0
    Mostra l'ammontare in Euro e centesimi.
    Arrotonda dopo due decimali.
    Non inserisce una nuova riga dopo la fine dell'output.
    */
    public static void scriviPositivo(double somma) {
        int centesimiTotali = (int)(Math.round(somma * 100));
        int euro = centesimiTotali / 100;
        int centesimi = centesimiTotali % 100;

        System.out.print(euro);
        System.out.print('.');

        if (centesimi < 10) {
            System.out.print('0');
            System.out.print(centesimi);
        } else
            System.out.print(centesimi);
    }

    /**
    Mostra l'ammontare in Euro e centesimi.
    Arrotonda dopo due decimali.
    Inserisce una nuova riga dopo la fine dell'output.
    */
    public static void scriviRiga(double somma) {
        scrivi(somma);
        System.out.println();
    }
}
```

Questa logica è più semplice, ma equivalente a quella utilizzata nel Listato 5.9

Ri-collaudare

Ogni volta che si cambia la definizione di un metodo è buona norma ripetere il collaudo sul nuovo metodo.

5.4.1 Decomposizione

Nel caso di studio precedente è stato usato lo pseudocodice che segue per definire un metodo che visualizza un numero di tipo `double` che rappresenta un valore monetario:

1. `euro` = il numero di Euro nella variabile `somma`.
2. `centesimi` = il numero di centesimi nella variabile `somma`. Si arrotondi se ci sono più di due cifre nella parte decimale.
3. Visualizza il simbolo dell'Euro, la variabile `euro` e un punto decimale.
4. Visualizza la variabile `centesimi` come un intero di due cifre decimali.

Quello che è stato fatto con questo pseudocodice è stato quello di **decomporre** l'attività di visualizzazione del valore monetario in più sotto-attività. Per esempio, la prima attività descritta nello pseudocodice è un'abbreviazione dell'attività:

> Calcola il numero di Euro presenti in `somma` e memorizzalo in una variabile di tipo `int` chiamata `euro`.

Ciascuna di queste sotto-attività è stata affrontata separatamente ed è stato prodotto il codice che corrisponde a ciascuna di esse. Per completare il programma si sono, quindi, combinate le implementazioni delle sotto-attività.

Se una sotto-attività è di grandi dimensioni, è bene suddividerla in sotto-attività ancora più piccole, da risolvere separatamente. Queste sotto-attività potrebbero a loro volta essere decomposte in attività più piccole, finché le attività non diventano abbastanza piccole da poter essere progettate e implementate facilmente.

5.4.2 Affrontare i problemi di compilazione

Il compilatore controlla che siano state svolte tutte le operazioni necessarie, come l'inizializzazione delle variabili o l'inclusione di un'istruzione `return` nella definizione di un metodo che deve restituire un valore. A volte il compilatore chiede di effettuare una di queste operazioni anche se il programmatore è convinto di averla effettuata o di non aver bisogno di farla. In questi casi, di solito ha ragione il compilatore. Anche se non si riesce a individuare il problema, è bene modificare comunque il codice in modo che il suo significato risulti più chiaro al compilatore.

Per esempio, si supponga di dover implementare un metodo che restituisce un valore di tipo `int` e che il metodo termini nel seguente modo:

```
if (valore1 > valore2)
    return valore1;
else if (valore1 < valore2)
    return valore2;
```

Si noti che l'istruzione non affronta il caso in cui `valore1` sia uguale a `valore2` e il compilatore potrebbe indicare che è necessario introdurre un'istruzione di ritorno. Anche se il codice implementato è corretto, perché si sa che `valore1` non sarà mai uguale a `valore2`, è necessario modificare l'istruzione `if-else` in questo modo:

```
if (valore1 > valore2)
    return valore1;
```

```
else if (valore1 < valore2)
    return valore2;
else
    return 0;
```

Oppure è possibile scrive le seguenti istruzioni equivalenti:

```
int risposta = 0;
if (valore1 > valore2)
    risposta = valore1;
else if (valore1 < valore2)
    risposta = valore2;
return risposta;
```

In precedenza è stato suggerito di scrivere metodi che non contengano più di un'istruzione di ritorno. Seguendo questa indicazione, il compilatore individuerà meno problemi. Si consideri un nuovo esempio in cui si dichiara una variabile come segue:

```
int valore;
```

Se il compilatore insiste a indicare che è necessario inizializzare la variabile, si potrebbe cambiare la dichiarazione nel modo seguente:

```
int valore = 0;
```

Tuttavia, potrebbero verificarsi situazioni in cui il compilatore insista sull'inizializzazione di una variabile anche quando non è necessario.

Il compilatore ha sempre ragione

Questa affermazione è sempre vera, perciò è bene effettuare il debugging dei programmi pensando di aver davvero commesso un errore e non il contrario, altrimenti non si riuscirà a individuare l'errore.

5.4.3 Collaudare i metodi

Per verificare la correttezza di un metodo, si può usare un programma *driver* come quello presentato nel Listato 5.10. I programmi *driver* vengono usati dal programmatore per collaudare il sistema e possono essere molto semplici. Per esempio, non hanno bisogno di output elaborati o di interfacce grafiche. Tutto quello che questi programmi devono fare è fornire al metodo da collaudare una serie di argomenti e invocare il metodo stesso.

Ogni metodo definito in una classe dovrebbe essere collaudato. Inoltre, dovrebbe essere collaudato in un programma specifico per la verifica di quel metodo e non di altri. In questo modo, in caso di output inattesi o errati, sarà più semplice risalire alla causa dell'errore.

Se si collaudano più metodi in uno stesso programma si potrebbe attribuire il difetto, cioè le istruzioni che portano a un risultato errato, a un altro metodo.

Collaudare i metodi separatamente

È buona norma collaudare ciascun metodo del sistema in programmi distinti, ciascuno focalizzato sul collaudo di un metodo ben preciso.

Un primo modo per collaudare ciascun metodo separatamente è detto **testing bottom-up** (letteralmente "collaudo dal basso verso l'alto"). Se un metodo A invoca il metodo B, l'approccio *bottom-up* prevede che il metodo B venga collaudato a fondo prima di collaudare il metodo A. L'approccio *bottom-up* è efficace, tuttavia potrebbe diventare tedioso. Altri approcci al *testing* potrebbero permettere di individuare i difetti più velocemente e con meno fatica. Alle volte potrebbe essere necessario individuare i difetti di un metodo, prima di collaudare tutti i metodi che questo invoca. Per esempio, si potrebbe voler verificare la correttezza della soluzione prima di scrivere tutti i metodi del sistema. Anche in questo caso, tuttavia, si dovrebbe collaudare ciascun metodo in un programma specifico in cui è l'unico metodo a non essere stato collaudato a fondo. Questo crea problemi nel caso in cui il metodo A invochi il metodo B, che però non è ancora stato collaudato. In questo caso, infatti, risulta difficile riuscire a far sì che A sia l'unico metodo non collaudato, a meno che non si utilizzi uno *stub* per il metodo B. Uno **stub** (letteralmente "prototipo") è una versione semplificata di un metodo che non può esser utilizzata all'interno del programma finale, ma è sufficiente per permettere il collaudo del sistema ed è abbastanza semplice per far sì che il programmatore sia certo che il risultato restituito da questo metodo sia corretto. Si supponga, per esempio, di dover collaudare la classe `FormattaEuro` del Listato 5.11 e di voler collaudare il metodo `scriviRiga` prima di aver collaudato il metodo `scrivi`. In questo caso si potrebbe usare uno *stub* per il metodo `scrivi`, nel seguente modo:

```
public static void scrivi(double somma) {
    System.out.print("E99.12"); //STUB
}
```

Questa chiaramente non è una definizione corretta del metodo `scrivi`, in quanto scrive sempre `E99.12` indipendentemente dal valore degli argomenti che riceve. Tuttavia, questa definizione è sufficiente per collaudare il metodo `scriviRiga`. Se si collauda il metodo `scriviRiga` usando questo *stub* per il metodo `scrivi` e si nota che il metodo `scriviRiga` mostra la scritta `E99.12` in modo corretto, allora il metodo `scriviRiga` è quasi sicuramente corretto. Si noti che il fatto di scrivere questo *stub* per il metodo `scrivi` permette di collaudare il metodo `scriviRiga` prima di completare sia il metodo `scrivi` sia il metodo `scriviPositivo`.

5.5 Riepilogo

- Ci sono due tipi di metodi: quelli che restituiscono un valore e i metodi `void`, che non restituiscono nulla.
- Si può usare l'invocazione di un metodo che restituisce un valore in qualsiasi posto in cui si può usare un valore dello stesso tipo di quello restituito.

- Una variabile locale è una variabile dichiarata all'interno della definizione di un metodo. La variabile non esiste al di fuori del metodo.
- Gli argomenti di un'invocazione di metodo devono corrispondere ai parametri formali dell'intestazione del metodo in termini di numero, ordine e tipo.
- I parametri formali di un metodo si comportano come variabili locali. Ciascuno viene inizializzato con il valore dell'argomento corrispondente nel momento in cui viene invocato il metodo. Questo meccanismo è detto chiamata per valore.
- I metodi possono avere parametri di tipo primitivo che vengono inizializzati al valore primitivo dell'argomento corrispondente.
- Ogni cambiamento operato su un parametro di tipo primitivo non viene effettuato sull'argomento corrispondente.
- Una definizione di metodo può includere un'invocazione a un altro metodo definito nella stessa o in un'altra classe.
- Un blocco è un'istruzione composta che dichiara variabili locali.
- Ogni volta che si scrive la definizione di un metodo è bene suddividere le attività da svolgere in sotto-attività.
- Ogni metodo dovrebbe essere collaudato in un programma scritto appositamente per verificarlo a fondo.

5.6 Esercizi

1. Si realizzi una classe Java in cui è definito il metodo `confronta` che accetta in ingresso due interi e restituisce il primo meno il secondo se il primo è maggiore del secondo, oppure restituisce il secondo meno il primo. Scrivere quindi un programma *driver* per collaudare il metodo.
2. Si realizzi una classe Java che definisce:
   a. il metodo `saluta` che accetta in ingresso una stringa `nome` e un intero `n` e stampa a video `n` volte la frase "`Ciao`" seguita dal valore di `nome`. Se per esempio viene inserito `Marco` e `3`, l'output a video dovrebbe essere:

      ```
      Ciao Marco
      Ciao Marco
      Ciao Marco
      ```

   b. il metodo `main` che chiede all'utente di inserire una stringa e un intero e invoca il metodo `saluta` con i valori letti.
3. Si realizzi una classe Java che abbia definito un metodo chiamato `divisibile` che accetta in ingresso due interi e restituisce `true` se il primo intero è divisibile per il secondo, `false` in caso contrario.
4. Si realizzi una classe Java che abbia definito un metodo che accetta in ingresso 3 interi `min`, `max` e `valore`. Tale metodo deve verificare se valore è all'interno dell'intervallo `min` – `max` estremi inclusi. Se è all'interno, il metodo restituisce `true`, `false` in caso contrario.

5. Si realizzi una classe Java che definisce:
   a. il metodo `contaVocali` che accetta in ingresso una stringa e restituisce il numero di vocali presenti nella stringa;
   b. il metodo `main` che iterativamente chiede all'utente di inserire una stringa se la stringa inserita ha un numero di vocali minore od uguale a 5. Stampa quindi il numero di vocali dell'ultima stringa inserita.
6. Si realizzi una classe Java che definisce:
   a. il metodo con nome `trova` che accetta in ingresso una stringa e un carattere e restituisce `true` se il carattere è presente almeno una volta nella stringa;
   b. il metodo `main` che legge in input due stringhe inserite dall'utente. Se le due stringhe hanno la stessa lunghezza, invoca il metodo `trova` passandogli la prima stringa e il primo carattere della seconda; se hanno lunghezza diversa, invoca il metodo `trova` passando la seconda stringa e l'ultimo carattere della prima stringa. Stampa a video il risultato dell'invocazione del metodo.
7. Si realizzi una classe Java che abbia definito il metodo `ordinaEStampa` che accetta in ingresso tre valori interi e visualizza quindi gli interi in ordine crescente. Si scriva un programma *driver* per collaudare il metodo.
8. Si realizzi una classe Java che definisce:
   a. il metodo `primo` che accetta in ingresso un numero intero e restituisce `true` se il numero è primo oppure restituisce `false` (un numero è primo se è divisibile solo per 1 o per se stesso);
   b. il metodo `divisore` che prende in ingresso un numero intero e restituisce il suo minimo divisore (escluso 1);
   c. il metodo `main` che legge in input un numero intero diverso da 0 e, utilizzando i metodi `primo` e `divisore`, stampa a video il messaggio "`il numero inserito è un numero primo`" se il numero inserito è primo, altrimenti stampa il messaggio "`il più piccolo divisore di N è D`", dove `N` e `D` devono essere il numero inserito dall'utente e il suo divisore.
9. Si realizzi una classe Java che definisce:
   a. il metodo `conta` che accetta in ingresso una stringa e un carattere e restituisce il numero di occorrenze del carattere all'interno della stringa;
   b. il metodo `main` che legge da input una stringa e un numero intero n. Invoca il metodo `conta` passandogli la stringa letta da input e il carattere che si trova in posizione n (intero letto precedentemente da input) nella stringa stessa e stampa a video un messaggio che indichi quante volte il carattere è stato trovato nella stringa.

   Esempio: `stringa = "Pippo"`, `n = 2`, il numero di volte in cui compare il carattere 'p' è 2.
10. Si realizzi una classe Java che definisce il metodo `main` che continua a chiedere in ingresso una stringa finché l'utente inserisce la parola "fine". Per ogni stringa inserita, verifica se la stringa contiene più di 5 vocali (utilizzando il metodo `contaVocali`

definito nell'Esercizio 5). Memorizza in una variabile di appoggio `piuLunga` la stringa inserita con più di 5 vocali e che è al momento la più lunga inserita. All'uscita dal ciclo stampa il valore della variabile `piuLunga`.

11. Si realizzi un programma che definisca:
    a. il metodo `inverti` che accetta in ingresso una stringa `daInvertire` e un intero n e restituisce una stringa con i caratteri invertiti a partire dal carattere di indice n, se l'indice è valido (se, per esempio, `daInvertire` = "`programmazione`" e n = 5, il metodo restituisce "`progrenoizamma`") oppure restituisce la stringa "`errore`";
    b. il metodo `main` che legge da input standard una stringa e un intero positivo, invoca il metodo `inverti` utilizzando la stringa e il numero inseriti dall'utente e stampa la stringa restituita oppure un messaggio che avverta l'utente che c'è stato un errore.
12. Si realizzi un programma che definisca:
    a. il metodo `shift` che accetta in ingresso una stringa `daShiftare` e un numero intero n e restituisce una stringa ottenuta "riavvolgendo" i caratteri della stringa di tante posizioni pari al numero passato come parametro. Per esempio, se `daShiftare` = "`programmazione`" e n = 3, il metodo restituisce "`grammazionepro`");
    b. il metodo `main` che continua a chiedere in input una stringa e un numero, invoca il metodo `shift` utilizzando nell'invocazione la stringa e il numero inseriti dall'utente ed esce dal ciclo quando il primo e l'ultimo carattere della stringa restituita dal metodo sono entrambi uguali ad '`a`'.
13. Si realizzi una classe Java che definisce:
    a. il metodo `cercaCarattere` che accetta in ingresso due stringhe, confronta a uno a uno i caratteri delle due stringhe e restituisce il primo carattere uguale trovato oppure restituisce il carattere '`*`' se le due stringhe non hanno nemmeno un carattere uguale;
    b. il metodo `main` che continua a leggere in input due stringhe e invoca il metodo `cercaCarattere` passandogli le stringhe inserite dall'utente, finché il carattere restituito dal metodo è diverso dall'ultimo carattere della prima stringa.
14. Si realizzi una classe Java che definisce:
    a. il metodo `areaRettangolo` che calcola e restituisce l'area di un rettangolo date la base e l'altezza. La base e l'altezza sono di tipo `int` così come l'area calcolata e restituita;
    b. il metodo `areaQuadrato` che accetta in ingresso il lato e sfrutta il metodo `areaRettangolo` per calcolare l'area del quadrato. L'area calcolata viene restituita. Sia il lato che l'area calcolata e restituita sono di tipo `int`;
    c. il metodo `main` che chiede all'utente un valore per la base e uno per l'altezza e stampa a video il ritorno dell'invocazione al metodo `areaRettangolo`. Chiede infine all'utente il lato di un quadrato e stampa a video il ritorno dell'invocazione al metodo `areaQuadrato`.

5.7 Progetti

1. Si definisca una classe per visualizzare valori di tipo `double`. Si chiami questa classe `DoubleOut`. Si includano tutti i metodi della classe `FormattaEuro` del Listato 5.11 e un metodo chiamato `scriviScientifico` che mostra un valore di tipo `double` che usa la notazione esponenziale, come `2.13e-12`. Questa notazione è detta anche notazione scientifica, da cui il nome del metodo. Quando viene visualizzato in notazione scientifica, un numero deve apparire con una sola cifra prima del punto decimale, a meno che il numero non sia `0`. Il metodo `scriviScientifico` non prosegue su una nuova riga. Si aggiunga, inoltre, un altro metodo chiamato `scriviScientificoNuovaRiga` che corrisponde al metodo `scriviScientifico`, ma che avanza su una nuova riga. Tutte le istruzioni, tranne le ultime due, possono essere copiate dal testo.

   Si scriva un programma *driver* per collaudare il metodo `scriviScientificoNuovaRiga`. Il programma *driver* dovrebbe usare uno *stub* al posto del metodo `scriviScientifico` (questo vuol dire che si può scrivere e collaudare `scriviScientificoNuovaRiga` prima ancora di implementare `scriviScientifico`). Si scriva, quindi, un programma *driver* per collaudare il metodo `scriviScientifico`. Infine, si scriva un programma che sia una sorta di *super-driver* che riceve in input un valore `double` e mostra il suo valore usando i metodi `scriviRiga` e `scriviScientificoNuovaRiga`. Si usi il numero `5` per il numero di cifre dopo la virgola quando occorre specificare questo numero. Quesro programma *super-driver* deve permettere all'utente di ripetere il test con numeri aggiuntivi di tipo `double` fino a che l'utente è pronto a terminare il programma.

2. Si scriva una classe `FormattaEuroTronco` che corrisponde alla classe `FormattaEuro` del Listato 5.11, tranne per il fatto che invece di arrotondare il valore dei centesimi, lo tronca per ottenere solo due cifre dopo la virgola. Le cifre in eccesso vengono troncate, ovvero rimosse. Quindi il valore `1.229`, per esempio, diventa `1.22` e non `1.23`. Si ripeta il Progetto 9 del Capitolo 4 usando questa classe.

3. Scrivere una classe Java chiamata `Calcolatrice` che definisce quattro metodi che rispettivamente sommano, moltiplicano, dividono e sottraggono due valori interi. Scrivere quindi un programma *driver* per collaudare i quattro metodi.

4. Scrivere una classe Java che contiene un metodo `determinaSegnoZodiacale` che accetta in ingresso una coppia di interi che rappresentano il giorno e il mese di nascita e determina, restituendolo in formato stringa, il segno zodiacale di appartenenza. Il metodo deve verificare la correttezza dei valori passati in ingresso (il 30 Febbraio, per esempio, non esiste). Se i valori non sono corretti, termina l'esecuzione del programma stampando a video un messaggio. Scrivere quindi un programma *driver* per collaudare il metodo. Si ricorda che i segni zodiacali sono i seguenti:

   Ariete: 21 marzo - 20 aprile

   Toro: 21 aprile - 20 maggio

   Gemelli: 21 maggio - 21 giugno

   Cancro: 22 giugno - 22 luglio

Leone: 23 luglio - 23 agosto

Vergine: 24 agosto - 22 settembre

Bilancia: 23 settembre - 22 ottobre

Scorpione: 23 ottobre - 22 novembre

Sagittario: 23 novembre- 21 dicembre

Capricorno: 22 dicembre - 20 gennaio

Acquario: 21 gennaio - 19 febbraio

Pesci: 20 febbraio - 20 marzo

5. Si ripeta il Progetto 8 del Capitolo 4 ma realizzando un metodo che determina se due stringhe sono palindrome. Scrivere quindi un programma *driver* per collaudare il metodo.
6. Si ripeta il Progetto 5 del Capitolo 3 ma realizzando due metodi che rispettivamente convertono in gradi Celsius una temperatura fornita in gradi Fahrenheit e viceversa.
7. Si consideri il Progetto 8 del Capitolo 3. Si realizzi un metodo che accetta in ingresso tre valori interi e che rappresentano in ordine il giorno, il mese e l'anno e che restituisce `true` se il formato della data risulta corretto, `false` in caso contrario. Scrivere quindi un programma `driver` per collaudare il metodo.

Array

OBIETTIVI

- Capire le caratteristiche degli array e gli scopi per cui sono utili
- Utilizzare gli array in semplici programmi Java.
- Definire metodi aventi un array come parametro.
- Definire metodi che restituiscono un array.
- Utilizzare un array non completamente pieno.
- Ordinare gli elementi di un array.
- Ricercare un particolare elemento in un array.
- Definire e utilizzare array multipli.

Un array viene utilizzato per memorizzare collezioni di dati. Tutti i dati memor nell'array devono essere dello stesso tipo. Per esempio, è possibile utilizzare un array memorizzare un elenco di valori di tipo `double` che rappresentano i centimetri di pio caduta.

In questo capitolo saranno introdotti gli array e si mostrerà il loro utilizzo in Java

Prerequisiti

I contenuti del Paragrafo 6.1 richiedono la lettura dei Capitoli da 1 a 4. I paragrafi successivi richiedono anche la comprensione di quanto presentato nel Capitolo 5.

6.1 Concetti di base sugli array

Si supponga di voler calcolare la temperatura media registrata durante i sette giorni della settimana. A tale scopo è possibile utilizzare il seguente codice:

```
Scanner tastiera = new Scanner(System.in);
System.out.println("Inserire 7 temperature:");
double somma = 0;
```

```
for (int indice = 0; indice < 7; indice++){
    double t = tastiera.nextDouble();
    somma = somma + t;
}
double media = somma / 7;
```

Il codice proposto funziona correttamente se tutto ciò che si vuole calcolare è la media delle temperature. Ora si supponga di voler sapere quali temperature sono al di sopra e quali al di sotto del valore medio. In tal caso si presenta un problema. Per realizzare quanto richiesto è necessario leggere le sette temperature, calcolare la media e infine confrontare la media con le sette temperature inserite. Quindi, per poter confrontare ogni temperatura rispetto alla media, è necessario memorizzare i valori delle sette temperature. In che modo è possibile fare questo? La risposta più ovvia prevede l'utilizzo di sette variabili di tipo `double`. Tale scelta è piuttosto inopportuna, in quanto è necessario dichiarare un numero considerevole di variabili; in questo caso sono solo sette, ma in altre situazioni il numero di variabili da dichiarare potrebbe essere notevolmente maggiore. Si consideri, per esempio, di voler eseguire gli stessi calcoli per tutti i giorni dell'anno anziché per i giorni di una sola settimana. Dichiarare 365 variabili sarebbe una scelta assurda. Gli array rappresentano un modo elegante per dichiarare una collezione di variabili fra loro correlate. Un **array** è una collezione di elementi dello stesso tipo. È paragonabile a un elenco di variabili, ma con una gestione dei nomi più immediata e compatta.

6.1.1 Creazione e accesso a un array

In Java, un array è un particolare tipo di oggetto, ma è più semplice considerarlo come una collezione di variabili dello stesso tipo. Per esempio, un array composto da una collezione di sette variabili di tipo `double` può essere creato nel seguente modo:

```
double[] temperatura = new double[7];
```

Questo equivale a dichiarare le seguenti sette variabili di tipo `double`:

```
temperatura[0], temperatura[1], temperatura[2], temperatura[3],
temperatura[4], temperatura[5], temperatura[6]
```

Le variabili come `temperatura[0]` e `temperatura[1]` che hanno un'espressione intera fra parentesi quadre sono dette **variabili indicizzate** (*indexed variables*), **elementi dell'array** (*array elements*) o semplicemente **elementi** (*elements*). L'espressione tra parentesi quadre è detta **indice** (*index*). Si noti che gli indici partono da 0 e non da 1.

Gli indici di un array partono da zero

In Java gli indici di un array partono da 0. Non iniziano mai da 1 o da un qualsiasi altro numero diverso da 0.

Ciascuna di queste sette variabili può essere utilizzata come una qualsiasi variabile di tipo `double`. Per esempio, in Java, tutte le seguenti istruzioni sono lecite:

```
temperatura[3] = 32;
temperatura[6] = temperatura[3] + 5;
System.out.println(temperatura[6]);
```

Quando si considerano queste sette variabili indicizzate come raggruppate in un unico elemento, allora ci si sta riferendo all'array. È quindi possibile far riferimento a questo array con il nome `temperatura` senza dover utilizzare le parentesi quadre. La Figura 6.1 mostra l'array `temperatura`.

Le sette variabili sono molto più che semplici variabili di tipo `double`. Il numero fra parentesi quadre è parte integrante del nome di ognuna di queste variabili e non deve essere necessariamente una costante intera. È possibile, infatti, usare una qualsiasi espressione intera che assuma valori tra 0 e 6 (per l'esempio considerato). Il seguente codice è quindi corretto:

```
Scanner tastiera = new Scanner(System.in);
System.out.println("Inserire il numero del giorno (0 - 6):");
int indice = tastiera.nextInt();
System.out.println("Inserire la temperatura per il giorno " + indice);
temperatura[indice] = tastiera.nextDouble();
```

Poiché un indice può essere un'espressione, è possibile scrivere un ciclo per l'inserimento dei valori di temperatura nell'array. Il codice è il seguente:

```
System.out.println("Inserire 7 temperature:");
for (int indice = 0; indice < 7; indice++)
    temperatura[indice] = tastiera.nextDouble();
```

L'utente può inserire i valori su righe separate oppure può inserire i valori separati da uno spazio su un'unica riga.

Una volta inseriti i valori nell'array, è possibile visualizzarli attraverso il seguente codice:

```
System.out.println("Le 7 temperature sono ");
for (int indice = 0; indice < 7; indice++)
    System.out.print(temperatura[indice] + " ");
```

Il programma nel Listato 6.1 mostra un esempio di utilizzo dell'array `temperatura`. Si noti che il programma utilizza cicli simili a quelli appena considerati.

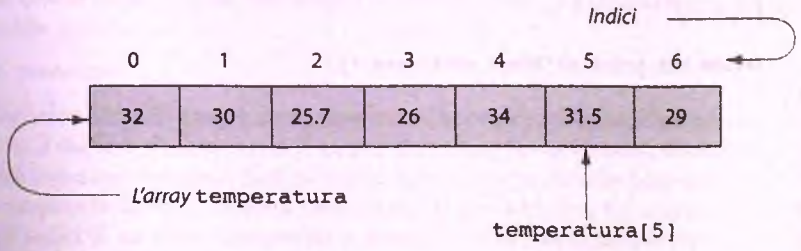


Figura 6.1 Rappresentazione classica di un array.

LISTATO 6.1 Un array di temperature.

```
/**
Legge i valori di 7 temperature inserite dall'utente e mostra quali di
esse sono al di sopra e al di sotto della media delle temperature stesse.
*/
import java.util.Scanner;

public class ArrayDiTemperature {

    public static void main(String[] args) {

        double[] temperatura = new double[7];

        // Lettura delle temperature e calcolo della loro media:
        Scanner tastiera = new Scanner(System.in);
        System.out.println("Inserire 7 temperature:");
        double somma = 0;

        for (int indice = 0; indice < 7; indice++) {
            temperatura[indice] = tastiera.nextDouble();
            somma = somma + temperatura[indice];
        }

        double media = somma / 7;
        System.out.println("La temperatura media e' " + media);

        // Mostra ogni temperatura e la relazione
        // rispetto alla temperatura media:
        System.out.println("Le 7 temperature sono");
        for (int indice = 0; indice < 7; indice++) {
            if (temperatura[indice] < media)
                System.out.println(temperatura[indice] + " sotto la media");

            else if (temperatura[indice] > media)
                System.out.println(temperatura[indice] + " sopra la media");

            else //temperatura[indice] == media
                System.out.println(temperatura[indice] + " pari alla media");
        }

        System.out.println("Buona settimana.");
    }
}
```

Esempio di output

```
Inserire 7 temperature:
12
15
16
18
15
14
15
La temperatura media e' 15.0
Le 7 temperature sono
12.0 sotto la media
15.0 pari alla media
16.0 sopra la media
18.0 sopra la media
15.0 pari alla media
14.0 sotto la media
15.0 pari alla media
Buona settimana.
```

6.1.2 Dettagli sugli array

È possibile creare un array tramite l'operazione new. Come si vedrà nel Capitolo 8, tale operazione sarà usata per creare oggetti di tipo classe. Qui viene usata con una notazione differente. Ecco la sintassi da usare per creare un array di elementi di tipo *tipo_base*:

tipo_base`[]` *nome_array* `=` `new` *tipo_base*`[`*dimensione*`];`

Per esempio, il seguente codice crea un array di nome `pressione` che è equivalente a 100 variabili di tipo `int`:

```
int[] pressione = new int[100];
```

In alternativa, l'istruzione precedente può essere spezzata nelle due istruzioni seguenti:

```
int[] pressione;
pressione = new int[100];
```

La prima istruzione dichiara `pressione` come un array di interi. La seconda istruzione alloca la memoria necessaria affinché l'array possa contenere 100 valori interi.

È possibile dichiarare un array utilizzando una sintassi alternativa che prevede le parentesi quadre dopo il nome dell'array e non dopo il tipo. La dichiarazione precedente diventerebbe quindi:

```
int pressione[];
```

Il tipo degli elementi dell'array è detto **tipo base** (*base type*) dell'array. Nell'esempio, il tipo base è `int`. Il numero di elementi dell'array è detto **lunghezza** (*length*), **dimensione** (*size*) o **capacità** (*capacity*) dell'array. Nell'esempio, l'array `pressione` ha lunghezza pari a 100, quindi comprende tutte le variabili indicizzate da `pressione[0]` a `pressione[99]`. Poiché gli indici di un array iniziano da 0, è evidente che l'array `pressione`, di 100 elementi, *non* contiene la variabile indicizzata `pressione[100]`.

Dichiarazione e creazione di un array

La dichiarazione e creazione di un array avviene mediante l'operazione new.

Sintassi

tipo_base[] *nome_array* = new *tipo_base*[*dimensione*];

o, in alternativa:

tipo_base nome_array[] = new *tipo_base*[*dimensione*];

Esempio

```
char[] simbolo = new char[80];
double[] valore = new double[100];

char simbolo[] = new char[80];
double valore[] = new double[100];
```

Come detto in precedenza, il valore fra parentesi quadre può essere una qualsiasi espressione che restituisca un intero. Quando si crea un array, al posto di usare un numero intero, è preferibile utilizzare una costante intera quando si sa esattamente il numero di elementi che l'array dovrà contenere. Per esempio, quando si crea l'array `pressione` è preferibile utilizzare una costante come `NUMERO_DI_ELEMENTI` al posto di 100:

```
public static final int NUMERO_DI_ELEMENTI = 100;
int[] pressione = new int[NUMERO_DI_ELEMENTI];
```

Java alloca a run-time la memoria per un array. Quindi, se non si conosce la dimensione di un array durante la scrittura del codice, è possibile inserirla da tastiera durante l'esecuzione, come mostrato di seguito:

```
System.out.println("Quante temperature si devono inserire?");
int dimensione = tastiera.nextInt();
double[] temperatura = new double[dimensione];
```

È altresì possibile utilizzare un'espressione (che restituisce un intero) per accedere a una certa variabile indicizzata dell'array, come nel seguente esempio:

```
int indice = 2;
temperatura[indice + 3] = 32;
System.out.println("La temperatura e' " +
                   temperatura[indice + 3]);
```

Si noti che nel precedente esempio, `temperatura[indice + 3]` è equivalente a `temperatura[5]`, in quanto `indice + 3` è uguale a `5`.

La Figura 6.2 riassume i termini più utilizzati quando si lavora con un array. Si noti che il termine *elemento* ha due significati: può essere utilizzato per riferirsi a una variabile indicizzata oppure al valore di una variabile indicizzata.

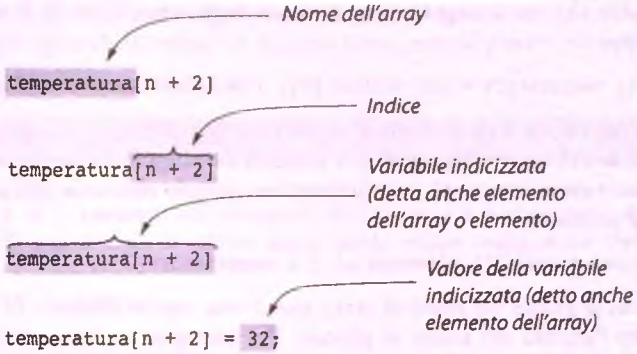


Figura 6.2 I termini relativi a un array.

Uso delle parentesi quadre con gli array

Esistono quattro diversi utilizzi delle parentesi quadre quando si lavora con gli array. Le parentesi possono essere utilizzate nei seguenti modi.

- Con il tipo di dato quando si dichiara un array. Per esempio,

  ```
  int[] pressione;
  ```

  dichiara (ma non alloca memoria) `pressione` come un array di interi.
- Con il nome dell'array quando lo si dichiara. Per esempio,

  ```
  int pressione[];
  ```

  dichiara (ma non alloca memoria) `pressione` come un array di interi.
- Per racchiudere un'espressione `int` quando si crea un nuovo array. Per esempio,

  ```
  pressione = new int[100];
  ```

  alloca la memoria necessaria per l'array `pressione` costituito da 100 interi.
- Per indicare una variabile indicizzata dell'array. Per esempio, nelle due righe seguenti `pressione[3]` è una variabile indicizzata:

  ```
  pressione[3] = tastiera.nextInt();
  System.out.println("Hai inserito " + pressione[3]);
  ```

In generale, si utilizzino nomi al singolare per gli array

Per definire un array che contiene temperature, si potrebbe usare il seguente codice:

```
double[] temperature = new double[20]; //Valido ma stilisticamente discutibile.
```

L'utilizzo del plurale, come `temperature`, sembra essere sensato, in quanto l'array contiene più elementi. Nonostante ciò, i programmatori trovano maggiormente leg-

gibile il codice che usa il singolare per il nome degli array. Quindi, il seguente codice è da preferire:

```
double[] temperatura = new double[20]; //Stilisticamente migliore.
```

La ragione per cui qui è da preferire il singolare è che, durante una qualsiasi computazione, il nome dell'array viene usato per indicare un singolo elemento dell'array stesso. L'espressione `temperatura[2]` rappresenta un singolo elemento dell'array, come nella seguente istruzione:

```
System.out.println("L'elemento e' " + temperatura[2]);
```

L'utilizzo del singolare nei nomi di array non è una regola assoluta. In certi casi è più significativo l'utilizzo del nome al plurale. Per esempio, se una variabile indicizzata contiene le ore di lavoro dell'impiegato n, la scelta della forma plurale `ore[n]` è sensata. L'unica regola per decidere se utilizzare il singolare o il plurale consiste nel vedere come risulterebbe il nome della variabile indicizzata nel contesto del codice Java.

6.1.3 La proprietà `length`

Un array è un particolare tipo di oggetto e, come si vedrà più approfonditamente nel Capitolo 8, può avere delle proprietà, chiamate anche variabili di istanza. Come si vedrà nel Capitolo 8, è possibile far riferimento alle proprietà accessibili degli oggetti utilizzando una notazione che prevede il nome dell'oggetto (nel nostro caso il nome dell'array), seguito da un punto e quindi dal nome della proprietà. Un array ha una sola proprietà accessibile: `length`. Tale variabile è uguale alla lunghezza dell'array. Per esempio, creando un array con il seguente codice:

```
double[] temperatura = new double[20];
```

`temperatura.length` ha come valore 20. Utilizzando la variabile di istanza `length` anziché il numero 20, è possibile rendere più generale e comprensibile il programma. Per chi legge il codice, un nome come `temperatura.length` è più significativo rispetto a un numero il cui significato non è sempre immediato. Inoltre, qualora si decidesse di cambiare le dimensioni dell'array, non sarebbe necessario apportare alcuna modifica alle occorrenze di `temperatura.length`. Si noti che `length` è una variabile di istanza `final`, per cui il suo valore non può essere modificato.

Assegnare un valore alla variabile di istanza `length`

All'interno di un programma non è possibile assegnare un valore alla variabile di istanza `lenght`, in quanto è una variabile `final`. Quando una variabile è dichiarata `final`, il suo valore non può essere modificato. Per esempio, la seguente istruzione non è valida:

```
temperatura.length = 10;   //Illegale!
```

`temperatura.length`. Comunque, poiché `dimensione` non è `final`, il suo valore può cambiare. Ne consegue che il valore di `dimensione` potrebbe essere diverso dal valore di `temperatura.length`.

LISTATO 6.2 Array di temperature – variante.

MyLab

```
/**
Legge i valori di 7 temperature inserite dall'utente e mostra quali di
esse sono al di sopra e al di sotto della media delle temperature stesse.
*/
import java.util.Scanner;

public class ArrayDiTemperature2 {

   public static void main(String[] args) {
      Scanner tastiera = new Scanner(System.in);
      System.out.println("Quante temperature si devono inserire?");
      int dimensione = tastiera.nextInt();
      double[] temperatura = new double[dimensione];

      // Lettura delle temperature e calcolo della loro media:
      System.out.println("Inserire " + temperatura.length + " temperature:");
      double somma = 0;

      for (int indice = 0; indice < temperatura.length; indice++) {
         temperatura[indice] = tastiera.nextDouble();
         somma = somma + temperatura[indice];
      }

      double media = somma / temperatura.length;
      System.out.println("La temperatura media e' " + media);

      // Mostra ogni temperatura e la relazione rispetto alla temperatura media:
      System.out.println("Le " + temperatura.length + " temperature sono");

      for (int indice = 0; indice < temperatura.length; indice++) {
         if (temperatura[indice] < media)
            System.out.println(temperatura[indice] + " sotto la media");
         else if (temperatura[indice] > media)
            System.out.println(temperatura[indice] + " sopra la media");
         else //temperatura[indice] == media
            System.out.println(temperatura[indice] + " pari alla media");
      }
      System.out.println("Buona settimana.");
   }
}
```

Esempio di output

```
Quante temperature si devono inserire?
```

```
Inserire 3 temperature:
32
26.5
27
La temperatura media e' 28.5
Le 3 temperature sono
32.0 sopra la media
26.5 sotto la media
27.0 sotto la media
Buona settimana.
```

Utilizzare il ciclo `for` per la scansione di un array

L'istruzione `for` costituisce il costrutto più semplice per effettuare la scansione degli elementi di un array. Per esempio, il seguente ciclo `for`, presente nel Listato 6.2, esegue una scansione di tutti gli elementi dell'array:

```
for (int indice = 0; indice < temperatura.length; indice++) {
    temperatura[indice] = tastiera.nextDouble();
    somma = somma + temperatura[indice];
}
```

6.1.4 Ulteriori dettagli sugli indici di un array

Come detto, in Java l'indice del primo elemento di un array è 0. L'ultimo indice valido per un array di lunghezza n è pertanto $n - 1$. In particolare, l'ultimo indice valido dell'array `temperatura` è `temperatura.length - 1`.

Un errore di programmazione comune quando si usano gli array è l'utilizzo, all'interno delle parentesi quadre, di espressioni che restituiscono indici non validi per l'array considerato. Si osservi per esempio la seguente dichiarazione di array:

```
double[] elemento = new double[5];
```

Gli indici validi per l'array `elemento` sono gli interi 0, 1, 2, 3 e 4. Per esempio, se il programma contiene la variabile indicizzata `elemento[n + 2]`, il valore di `n + 2` deve essere uno dei cinque precedenti interi. Se l'indice restituito da un'espressione non è compreso fra 0 e `array.length - 1`, si dice che l'indice dell'array è **fuori dai limiti** (*array index out of bounds*) o **non valido**. Anche se il codice contiene un indice non valido, viene comunque compilato senza alcun errore, ma si avrà un errore durante l'esecuzione.

Gli indici dell'array devono essere nei limiti affinché siano validi

Poiché in Java l'indice del primo elemento di un array è sempre 0, l'ultimo indice valido non coincide con la dimensione dell'array, ma con la dimensione dell'array meno uno. Occorre quindi prestare attenzione affinché gli indici siano contenuti nell'intervallo corretto.

Spesso gli indici di un array escono dai limiti previsti quando un ciclo utilizzato per la scansione dell'array viene iterato troppe volte. Si consideri, per esempio, un ciclo per riempire un array. Si supponga di voler leggere dalla tastiera una sequenza di numeri non negativi e di terminare la lettura quando viene digitato un numero negativo (valore sentinella). È possibile utilizzare il seguente codice:

```
System.out.println("Inserire una lista di interi non negativi.");
System.out.println("Terminare la sequenza con un numero negativo.");
int[] lista = new int[10];
Scanner tastiera = new Scanner(System.in);
int numero = tastiera.nextInt();
int i = 0;
while (numero >= 0) {
    lista[i] = numero;
    i++;
    numero = tastiera.nextInt();
}
```

Se l'utente inserisce un numero di valori maggiori rispetto alla capacità dell'array, il codice produce un errore a causa di un indice fuoriuscito dai limiti dell'array stesso.

Una versione migliore del precedente ciclo `while` è la seguente:

```
while ((i < lista.length) && (numero >= 0)) {
    lista[i] = numero;
    i++;
    numero = tastiera.nextInt();
}
if (numero >= 0) {
    System.out.println("Impossibile leggere ulteriori numeri.");
    System.out.println("E' possible leggere solo " + lista.length +
                       " numeri.");
}
```

Il controllo `i < lista.length` sull'indice `i` garantisce che il ciclo termini qualora l'array sia pieno.

Indici dell'array fuori dai limiti

Un indice minore di 0 oppure maggiore o uguale alla dimensione dell'array causa un messaggio d'errore durante l'esecuzione del programma.

Si supponga di voler numerare i dati presenti in un array a partire da 1. Per esempio, gli impiegati di un'azienda sono identificati a partire dal numero 1. Java, però, inizia un array con l'indice 0. Un modo per gestire tale situazione è quello di intervenire sul codice per mappare correttamente gli indici dell'array sullo schema di numerazione desiderato. Per esempio, il seguente codice potrebbe appartenere a un programma di gestione dei pagamenti:

```
public static final int NUMERO_DI_IMPIEGATI = 100;
. . .
```

```
int[] ore = new int[NUMERO_DI_IMPIEGATI];
Scanner tastiera = new Scanner(System.in);
System.out.println("Inserire le ore di lavoro di ogni impiegato:");
for (int indice = 0; indice < ore.length; indice++) {
   System.out.println("Inserire le ore per l'impiegato " +
                      (indice + 1));
   ore[indice] = tastiera.nextInt();
}
```

Con questo codice gli impiegati sono numerati da 1 a 100, ma le ore di lavoro vengono memorizzate negli elementi da `ore[0]` a `ore[99]`.

Queste situazioni possono generare confusione e portare a introdurre errori nel codice. In generale, il codice è più comprensibile e leggibile se i due schemi di numerazione coincidono. Per fare ciò, è consigliabile aumentare di 1 la dimensione dell'array e ignorare l'elemento all'indice 0, come nel codice seguente:

```
int[] ore = new int[NUMERO_DI_IMPIEGATI + 1];
Scanner tastiera = new Scanner(System.in);
System.out.println("Inserire le ore di lavoro di ogni impiegato:");
for (int indice = 1; indice < ore.length; indice++) {
   System.out.println("Inserire le ore per l'impiegato " +
                      indice);
   ore[indice] = tastiera.nextInt();
}
```

Con questo codice gli impiegati sono sempre numerati da 1 a 100 e le ore di lavoro sono memorizzate negli elementi da `ore[1]` a `ore[100]`. L'elemento `ore[0]` non viene utilizzato e, poiché la dimensione dell'array è 101, `ore[100]` è un elemento valido.

Si noti che l'ultimo valore valido di `indice` è sempre `ore.length - 1`, così come nella prima versione del codice proposto. Chiaramente, in quest'ultimo esempio, `ore.length` è più grande di 1 rispetto al primo esempio. Ma sostituendo nell'istruzione `for`:

```
indice < ore.length
```

con:

```
indice <= NUMERO_DI_IMPIEGATI
```

si rende il codice più comprensibile.

Non bisogna preoccuparsi di sprecare un elemento dell'array

Nell'esempio precedente non si utilizza `ore[0]`. È stato, quindi, "sprecato" un elemento dell'array. Ora, però, il codice è più comprensibile e meno soggetto a errori. In Java, la quantità di memoria sprecata non è significativa.

Utilizzare l'indice zero

L'indice del primo elemento di un array è 0. Se non sussistono buone motivazioni per non utilizzare l'indice 0, di norma la pratica appena descritta è da evitare. Nell'esempio precedente il fatto di allineare i due sistemi di numerazione riduce le possibilità di errore e migliora la leggibilità del codice. Comunque, modificare il codice per evitare di usare l'indice 0 non deve diventare una pratica di *routine*.

6.1.5 Inizializzare gli array

Un array può essere inizializzato in fase di dichiarazione. Per fare ciò, basta racchiudere i valori delle variabili indicizzate fra parentesi graffe dopo l'operatore di assegnamento. Si consideri il seguente esempio:

```
double[] valore = {3.3, 15.8, 9.7};
```

La dimensione dell'array (ovvero la sua lunghezza) diviene in tal caso uguale al minimo necessario per contenere i valori elencati. Il precedente codice è equivalente al seguente:

```
double[] valore = new double[3];
valore[0] = 3.3;
valore[1] = 15.8;
valore[2] = 9.7;
```

Se gli elementi di un array non vengono inizializzati esplicitamente in fase di dichiarazione, vengono comunque inizializzati col valore di default del loro tipo base. Per esempio, anche se un array di interi non viene inizializzato, ogni elemento dell'array assumerà comunque il valore 0. A ogni modo, è preferibile eseguire esplicitamente l'inizializzazione. È possibile inizializzare l'array sia con i valori tra parentesi graffe sia, come già visto, andando a leggere i valori da tastiera o, ancora, assegnando dei valori, come nel seguente ciclo `for`:

```
int[] numero = new int[100];
for (int i = 0; i < 100; i++)
    numero[i] = 0;
```

6.1.6 Array parzialmente riempiti

L'array `temperatura` introdotto nella parte iniziale del capitolo immagazzina valori di temperatura. Quando tale collezione di temperature non è piena, si ha a che fare con un **array parzialmente riempito**. In alcune situazioni non è necessario che l'array sia completamente riempito. Questo significa che ad alcune variabili indicizzate non è stato assegnato un valore utile ai fini del programma. Quando l'array è solo parzialmente riempito, è necessario tenere traccia di quanto l'array sia effettivamente utilizzato, così da conoscere anche quanto ne rimane a disposizione. Solitamente si utilizza una variabile `int` per contare gli elementi che vengono inseriti, mano a mano, nell'array. Per esempio, si supponga di utilizzare una variabile chiamata `numeroElementi` per contare le temperature inserite all'interno dell'array `temperatura`. In particolare, `numeroElementi` implica che l'array contiene elementi i cui indici vanno da `0` a `numeroElementi - 1`, come illustrato

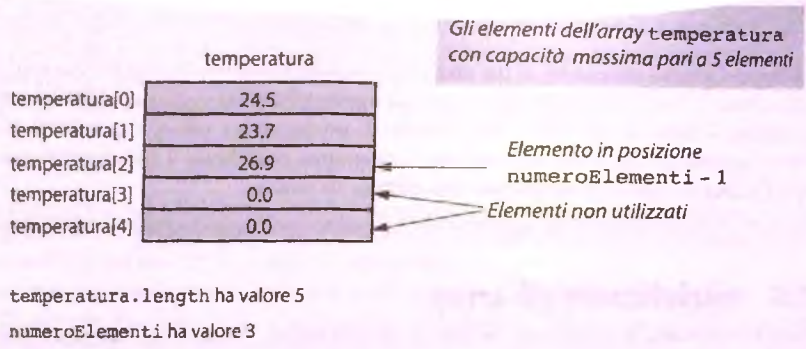


Figura 6.3 Array riempito parzialmente.

nella Figura 6.3. È molto importante tenere traccia del livello di riempimento dell'array, dato che gli altri elementi contengono valori non rilevanti. Quando si accede a un array parzialmente riempito, in realtà si vuole accedere solo agli elementi significativi. Si ignora, quindi, la parte rimanente dell'array. Ovviamente, aggiungendo o cancellando elementi dall'array parzialmente riempito, il confine tra valori significativi e non si può spostare. Di conseguenza, tale spostamento deve essere opportunamente registrato modificando il valore della variabile `int numeroElementi`.

Tipicamente, la parte di array che contiene valori significativi comincia all'inizio dell'array, ma questo non è un requisito obbligatorio. Per considerare una porzione di array che non inizia dall'indice 0, si devono definire due variabili `int` (come `inizio` e `fine`) per memorizzare gli indici del primo e dell'ultimo elemento dell'insieme che si vuole considerare.

Numero di elementi di un array vs. lunghezza dell'array

Occorre distinguere tra *il numero di elementi utilizzati* all'interno di un array e *la sua capacità*. La capacità dell'array `a` è esattamente `a.length`. Questo è il numero di elementi che sono stati creati quando si è definito l'array. È possibile non utilizzare tutti questi elementi. In questo caso è necessario tenere traccia di quanti sono effettivamente utilizzati. Tipicamente si hanno due interi: uno è la capacità dell'array e uno è il numero di elementi che sono effettivamente utilizzati.

6.1.7 Utilizzare il ciclo `for-each` con gli array

Come si è già visto, è possibile utilizzare un ciclo `for` per scorrere gli elementi di un array. Per esempio,

```
double [] a = new double [10];
<Codice che riempia l'array a>
for (int i = 0; i < a.length; i++)
    System.out.println(a[i]);
```

A partire dalla versione 5 di Java, è stato introdotto un nuovo tipo di ciclo che non richiede di utilizzare esplicitamente gli indici degli elementi dell'array. Questo nuovo tipo di ciclo `for` è detto ciclo **`for-each`** o **ciclo `for` potenziato** ed è stato già brevemente introdotto nel Capitolo 4. Il codice che segue contiene un ciclo `for-each` equivalente al normale ciclo `for` utilizzato nell'esempio precedente:

```
double [] a = new double [10];
<Codice che riempia l'array a>
for (double elemento : a)
    System.out.println(elemento);
```

Si può leggere la linea che inizia con il `for` come "per ogni elemento in a, esegui le operazioni che seguono". Si noti che la variabile `elemento` è dello stesso tipo degli elementi dell'array. La variabile deve essere dichiarata nel ciclo `for-each` come mostrato. Se si provasse a dichiarare la variabile `elemento` prima del ciclo, si otterrebbe un errore in fase di compilazione.

La sintassi generale per l'uso di un ciclo `for-each` con un array è

```
for (tipo_base variabile : nome_array)
    istruzione
```

Si faccia attenzione a utilizzare i due punti (non un punto e virgola) dopo la variabile. Naturalmente è possibile utilizzare qualunque nome di variabile ammesso per la variabile, non è obbligatorio usare `elemento`. Nonostante non sia necessario, tipicamente le istruzioni all'interno del ciclo utilizzeranno la variabile. Quando il ciclo viene eseguito, le istruzioni corrispondenti vengono eseguite una volta per ogni elemento dell'array. Più precisamente, per ogni elemento dell'array, la variabile viene posta uguale all'elemento e successivamente vengono eseguite le istruzioni.

L'utilizzo del ciclo `for-each` può rendere il codice molto più pulito e meno soggetto a errori. Se non serve utilizzare l'indice dell'array in un ciclo `for` per altro scopo che per scorrere gli elementi dell'array, è preferibile un ciclo `for-each`. Per esempio,

```
for (double elemento : a)
    somma += elemento;
```

è preferibile a

```
for (int i = 0; i < a.length; i++)
    somma += a[i];
```

I due cicli fanno la stessa cosa, ma il secondo utilizza l'indice `i`, che non ha altro scopo che scorrere gli elementi dell'array. Inoltre, la sintassi del ciclo `for-each` è più semplice di quella del ciclo `for` normale.

D'altra parte, il seguente ciclo `for` dovrebbe essere lasciato così com'è, senza cercare di convertirlo in un `for-each`:

```
for (int i = 0; i < a.length; i++)
    a[i] = 2 * i;
```

Questo ciclo, infatti, utilizza l'indice `i` nel corpo del ciclo in modo essenziale: non avrebbe senso convertirlo in un `for-each`.

Uso del ciclo `for-each` con gli array

Sintassi

```
for (tipo_base variabile : nome_array)
    istruzione
```

Esempi

```
for (double elemento : a)
    somma += elemento;
```

In questo esempio, l'array `a` ha come tipo base `double`. Questo ciclo scorre tutti gli elementi dell'array e somma ogni elemento alla variabile `somma`.

Un buon modo per leggere la prima riga dell'esempio è "Per ogni elemento nell'array `a`, esegui le seguenti istruzioni".

6.2 Utilizzare gli array nei metodi

I metodi possono ricevere come argomento una variabile indicizzata o un intero array e possono restituire un array.

6.2.1 Variabili indicizzate come argomenti di un metodo

Una variabile indicizzata di un array `a`, come `a[i]`, può essere utilizzata ogni volta che è possibile utilizzare una variabile del tipo base dell'array. Una variabile indicizzata può quindi essere un argomento di un metodo, così come ogni altra variabile dello stesso tipo base dell'array.

Per esempio, il programma nel Listato 6.3 mostra l'utilizzo di una variabile indicizzata come argomento di un metodo. Il metodo `getMedia` ha due argomenti di tipo `int`. L'array `punteggioSeguente` ha `int` come tipo base e quindi il programma può utilizzare `punteggioSeguente[i]` come argomento del metodo `getMedia` come mostrato nella seguente riga di codice:

```
double possibileMedia = getMedia(punteggioIniziale, punteggioSeguente[i]);
```

La variabile `punteggioIniziale` è una normale variabile di tipo `int`. Per evidenziare il fatto che la variabile `punteggioSeguente[i]` può essere utilizzata come una qualsiasi variabile di tipo `int`, si noti come `getMedia` si comporterebbe allo stesso modo scambiando i suoi due argomenti:

```
double possibileMedia = getMedia(punteggioSeguente[i], punteggioIniziale);
```

La definizione del metodo `getMedia` non contiene alcuna indicazione del fatto che i suoi argomenti possono essere variabili indicizzate di un array di `int`. Il metodo accetta argomenti di tipo `int`, ma non è importante sapere se questi interi provengono da variabili indicizzate, da comuni variabili di tipo `int` o da costanti intere.

C'è un aspetto che è importante sottolineare quando si utilizzano le variabili indicizzate come argomento di un metodo. Si considerino, per esempio, le precedenti chiamate del metodo. Se il valore di `i` è 2, l'argomento è `punteggioSeguente[2]`. Allo stesso modo, se il valore di `i` è 0, l'argomento è `punteggioSeguente[0]`. L'espressione usata come indice viene valutata per determinare quale variabile indicizzata rappresenta l'argomento del metodo.

È importante notare che una variabile indicizzata di un array `a`, come `a[i]`, è una variabile del tipo base dell'array. Quando `a[i]` viene usata come argomento di un metodo, viene gestita esattamente come una variabile del tipo base dell'array `a`. In particolare, se il tipo base dell'array è un tipo primitivo, come `int`, `double` o `char`, il metodo non può modificare il valore di `a[i]`. Questo non deve stupire. Si ricordi che una variabile indicizzata, come `a[i]`, è una variabile del tipo base dell'array e viene gestita allo stesso modo di ogni altra variabile di quel tipo.

LISTATO 6.3 Variabili indicizzate come argomento di un metodo.

MyLab

```
import java.util.Scanner;

/**
Utilizzo di variabili indicizzate.
*/
public class ArgomentiDemo {

   public static void main(String[] args) {
      Scanner tastiera = new Scanner(System.in);
      System.out.println("Inserire il voto dell'esame 1:");
      int punteggioIniziale = tastiera.nextInt();
      int[] punteggioSeguente = new int[3];

      for (int i = 0; i < punteggioSeguente.length; i++)
         punteggioSeguente[i] = punteggioIniziale + 5 * i;

      for (int i = 0; i < punteggioSeguente.length; i++) {
         double possibileMedia =
            getMedia(punteggioIniziale, punteggioSeguente[i]);
         System.out.println("Se il voto all'esame 2 sara' " +
                            punteggioSeguente[i]);
         System.out.println("la media sara' uguale a " + possibileMedia);
      }
   }

   public static double getMedia(int n1, int n2) {
      return (n1 + n2) / 2.0;
   }
}
```

```
Se il voto all'esame 2 sara' 20
la media sara' uguale a 20.0
Se il voto all'esame 2 sara' 25
la media sara' uguale a 22.5
Se il voto all'esame 2 sara' 30
la media sara' uguale a 25.0
```

Variabili indicizzate come argomento

Una variabile indicizzata può essere usata come argomento di un metodo in tutte le situazioni in cui può essere utilizzato il tipo base dell'array. Per esempio, si consideri:

```
double[] a = new double[10];
```

Variabili indicizzate come `a[3]` e `a[indice]` possono essere usate come argomenti di qualsiasi metodo che accetta come argomento una variabile `double`.

FAQ Quando un metodo può modificare un argomento che è una variabile indicizzata?

Sia `a[i]` una variabile indicizzata dell'array `a` e sia `a[i]` l'argomento in un'invocazione di un metodo come:

```
mioMetodo(a[i]);
```

Il fatto che `mioMetodo` possa modificare o meno l'elemento `a[i]` dipende dal tipo base dell'array `a`. Se il tipo base dell'array `a` è un tipo primitivo, come `int`, `double` o `char`, il metodo `mioMetodo` riceve il *valore* di `a[i]`. Come si vedrà più avanti quando si tratteranno le classi e gli oggetti, se il tipo base dell'array `a` è una classe, il metodo `mioMetodo` riceve un riferimento (*reference*) ad `a[i]`. Il metodo è quindi in grado di modificare lo *stato* dell'oggetto referenziato da `a[i]`, ma non può sostituire l'oggetto con un altro.

6.2.2 Array come argomenti di un metodo

Si è già visto come una variabile indicizzata possa essere utilizzata come argomento di un metodo. Il modo con cui si specifica che l'argomento di un metodo è un array è simile al modo con cui si dichiara un array. Per esempio, il seguente metodo `incrementaArrayDi2` accetta come argomento un qualsiasi array di `double`:

```
public class ClasseEsempio {
    public static void incrementaArrayDi2(double[] unArray) {
        for (int i = 0; i < unArray.length; i++)
            unArray[i] = unArray[i] + 2;
    }
    <Inserire qui il restante codice per la definizione della classe.>
}
```

Quando si utilizza come parametro un array, è necessario indicare il tipo base dell'array, ma non si deve impostare la lunghezza dell'array stesso.

È possibile utilizzare una sintassi alternativa per specificare che un array è un argomento di un metodo. Tale sintassi è simile a quella alternativa utilizzata in fase di dichiarazione di un array: è possibile specificare le parentesi quadre dopo il nome dell'array invece che dopo il tipo. La dichiarazione del precedente metodo diventa quindi:

```
public static void incrementaArrayDi2(double unArray[]) {
```

Per illustrare l'utilizzo della classe `ClasseEsempio`, si consideri che le istruzioni

```
double[] a = new double[10];
double[] b = new double[30];
```

siano inserite all'interno della definizione di un qualche metodo e si supponga che agli elementi degli array `a` e `b` siano stati assegnati dei valori. Entrambe le seguenti invocazioni di metodo sono corrette:

```
ClasseEsempio.incrementaArrayDi2(a);
ClasseEsempio.incrementaArrayDi2(b);
```

Il metodo `incrementaArrayDi2` accetta come argomento un array di qualsiasi dimensione ed è in grado di modificare i valori degli elementi dell'array. Dopo le precedenti invocazioni del metodo, gli elementi degli array `a` e `b` vengono incrementati di 2.

Array come parametri

Anche un intero array può essere utilizzato come argomento di un metodo. Si usa la seguente sintassi nell'intestazione del metodo.

Sintassi

```
public static tipo_di_ritorno nome_del_metodo(tipo_base[] nome_parametro)
```

o, in alternativa:

```
public static tipo_di_ritorno nome_del_metodo(tipo_base nome_parametro[])
```

Esempi

```
public static int getUnElemento(char[] unArray, int indice)
public static void leggiArray(int[] unArray)

public static int getUnElemento(char unArray[], int indice)
public static void leggiArray(int unArray[])
```

Non si deve specificare la lunghezza degli array nell'intestazione del metodo

All'interno dell'intestazione del metodo si deve specificare il tipo base dell'array, ma non la lunghezza dell'array. Per esempio, la seguente intestazione di un metodo specifica come parametro un array di caratteri:

```
public static void visualizzaArray(char[] a)
```

Utilizzo degli array come argomenti di un metodo

- Quando si passa un intero array come argomento a un metodo, non devono essere usate le parentesi quadre.
- Si può passare un array di qualsiasi lunghezza come argomento a un metodo che accetta come parametro un array.
- Un metodo può modificare il valore degli elementi di un array passato come argomento.

Ognuna di queste proprietà è stata presentata nel metodo `incrementaArrayDi2`.

6.2.3 Argomenti del metodo `main`

L'intestazione del metodo `main` di un programma è la seguente:

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

La dichiarazione del parametro `String[] args` indica che `args` è un array il cui tipo base è `String`. Di conseguenza, il metodo `main` accetta come parametro un array di valori di tipo `String`. Ma finora non si è mai passato alcun argomento al metodo `main`. In effetti, il metodo `main` non è mai stato invocato! Come funzionano le cose?

L'invocazione del metodo `main` è particolare: non viene effettuata esplicitamente. Quando si esegue un programma, il `main` viene invocato automaticamente e come argomento gli viene fornito un array di stringhe di default. È però possibile fornire delle stringhe come argomento del programma e queste stringhe diventano automaticamente elementi dell'array `args` che rappresenta l'argomento di `main`. Di solito si passano argomenti a un programma quando lo si esegue dalla riga di comando, come nel seguente esempio:

```
java ProgrammaDiTest Mario Rossi
```

Questo comando assegna `"Mario"` ad `args[0]` e `"Rossi"` ad `args[1]`. Queste due variabili indicizzate possono essere utilizzate all'interno del metodo `main`.

Per esempio, si consideri il seguente codice:

```
public class ProgrammaDiTest {
    public static void main(String[] args) {
        System.out.println("Ciao " + args[0] + " " + args[1]);
    }
}
```

Dopo aver lanciato `ProgrammaDiTest` usando il comando:

```
java ProgrammaDiTest Luca Bianchi
```

l'output prodotto dal programma sarà:

```
Ciao Luca Bianchi
```

È importante sottolineare che l'argomento del `main` è un array di *stringhe*. Se si vogliono utilizzare numeri, si devono convertire le relative stringhe in uno dei tipi numerici. Dal momento che l'identificatore `args` è un parametro, è possibile utilizzare un qualsiasi altro identificatore al posto di `args` e quindi cambiare ogni occorrenza di `args` presente nel corpo del `main` col nome del nuovo identificatore. L'utilizzo dell'identificatore `args` per indicare il parametro del main è però una pratica comune.

6.2.4 Assegnamento e uguaglianza di array

Anche se la gestione degli oggetti in memoria sarà trattata nel Capitolo 8, è importante qui anticipare alcuni concetti per comprendere il funzionamento degli operatori = e ==, poiché gli array sono oggetti. Essendo tali, gli operatori di assegnamento (=) e di uguaglianza (==) si comportano con gli array allo stesso modo in cui si comportano con ogni altro oggetto. Per capire il loro funzionamento con gli array, è necessario comprendere meglio come gli array vengono memorizzati dal computer. L'aspetto importante è che l'intero contenuto dell'array (cioè il contenuto di tutte le variabili indicizzate) è memorizzato in un'unica area di memoria. In tal modo, la posizione dell'array può essere specificata con un unico indirizzo di memoria.

Come sarà ampiamente trattato nel Capitolo 8, una variabile a cui viene assegnato un oggetto contiene l'indirizzo di memoria in cui si trova l'oggetto stesso.

L'operatore di assegnamento copia questo indirizzo. Per esempio, si consideri il seguente codice:

```
int[] a = new int[3];
int[] b = new int[3];
for (int i = 0; i < a.length; i++)
    a[i] = i;
b = a;
System.out.println("a[2] = " + a[2] + ", b[2] = " + b[2]);
a[2] = 2001;
System.out.println("a[2] = " + a[2] + ", b[2] = " + b[2]);
```

L'output prodotto è il seguente:

```
a[2] = 2, b[2] = 2
a[2] = 2001, b[2] = 2001
```

L'assegnamento `b = a` nel codice precedente, assegna alla variabile `b` lo stesso indirizzo di memoria della variabile `a`. Quindi, `a` e `b` diventano due diversi nomi per lo stesso array. Di conseguenza, quando si modifica il valore di `a[2]`, si modifica anche il valore di `b[2]`. Per questo motivo, è preferibile non utilizzare l'operatore = con gli array. Se si vuole che gli array `a` e `b` siano distinti, ma che contengano gli stessi valori, si deve scrivere un codice come il seguente:

```
for (int i = 0; i < a.length; i++)
    b[i] = a[i];
```

al posto dell'istruzione di assegnamento:

```
b = a;
```

Si noti che nel ciclo precedente si suppone che i due array a e b abbiano la stessa lunghezza.

L'operatore di uguaglianza == verifica se due array sono memorizzati nella stessa area di memoria del computer. Per esempio, il codice:

```
int[] a = new int[3];
int[] b = new int[3];
for (int i = 0; i < a.length; i++)
    a[i] = i;
for (int i = 0; i < b.length; i++)
    b[i] = i;
if (b == a)
    System.out.println("Uguali secondo ==");
else
    System.out.println("Non uguali secondo ==");
```

produce come output:

```
Non uguali secondo ==
```

Anche se gli array a e b contengono gli stessi valori nello stesso ordine, gli array sono memorizzati in differenti aree di memoria. Per tale motivo, b == a è falso, poiché == verifica l'uguaglianza fra gli indirizzi di memoria.

Se si vuole verificare se due array contengono gli stessi elementi, si deve eseguire il confronto elemento per elemento. Il Listato 6.4 contiene un esempio che mostra come eseguire tale confronto.

Utilizzo degli operatori = e == con gli array

È possibile utilizzare l'operatore di assegnamento = per assegnare più nomi a uno stesso array. Non è possibile usare l'operatore = per copiare il contenuto di un array in un altro array. Analogamente, l'operatore di uguaglianza == verifica se due array si riferiscono alla stessa area di memoria. L'operatore == non verifica se due array contengono gli stessi elementi.

LISTATO 6.4 Due tipologie di uguaglianza.

```
/**
Esempio di programma che verifica se due array sono uguali.
*/
public class TestUguaglianzaArray {

   public static void main(String[] args) {
      int[] a = new int[3];
      int[] b = new int[3];
      setArray(a);
      setArray(b);
```

Gli array a e b contengono gli stessi interi nel medesimo ordine.

```
        if (b == a)
            System.out.println("Uguali secondo l'operatore ==.");
        else
            System.out.println("Diversi secondo l'operatore ==.");

        if (equals(b, a))
            System.out.println("Uguali secondo il metodo equals.");
        else
            System.out.println("Diversi secondo il metodo equals.");
    }

    public static boolean equals(int[] a, int[] b) {
        boolean elementiUguali = true;  //si ipotizza che gli array siano uguali
        if (a.length != b.length)
            elementiUguali = false;
        else {
            int i = 0;
            while (elementiUguali && (i < a.length)) {
                if (a[i] != b[i])
                    elementiUguali = false;
                i++;
            }
        }
        return elementiUguali;
    }

    public static void setArray(int[] array) {
        for (int i = 0; i < array.length; i++)
            array[i] = i;
    }
}
```

Esempio di output

```
Diversi secondo l'operatore ==.
Uguali secondo il metodo equals.
```

Array e reference

Una variabile di tipo array contiene solo l'indirizzo in cui l'array è immagazzinato in memoria. Questo indirizzo di memoria è detto riferimento (*reference*) all'oggetto array in memoria. Per tale motivo le variabili di tipo array sono dette di tipo riferimento (*reference type*). Nel Capitolo 8 si vedrà che un tipo riferimento è un qualsiasi tipo le cui variabili contengono indirizzi di memoria anziché i valori degli elementi che riferiscono. Array e classi sono tipi riferimento. I tipi primitivi non lo sono.

FAQ Gli array sono realmente oggetti?

Gli array non appartengono ad alcuna classe. Altre caratteristiche proprie degli oggetti di una classe (come l'ereditarietà discussa nel Capitolo 10) non si applicano agli array. Non è quindi del tutto chiaro se considerare o no gli array come oggetti. Questo è fondamentalmente un dibattito teorico. In Java, gli array sono considerati oggetti. Anche la documentazione ufficiale di Java dice che ciò vale per gli oggetti vale anche per gli array.

6.2.5 Metodi che restituiscono array

Un metodo Java può restituire un array. Per fare ciò, specifica il tipo restituito dal metodo allo stesso modo con cui si specifica un parametro di tipo array. Per esempio, il Listato 6.5 contiene una versione riveduta del programma presentato nel Listato 6.3. Entrambi i programmi eseguono pressoché gli stessi calcoli, ma la nuova versione calcola i possibili punteggi medi all'interno del metodo `ottieniArrayDiMedie`. Questo nuovo metodo restituisce questi punteggi medi in un array. Per fare ciò, crea un nuovo array e lo restituisce con i seguenti passi:

```
double[] temp = new double[punteggioSeguente.length];
<Si riempie l'array temp.>
return temp;
```

MyLab

LISTATO 6.5 Metodo che restituisce un array.

```
import java.util.Scanner;

/**
Esempio di metodo che restituisce un array.
*/
public class RitornoDiArrayDemo {

    public static void main(String[] args) {
        Scanner tastiera = new Scanner(System.in);
        System.out.println("Inserire il voto dell'esame 1:");
        int punteggioIniziale = tastiera.nextInt();
        int[] punteggioSeguente = new int[3];

        for (int i = 0; i < punteggioSeguente.length; i++)
            punteggioSeguente[i] = punteggioIniziale + 5 * i;

        double[] punteggioMedio =
                    ottieniArrayDiMedie(punteggioIniziale, punteggioSeguente);

        for (int i = 0; i < punteggioSeguente.length; i++) {
            System.out.println("Se il voto all'esame 2 sara' " +
                               punteggioSeguente[i]);
            System.out.println("la media sara' uguale a " +
                               punteggioMedio[i]);
        }
    }
```

```
    public static double[] ottieniArrayDiMedie(int punteggioIniziale,
                                               int[] punteggioSeguente) {
        double[] temp = new double[punteggioSeguente.length];

        for (int i = 0; i < temp.length; i++)
            temp[i] = getMedia(punteggioIniziale, punteggioSeguente[i]);

        return temp;
    }

    public static double getMedia(int n1, int n2) {
        return (n1 + n2) / 2.0;
    }
}
```

L'output dell'esempio proposto è lo stesso del Listato 6.3.

Restituire un array

Un metodo può restituire un array nello stesso modo in cui può restituire un valore di un altro tipo.

Sintassi

```
public static tipo_base[] nome_metodo(lista_parametri){
    tipo_base[] temp = new tipo_base[dimensione_array]
    istruzioni_per_riempire_array
    return temp;
}
```

Esempio

```
public static char[] getVocali() {
    char[] nuovoArray = {'a', 'e', 'i', 'o', 'u'};
    return nuovoArray;
}
```

Nome del tipo base di un array

Il nome del tipo base di un array è sempre nella forma:

tipo_base[]

Questo è vero quando si dichiara una variabile array, quando si specifica il tipo base di un array usato come parametro o quando si indica che un metodo restituisce un array.

Esempi

```
int[] contatore = new int[10];
public static double[] riduci(int[] arrayDaRidurre) {
    ...
```

6.2.6 Metodi con un numero variabile di parametri

Come sarà approfondito nel Capitolo 9, è possibile definire all'interno della stessa classe più metodi con nome uguale, ma con lista di parametri formali diversa. Di conseguenza, è possibile per esempio avere, in una classe, un metodo massimo che restituisce il più grande tra due valori di tipo `int` e un altro metodo con lo stesso nome che richiede tre argomenti di tipo `int` e restituisce il più grande dei tre valori. Se fosse necessario un metodo che determini il massimo tra quattro numeri interi, si potrebbe definire un'altra versione del metodo massimo che accetti quattro argomenti. Tuttavia, seguendo questo approccio non si possono trattare tutti i possibili casi nei quali si deve determinare il massimo in un insieme di numeri interi, dato che occorrerebbero infinite definizioni del metodo massimo. Quella che serve è una singola definizione del metodo massimo che accetti un numero qualunque di argomenti di tipo `int`. A partire dalla versione 5, Java consente di definire metodi che accettano un numero variabile di argomenti. Per esempio, quella che segue è una definizione di un metodo massimo che accetta un numero qualunque di argomenti di tipo `int` e restituisce il più grande tra essi:

```
public static int massimo(int... arg) {
    if (arg.length == 0) {
        System.out.println("Errore: nessun valore specificato.");
        System.exit(0);
    }

    int m = arg[0];
    for (int i = 1; i < arg.length; i++)
        if (arg[i] > m)
            m = arg[i];
    return m;
}
```

Questo metodo prende gli argomenti e li organizza in un array chiamato `arg` con tipo base `int`. Per esempio, si consideri la seguente chiamata:

```
int punteggioPiuAlto = massimo(3, 2, 5, 1);
```

L'array `arg` viene dichiarato e inizializzato automaticamente nel modo seguente:

```
int[] arg = {3, 2, 5, 1};
```

Quindi, `arg[0] == 3`, `arg[1] == 2`, `arg[2] == 5` e `arg[3] == 1`. A questo punto, viene eseguito il codice nel corpo del metodo. Il Listato 6.6 mostra un esempio di programma che utilizza questo metodo `massimo`.

MyLab

LISTATO 6.6 Un metodo con un numero variabile di parametri.

```
import java.util.Scanner;

public class EsempioNumeroVariabileArgomenti {
    /**
    Restituisce il massimo tra un numero qualunque di interi.
    */
    public static int massimo(int... arg) {
```

```
        if (arg.length == 0) {
            System.out.println("Errore: nessun valore specificato.");
            System.exit(0);
        }

        int m = arg[0];
        for (int i = 1; i < arg.length; i++)
           if (arg[i] > m)
                m = arg[i];
        return m;
    }

    public static void main(String[] args) {
        System.out.println("Inserire i punteggi di Anna, Marco e Luca:");
        Scanner tastiera = new Scanner(System.in);
        int punteggioAnna = tastiera.nextInt();
        int punteggioMarco = tastiera.nextInt();
        int punteggioLuca = tastiera.nextInt();
        int punteggioPiuAlto = massimo(punteggioAnna, punteggioMarco,
                                                      punteggioLuca);
        System.out.println("Punteggio più alto = " + punteggioPiuAlto);
    }
}
```

Esempio di output

```
Inserire i punteggi di Anna, Marco e Luca:
55 100 99
Punteggio più alto = 100
```

Si noti che un metodo (come `massimo`) che accetta un numero variabile di argomenti è di fatto un metodo che accetta come argomento un array, ad eccezione del fatto che il lavoro di inserire gli elementi nell'array è svolto automaticamente senza che se ne debba preoccupare il programmatore. I valori vengono semplicemente passati come argomenti e Java crea automaticamente l'array e vi inserisce gli elementi.

Una specifica di parametro relativa a un numero variabile di parametri, come `int... arg`, è detta specifica **vararg** (sarebbe stato più corretto chiamarla specifica *varparameter*, ma il termine *vararg* è ormai di uso comune, quindi ci si atterrà a questa denominazione). I puntini nella specifica sono chiamati **ellissi**. Si noti che l'ellissi è a tutti gli effetti parte della sintassi Java e non un'abbreviazione utilizzata in questo libro.

Nella definizione di un metodo si può avere una sola specifica di un numero variabile di parametri. Tuttavia, è possibile avere, oltre a questa, anche le specifiche di un qualunque numero di parametri ordinari. In tal caso, la specifica di numero variabile deve essere l'ultima della lista, come mostrato nel Listato 6.6.

Un esempio di metodo che accetta un numero variabile di parametri è stato già incontrato nel Capitolo 2: si tratta del metodo `System.out.printf`. Tuttavia, per poter presentare le modalità di definizione di questi metodi è stato necessario attendere di aver trattato le basi degli array.

Metodi con un numero variabile di parametri

Un metodo con un numero variabile di parametri ha una specifica di tipo *vararg* come ultimo elemento della lista dei parametri. Una specifica *vararg* ha la forma seguente:

tipo... nome_array

Esempi di questo tipo di specifica sono

```
int... arg
double... a
String... indesiderate
```

I Listati 6.6 e 6.7 mostrano due esempi all'interno di definizione complete di metodi.

In ogni invocazione di un metodo con un numero variabile di parametri, si gestiscono per prima cosa e nel solito modo gli argomenti corrispondenti ai parametri ordinari. A seguire, si può inserire un numero qualunque di argomenti del tipo indicato nella specifica *vararg*. Questi argomenti verranno inseriti automaticamente nell'array indicato nella specifica.

ESEMPIO DI PROGRAMMAZIONE
UN ESEMPIO DI ELABORAZIONE DI STRINGHE

Questo esempio si basa sul contenuto della sezione "Metodi con un numero variabile di parametri". Il Listato 6.7 contiene un metodo di elaborazione delle stringhe chiamato `censura` e un esempio di programma che lo utilizza. Il metodo `censura` accetta un parametro di tipo `String` seguito da un numero qualunque di parametri aggiuntivi, anch'essi di tipo `String`. Il primo parametro conterrà una frase che potrebbe includere parole o stringhe che si vogliono eliminare. Il metodo restituisce il primo argomento dal quale sono state eliminate tutte le occorrenze degli altri argomenti.

Si noti che il metodo `censura` ha un parametro ordinario seguito dalla specifica di un numero qualunque di parametri di tipo stringa aggiuntivi. In questo caso, tutti i parametri sono di tipo `String`. Tuttavia, i parametri ordinari all'inizio della lista dei parametri di un metodo possono essere di qualunque tipo: non devono necessariamente essere dello stipo di quelli presenti in numero variabile.

Poiché qui sia l'argomento ordinario che quelli in numero variabile sono di tipo `String`, ci si potrebbe chiedere perché il primo parametro non sia stato omesso, lasciando solo la specifica di numero variabile e utilizzando quindi `indesiderate[0]` nel ruolo di `frase`. Se il metodo fosse stato definito in questo modo, potrebbe essere chiamato anche senza alcun argomento, dato che una specifica *vararg* permette qualunque numero di argomenti, compreso lo zero. La presenza del parametro `frase` garantisce che al metodo venga sempre passato almeno un argomento.

LISTATO 6.7 Un metodo con un numero variabile di parametri per l'elaborazione di stringhe.

MyLab

```
import java.util.Scanner;

public class EsempioNumeroVariabileArgomenti2 {
    /**
    Restituisce il primo argomento con tutte le occorrenze degli altri
    argomenti cancellate.
    */
    public static String censura(String frase, String... indesiderate) {
        for (int i = 0; i < indesiderate.length; i++)
            frase = cancellaStringa(frase, indesiderate[i]);
        return frase;
    }

    /**
    Restituisce frase con tutte le occorrenze di stringa rimosse.
    */
    public static String cancellaStringa(String frase, String stringa) {
        String finale;
        int posizione = frase.indexOf(stringa);
        while (posizione >= 0) { //Finché compare la stringa
            finale = frase.substring(posizione + stringa.length());
            frase = frase.substring(0, posizione) + finale;
            posizione = frase.indexOf(stringa);
        }
        return frase;
    }

    public static void main(String[] args) {
        System.out.println("Cos'hai mangiato per cena?");
        Scanner tastiera = new Scanner(System.in);
        String frase = tastiera.nextLine();
        frase = censura(frase, "caramelle", "patatine fritte",
                                    "salato", "birra");
        frase = censura(frase, " ,"); //Cancella le virgole in più
        System.out.println("Saresti più sano se avessi risposto:");
        System.out.println(frase);
    }
}
```

Esempio di output

```
Cos'hai mangiato per cena?
Ho mangiato merluzzo salato, broccoli, patatine fritte, e mele.
Saresti più sano se avessi risposto:
Ho mangiato merluzzo, broccoli, e mele.
```

6.3 Ordinamento e ricerca con gli array

Si supponga di avere un array di valori. Si potrebbe avere l'esigenza di ordinare in qualche modo questi valori. Per esempio, si potrebbe voler ordinare un array di numeri dal più piccolo al più grande (o viceversa) o magari si potrebbe volere organizzare un array di stringhe secondo l'ordine alfabetico. Organizzare un insieme di elementi secondo un particolare ordine viene chiamato **ordinamento**. Tipicamente gli array si ordinano in senso crescente o decrescente.

In questo paragrafo verrà dicusso e implementato un semplice algoritmo di ordinamento. Questo algoritmo sarà presentato come un modo per ordinare un array di interi. Tuttavia, con piccoli aggiustamenti, potrebbe essere adattato per ordinare array di valori di qualunque tipo. Per esempio, si potrebbe ordinare un array di dipendenti secondo il loro codice identificativo.

Si prenderà in esame anche la ricerca di un determinato elemento all'interno di un array. È possibile effettuare una ricerca all'interno di array ordinati o anche in array completamenti non organizzati. Sia l'ordinamento sia la ricerca sono fattori molto importanti, quindi è fondamentale impiegare algoritmi efficienti. Questa parte del capitolo fornirà solo una breve introduzione all'argomento.

6.3.1 Selection Sort

Si immagini un array a di interi che si vuole ordinare in senso crescente. Questo significa che occorre riorganizzare i valori contenuti nelle variabili indicizzate dell'array in modo che:

```
a[0] ≤ a[1] ≤ a[2] ≤ … ≤ a[a.length-1]
```

Verrà discusso uno dei più semplici algoritmi di ordinamento, il **selection sort**. Applicando questo algoritmo, i valori dell'array a saranno riorganizzati in modo che `a[0]` sia il più piccolo, `a[1]` il secondo più piccolo e così via. Questa richiesta porta al seguente pseudocodice:

```
for (indice = 0; indice < a.length; indice++)
    Posiziona il (indice + 1)-esimo più piccolo elemento in a[indice]
```

Si vuole che questo algoritmo operi direttamente sull'array a. Quindi, l'unico modo che si ha per muovere un elemento dell'array senza toccare gli altri è quello di scambiare la posizione degli elementi dell'array. Ogni algoritmo di ordinamento che scambia gli elementi, è chiamato algoritmo di **ordinamento basato su scambi** *(interchanges sorting algorithm)*. Di conseguenza, il *selection sort* è un algoritmo di ordinamento basato su scambi.

Si inizierà con un esempio per illustrare come gli elementi dell'array vengono scambiati di posizione. A tal proposito, la Figura 6.4 mostra come un array viene ordinato scambiando i suoi valori. Partendo da un array di valori non ordinato, si identifica al suo interno il valore più piccolo. In questo esempio, tale valore è 3 e si trova nella variabile indicizzata `a[4]`. Dato che si vuole che tale valore sia il primo dell'array, l'elemento in `a[4]` viene scambiato con l'elemento in `a[0]`. Dopo lo scambio il valore più piccolo sarà in `a[0]`.

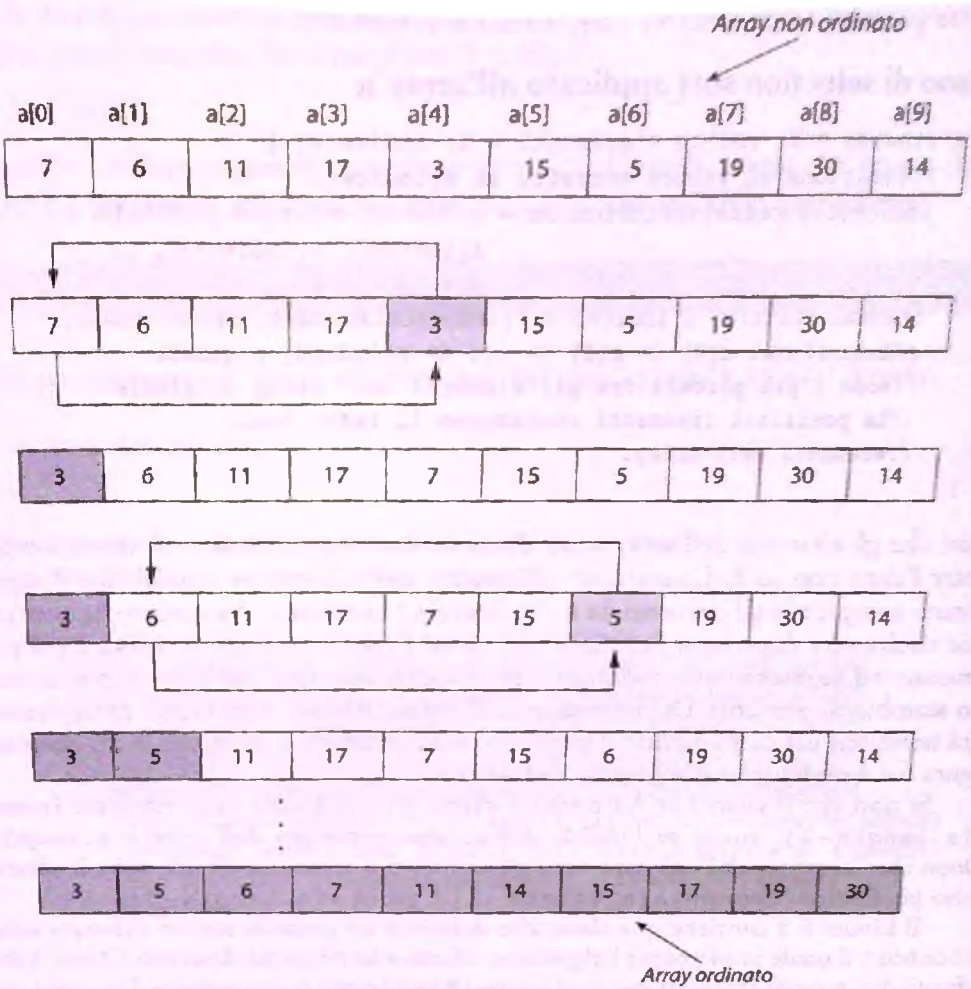


Figura 6.4 Selection Sort.

Il successivo valore più piccolo è 5 e si trova nella variabile indicizzata `a[6]`. Esso deve essere posto nella seconda posizione dell'array. Viene, quindi, scambiato il valore in `a[6]` con quello in `a[1]`. Dopo lo scambio, i valori in `a[0]` e in `a[1]` sono rispettivamente quello più piccolo e il secondo più piccolo, così come dovrebbero trovarsi nell'array ordinato. L'algoritmo procede scambiando il successivo valore più piccolo con quello in posizione `a[2]` e continua finché l'intero array non è ordinato.

Come è possibile trovare il secondo elemento più piccolo e poi il terzo e così via? Dopo aver trovato il valore più piccolo tra `a[0]`, `a[1]`, ..., `a[n]` e averlo assegnato ad `a[0]`, il secondo valore più piccolo dell'array a è il valore più piccolo tra `a[1]`, ..., `a[n]`. Dopo aver assegnato tale valore ad `a[1]`, il terzo valore più piccolo in a è il più piccolo valore tra `a[2]`, ..., `a[n]` e così via.

Il seguente pseudocodice descrive l'algoritmo *selection sort*.

Algoritmo di selection sort applicato all'array a

```
for (indice = 0; indice < a.length - 1; indice++) {
    //Posiziona il valore corretto in a[indice]:
    indiceDelSuccessivoPiuPiccolo = indice del valore più piccolo tra
                                    a[indice], a[indice+1], …,
                                    a[a.length-1]
    Scambia i valori in a[indice] e a[indiceDelSuccessivoPiuPiccolo].
    //Asserzione: a[0] <= a[1] <= ... <= a[indice] e questi
    //sono i più piccoli fra gli elementi dell'array originale.
    //Le posizioni rimanenti contengono il resto degli
    //elementi dell'array.
}
```

Si noti che gli elementi dell'array sono divisi in due segmenti: uno di questi è ordinato, mentre l'altro non lo è. L'asserzione all'interno dell'algoritmo implica che il segmento ordinato comprende gli elementi da `a[0]` fino a `a[indice]`. Va notato che questa asserzione risulta vera dopo ogni iterazione del ciclo. Ripetitivamente si ricerca il più piccolo elemento nel segmento non ordinato e lo si sposta alla fine del segmento ordinato con uno scambio di elementi. Di conseguenza, il segmento ordinato cresce di un elemento a ogni iterazione del ciclo, mentre il segmento non ordinato si restringe di un elemento. In Figura 6.4 è evidenziato il segmento ordinato.

Si noti che il ciclo `for` ha termine dopo aver collocato correttamente l'elemento `a[a.length-2]`, anche se l'indice dell'ultimo elemento dell'array è `a.length-1`. Dopo che l'algoritmo ha ordinato tutti gli elementi a eccezione di uno solo, il valore corretto per l'ultimo elemento `a[a.length-1]` è già in `a[a.length-1]`.

Il Listato 6.8 contiene una classe che definisce un metodo statico chiamato `selectionSort` il quale implementa l'algoritmo *selection sort* appena descritto. Questo metodo sfrutta due metodi chiamati `getIndiceDelPiuPiccolo` e `scambio`. Una volta capito il funzionamento di questi due metodi, risulta evidente che la definizione del metodo `selectionSort` è una traduzione diretta dello pseudocodice in codice Java. Di seguito verrà discusso il funzionamento di questi due metodi.

Il metodo `getIndiceDelPiuPiccolo` ricerca tra gli elementi dell'array

```
a[indiceInizio], a[indiceInizio + 1], …, a[a.length - 1]
```

l'indice della variabile indicizzata che contiene il valore più piccolo e lo restituisce. Questo viene fatto utilizzando due variabili locali `minimo` e `indiceDelMinimo`. In ogni istante della ricerca, `minimo` è uguale al più piccolo valore dell'array trovato fino a quel momento e `indiceDelMinimo` è l'indice di quel valore. Quindi `a[indiceDelMinimo]` contiene il valore `minimo`.

Inizialmente `minimo` è impostato ad `a[indiceInizio]`, che è il primo valore considerato per `minimo`, e `indiceDelMinimo` è impostato a `indiceInizio`. In seguito, viene considerato ogni elemento per verificare se è il nuovo minimo. Se lo è, `minimo` e `indiceDelMinimo` vengono aggiornati. Dopo aver controllato tutti gli elementi candidati dell'array, il metodo restituisce il valore di `indiceDelMinimo`.

Il metodo `scambio` scambia i valori di `a[i]` e `a[j]`. Va fatta una considerazione sottile riguardo a questo metodo. Se si eseguisse il codice:

```
a[i] = a[j];
```

si perderebbe il valore originale contenuto in `a[i]`. Quindi, prima che questa istruzione sia eseguita, è necessario salvare il valore di `a[i]` in una variabile locale `temp`.

LISTATO 6.8 Implementazione del Selection Sort.

MyLab

```
/**
Classe per ordinare un array di tipo int dal piu' piccolo al piu' grande.
*/
public class OrdinaArray {

    /**
    Precondizione: Ogni elemento nell'array ha un valore.
    Azione: Ordina l'array in senso crescente.
    */
    public static void selectionSort(int[] unArray) {
        for (int indice = 0; indice < unArray.length - 1; indice++) {
            // Posiziona il valore corretto in unArray[indice]
            int indiceDelSuccessivoPiuPiccolo =
                getIndiceDelPiuPiccolo(indice, unArray);
            scambio(indice, indiceDelSuccessivoPiuPiccolo, unArray);
            //Asserzione: unArray[0] <= unArray[1] <= ... <= unArray[indice]
            //e questi sono i piu' piccoli dell'array originale di elementi.
            //Le posizioni rimanenti contengono i rimanenti elementi
            //dell'array.
        }
    }

    /**
    Restituisce l'indice del piu' piccolo valore nella porzione di
    array che inizia dall'elemento il cui indice e' indiceInizio e
    termina all'ultimo elemento.
    */
    public static int getIndiceDelPiuPiccolo(int indiceInizio, int[] a) {
        int minimo = a[indiceInizio];
        int indiceDelMinimo = indiceInizio;

        for (int indice = indiceInizio + 1; indice < a.length; indice++) {
            if (a[indice] < minimo) {
                minimo = a[indice];
                indiceDelMinimo = indice;
                //minimo e' il piu' piccolo
                //da a[indiceInizio] fino a[indice]
            }
        }
        return indiceDelMinimo;
    }
```

```
    /**
    Precondizione: i e j sono indici validi per l'array a.
    Postcondizione: i valori di a[i] e a[j] sono stati scambiati.
    */
    public static void scambio(int i, int j, int[] a) {
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp; //valore originale di a[i]
    }
}
```

Lab

LISTATO 6.9 Dimostrazione del metodo `selectionSort`.

```
public class SelectionSortDemo {

    public static void main(String[] args) {
        int[] b = {7, 5, 11, 2, 16, 4, 18, 14, 12, 30};
        visualizza(b, "prima dell'ordinamento");
        OrdinaArray.selectionSort(b);
        visualizza(b, "dopo l'ordinamento");
    }

    public static void visualizza(int[] array, String quando) {
        System.out.println("Valori dell'array " + quando + ":");

        for (int i = 0; i < array.length; i++)
            System.out.print(array[i] + " ");

        System.out.println();
    }
}
```

Esempio di output

```
Valori dell'array prima dell'ordinamento:
7 5 11 2 16 4 18 14 12 30
Valori dell'array dopo l'ordinamento:
2 4 5 7 11 12 14 16 18 30
```

Il Listato 6.9 contiene un programma dimostrativo che illustra il metodo *selection sort* in azione.

6.3.2 Altri algoritmi di ordinamento

Sebbene l'algoritmo *selection sort* sia sufficiente come introduzione generale all'argomento dell'ordinamento, non si può dire che sia l'algoritmo di ordinamento più efficiente. Infatti, è significativamente meno efficiente rispetto ad altri algoritmi di ordinamento ben noti. Il *selection sort* è, tuttavia, molto semplice. L'implementazione in codice Java di un algoritmo semplice è meno soggetta a errori. Di conseguenza, se occorre codificare al volo un algoritmo di ordinamento, è molto più sicuro utilizzare un *selection sort* o un altro algoritmo altrettanto semplice.

D'altra parte, quando l'efficienza è una caratteristica importante, è più opportuno utilizzare un algoritmo più complesso ma anche più efficiente. Ma più l'algoritmo è complesso e più saranno lunghe le fasi di codifica, collaudo e debugging. L'efficienza è un argomento molto delicato. Ottenere un risultato rapidissimo ma sbagliato è, per definizione, sempre inefficiente. Fortunatamente la Java Class Library fornisce algoritmi di ordinamento efficienti. La classe `Arrays` del package `java.util`, definisce il metodo statico `sort`. Dato un`Array`, un array di valori primitivi o oggetti, l'istruzione:

```
Arrays.sort(unArray);
```

ordina gli elementi dell'intero array in senso crescente. Per ordinare la sola porzione di array compresa fra l'indice `inizio` e l'indice `fine`, basta scrivere:

```
Arrays.sort(unArray, inizio, fine);
```

La classe `Arrays` fornisce diverse versioni del metodo per gestire sia array di classi sia array di tutti i tipi primitivi. I Progetti 4 e 5 alla fine di questo capitolo descrivono altri due algoritmi di ordinamento.

6.3.3 Ricerca negli array

Se è necessario effettuare la ricerca di un elemento all'interno di un array, si può utilizzare la cosiddetta **ricerca sequenziale** (*sequential research*) di un array. L'algoritmo di ricerca sequenziale è molto semplice e lineare: si guardano gli elementi dell'array dal primo all'ultimo per vedere se l'elemento richiesto è uguale a qualche elemento presente nell'array. La ricerca termina quando nell'array viene trovato l'elemento desiderato o quando viene raggiunta la fine dell'array senza aver trovato l'elemento. Se l'array è parzialmente riempito, la ricerca considera solo la porzione di array che contiene valori significativi.

Una ricerca sequenziale può essere applicata a un array non ordinato, il che risulta un vantaggio di questo semplice algoritmo. Se, tuttavia, l'array è ordinato, è possibile migliorare la ricerca sequenziale nelle situazioni in cui l'elemento desiderato non sia presente nell'array. L'Esercizio 12 chiede di programmare questa ricerca sequenziale migliorata.

6.4 Array multidimensionali

In alcune situazioni può essere utile disporre di un array con più indici. Per esempio, si supponga di voler memorizzare in un qualche tipo di array gli importi in Euro mostrati in Figura 6.5. La parte di tabella evidenziata contiene 60 elementi di quel tipo. Se si utilizzasse un array con un solo indice, la lunghezza di tale array sarebbe 60. Mantenere traccia delle associazioni fra elementi e indici potrebbe risultare difficile. D'altra parte, se si utilizzassero due indici, si potrebbe usare un indice per le righe e un indice per le colonne della tabella. Un array di questo tipo è mostrato nella Figura 6.6.

Array che hanno esattamente due indici possono essere visualizzati su di un foglio come tabelle bidimensionali e vengono chiamati **array bidimensionali** (*two-dimensional arrays*). Per convenzione, si attribuisce al primo indice la numerazione delle righe della tabella e al secondo la numerazione delle colonne. Si noti che, come detto per gli array visti in precedenza, la numerazione degli indici parte da 0 e non da 1. La notazione Java per indicare un elemento di un array bidimensionale è:

nome_array[*indice_riga*][*indice_colonna*]

Risparmio nel saldo dei conti per diversi tassi di interesse a capitalizzazione annuale (arrotondato a valori interi di Euro)						
Anno	**5.00%**	**5.50%**	**6.00%**	**6.50%**	**7.00%**	**7.50%**
1	€1050	€1055	€1060	€1065	€1070	€1075
2	€1103	€1113	€1124	€1134	€1145	€1156
3	€1158	€1174	€1191	€1208	€1225	€1242
4	€1216	€1239	€1262	€1286	€1311	€1335
5	€1276	€1307	€1338	€1370	€1403	€1436
6	€1340	€1379	€1419	€1459	€1501	€1543
7	€1407	€1455	€1504	€1554	€1606	€1659
8	€1477	€1535	€1594	€1655	€1718	€1783
9	€1551	€1619	€1689	€1763	€1838	€1917
10	€1629	€1708	€1791	€1877	€1967	€2061

Figura 6.5 Una tabella di valori.

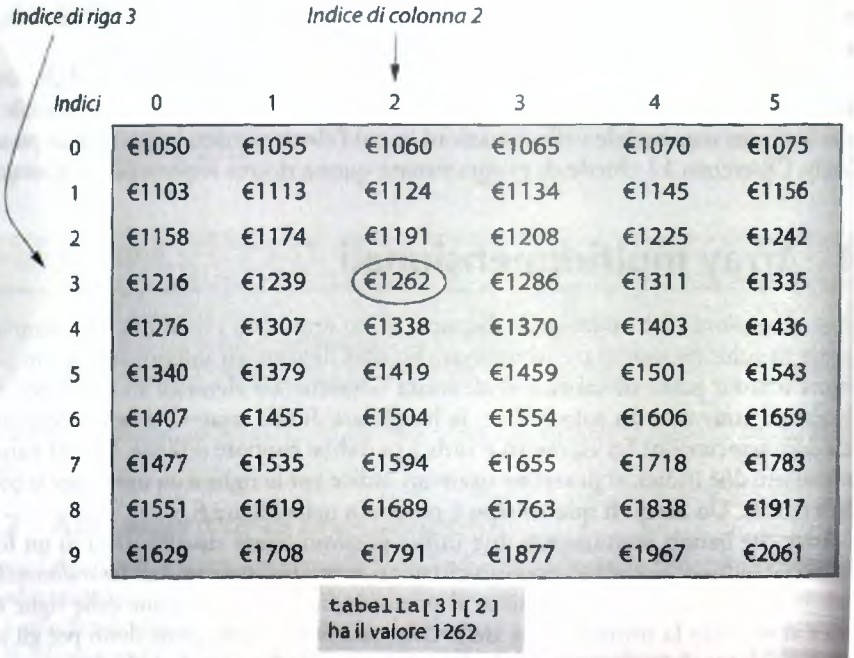


Figura 6.6 Indici di riga e colonna per un array chiamato `tabella`.

Per esempio, dato l'array chiamato `tabella` gestito da due indici, `tabella[3][2]` è l'elemento il cui indice di riga è 3 e l'indice di colonna è 2. Dato che gli indici partono da 0, questo elemento si trova nella *quarta* riga e nella *terza* colonna di `tabella`, come illustrato in Figura 6.6. Cercare di correlare gli indici dell'array con i numeri reali di riga e colonna potrebbe creare confusione e non è necessario.

Gli array con più indici vengono generalmente chiamati **array multidimensionali** (*multidimensional arrays*). In particolare, un array con *n* indici viene detto **array n-dimensionale** (*n-dimensional array*). Quindi, un semplice array con un indice è un **array monodimensionale** (*one-dimensional array*). Sebbene array con più di due dimensioni siano rari, in qualche applicazione possono risultare utili.

6.4.1 Fondamenti sugli array multidimensionali

Gli array a più indici sono trattati nello stesso modo degli array monodimensionali. Per illustrare i dettagli viene mostrato un esempio di programma Java che visualizza l'array di Figura 6.6. Il programma è mostrato nel Listato 6.10. L'array è chiamato `tabella`. Le seguenti istruzioni dichiarano che il nome dell'array è `tabella` e poi lo creano:

```
int[][] tabella = new int[10][6];
```

Questo è equivalente alle due istruzioni:

```
int[][] tabella;
tabella = new int[10][6];
```

Si noti che la sintassi risulta essere molto simile a quella utilizzata nel caso di array monodimensionale. L'unica differenza è l'aggiunta di una coppia di parentesi quadre e il fatto di utilizzare un secondo numero per specificare la seconda dimensione, corrispondente alle colonne. Si possono creare array con un numero arbitrario di indici. Per aggiungere un indice basta aggiungere nella dichiarazione una nuova coppia di parentesi quadre.

Le variabili indicizzate negli array multidimensionali sono del tutto identiche a quelle negli array monodimensionali, tranne per il fatto che esse hanno più indici, ognuno racchiuso in una coppia di parentesi quadre. Tutto questo è illustrato nel seguente ciclo `for`, estratto dal Listato 6.10:

```
for (int riga = 0; riga < 10; riga++)
    for (int colonna = 0; colonna < 6; colonna++)
        tabella[riga][colonna] =
            getBilancio(1000.00, riga + 1, (5 + 0.5 * colonna));
```

Si noti che sono utilizzati due cicli `for`, uno innestato nell'altro. Questo corrisponde al modo comune di scandire gli elementi indicizzati di un array bidimensionale. Se si avessero tre indici, si dovrebbero utilizzare tre cicli `for` innestati e così via. L'immagine nella Figura 6.6 aiuta a capire meglio il significato degli indici in `tabella[riga][colonna]` e il significato dei cicli `for` innestati.

Come le variabili indicizzate degli array monodimensionali, anche quelle degli array multidimensionali sono variabili di un tipo base specificato e possono essere utilizzate ovunque sia consentita una variabile di quel tipo base. Per esempio, la variabile indicizzata `tabella[3][2]` nell'array nel Listato 6.10 è una variabile di tipo `int` e può essere impiegata ovunque possa essere utilizzata una variabile ordinaria di tipo `int`.

Dichiarazione e creazione di un array multidimensionale

È possibile dichiarare il nome di un array multidimensionale e poi creare l'array nello stesso modo in cui si dichiara e poi crea un array monodimensionale. Basta utilizzare tante coppie di parentesi quadre quanti sono gli indici.

Sintassi

```
tipo_base[]...[] nome_array = new tipo_base[lunghezza_1]...[lunghezza_n];
```

Esempi

```
char[][] pagine = new char[100][80];
int[][] tabella = new int[10][6];
double[][][] immagineTreD = new double[10][20][30];
```

Lab

LISTATO 6.10 Utilizzo di un array bidimensionale.

```
/**
Visualizza una tabella bidimensionale mostrando come
i tassi di interesse influiscono sui bilanci delle banche.
*/
public class TabellaInteressi {

  public static void main(String[] args) {
    int[][] tabella = new int[10][6];

    for (int riga = 0; riga < 10; riga++)
      for (int colonna = 0; colonna < 6; colonna++)
        tabella[riga][colonna] =
          getBilancio(1000.00, riga + 1, (5 + 0.5 * colonna));

    System.out.println("Bilanci per vari Tassi di Interesse " +
                       " Capitalizzazione annuale");
    System.out.println("(Arrotondato a valori interi di Euro)");
    System.out.println();
    System.out.println("Anni 5.00% 5.50% 6.00% 6.50% 7.00% 7.50%");

    for (int riga = 0; riga < 10; riga++) {
      System.out.print((riga + 1) + "     ");

      for (int colonna = 0; colonna < 6; colonna++)
        System.out.print("€" + tabella[riga][colonna] + " ");

      System.out.println();
    }
```

Un'applicazione reale dovrebbe fare molte più cose con l'array `tabella`. Questo e' solo un programma dimostrativo.

```
    /**
    Restituisce il bilancio in conto dopo un dato numero di anni
    e il tasso di interesse con un bilancio iniziale di bilancioIniziale.
    L'interesse e' calcolato annualmente. Il bilancio e' arrotondato
    a un numero intero.
    */
    public static int getBilancio(double bilancioIniziale, int anni,
                                  double tasso) {
        double bilancioCorrente = bilancioIniziale;

        for (int conteggio = 1; conteggio <= anni; conteggio++)
            bilancioCorrente = bilancioCorrente * (1 + tasso / 100);

        return (int)(Math.round(bilancioCorrente));
    }
}
```

Esempio di output

```
Bilanci per vari Tassi di Interesse Capitalizzazione annuale
(Arrotondato a valori interi di Euro)

Anni 5.00% 5.50% 6.00% 6.50% 7.00% 7.50%
1    €1050 €1055 €1060 €1065 €1070 €1075
2    €1103 €1113 €1124 €1134 €1145 €1156
3    €1158 €1174 €1191 €1208 €1225 €1242
4    €1216 €1239 €1262 €1286 €1311 €1335
5    €1276 €1307 €1338 €1370 €1403 €1436
6    €1340 €1379 €1419 €1459 €1501 €1543
7    €1407 €1455 €1504 €1554 €1606 €1659
8    €1477 €1535 €1594 €1655 €1718 €1783
9    €1551 €1619 €1689 €1763 €1838 €1917
10   €1629 €1708 €1791 €1877 €1967 €2061
```

L'ultima riga risulta fuori allineamento perché 10 è formato da due caratteri. Questo è un problema facile da sistemare, ma complicherebbe la discussione degli array con l'introduzione di ulteriori concetti.

6.4.2 Array multidimensionali come parametri e come valori restituiti

I metodi possono usare array multidimensionali come parametri e valori restituiti. Questa situazione è simile a quella degli array monodimensionali, tranne per il fatto che si usano più parentesi quadre. Per esempio, il programma presentato nel Listato 6.11 è simile a quello del Listato 6.10 tranne per il fatto che ha un metodo per visualizzare un array bidimensionale passato come argomento. Si noti che il tipo per il parametro dell'array è `int[][]`. Si sono inoltre utilizzate costanti invece di letterali per definire i numeri di riga e colonna.

LISTATO 6.11 Array multidimensionale come parametro.

```
/**
Visualizza una tabella bidimensionale mostrando come
i tassi di interesse influiscono sui bilanci delle banche.
*/
public class TabellaInteressi2 {

    public static final int RIGHE = 10;
    public static final int COLONNE = 6;

    public static void main(String[] args) {
        int[][] tabella = new int[RIGHE][COLONNE];

        for (int riga = 0; riga < RIGHE; riga++)
            for (int colonna = 0; colonna < COLONNE; colonna++)
                tabella[riga][colonna] =
                    getBilancio(1000.00, riga + 1, (5 + 0.5 * colonna));

        System.out.println("Bilanci per vari Tassi di Interesse " +
                           " Capitalizzazione annuale ");
        System.out.println("(Arrotondato a valori interi di Euro)");
        System.out.println();
        System.out.println("Anni 5.00% 5.50% 6.00% 6.50% 7.00% 7.50%");
        visualizzaTabella(tabella);
    }

    /**
    Precondizione: L'array unArray ha RIGHE righe e COLONNE colonne.
    Postcondizione: Il contenuto dell'array e' visualizzato
    con il segno della valuta.
    */
    public static void visualizzaTabella(int[][] unArray) {

        for (int riga = 0; riga < RIGHE; riga++) {
            System.out.print((riga + 1) + "    ");

            for (int colonna = 0; colonna < COLONNE; colonna++)
                System.out.print("€" + unArray[riga][colonna] + " ");

            System.out.println();
        }
    }

    public static int getBilancio(double bilancioIniziale, int anni, double tasso)
    <La definizione di getBilancio è identica a quella nel Listato 6.10>
}
```

È possibile fornire una definizione migliore di `visualizzaTabella`.

L'output è lo stesso del Listato 6.10

Per restituire un array multidimensionale si può utilizzare lo stesso tipo di specifica che si utilizza per i parametri di tipo array monodimensionale. Per esempio, il seguente metodo restituisce un array bidimensionale il cui tipo base è `double`:

```
/**
Precondizione: Ogni dimensione di arrayIniziale ha
valore di dimensione.
Postcondizione: L'array restituito è una copia
dell'array arrayIniziale.
*/
public static double[][] copia(double[][] arrayIniziale,
                               int dimensione) {
    double[][] temp = new double[dimensione][dimensione];
    for (int riga = 0; riga < dimensione; riga++)
        for (int colonna = 0; colonna < dimensione; colonna++)
            temp[riga][colonna] = arrayIniziale[riga][colonna];
    return temp;
}
```

Array multidimensionali come parametri e come valori restituiti

Un parametro o un tipo di ritorno di un metodo può essere un array multidimensionale. La sintassi per l'intestazione del metodo è simile al caso in cui parametri o tipi restituiti siano array monodimensionali, ma bisogna utilizzare più parentesi quadre `[]`.

Sintassi

```
public static tipo_di_ritorno nome_metodo(tipo_base[]...[] nome_parametro)
```

o

```
public static tipo_base[]...[] nome_metodo(lista_parametri)
```

Esempi

```
public static int getUnElemento(char[][] a, int riga, int colonna)
public static void leggiArray(int[][] unArray)
public static char[][] copia(char[][] array)
public static int[][] getArray()
```

6.4.3 Rappresentazione Java di array multidimensionali

Il compilatore Java rappresenta un array multidimensionale come più array monodimensionali. Per esempio, si consideri il seguente array bidimensionale.

```
int[][] tabella = new int[10][6];
```

L'array `tabella` è, infatti, un array monodimensionale di lunghezza 10 e il suo tipo base è `int[]`. In altre parole, gli array multidimensionali sono array di array. Normalmente, non è necessario preoccuparsi di questi dettagli, dal momento che tutto viene gestito automaticamente dal compilatore. Tuttavia è sempre utile conoscere il modo in cui Java memorizza gli array multidimensionali.

Per esempio, si supponga di voler riempire un array bidimensionale con una serie di valori mediante un ciclo `for`. Il programma del Listato 6.11 controlla i cicli `for` tramite le costanti RIGHE e COLONNE. Un modo più generico per controllare i cicli `for` consiste nell'utilizzare la variabile di istanza `length`. Quando si utilizza `length` con un array multidimensionale è però necessario pensare in termini di array di array.

Il seguente codice è una revisione del ciclo `for` innestato che appare nel metodo `main` del Listato 6.11:

```
for (int riga = 0; riga < tabella.length; riga++)
    for (int colonna = 0; colonna < tabella[riga].length; colonna++)
        tabella[riga][colonna] =
            getBilancio(1000.00, riga + 1, (5 + 0.5 * colonna));
```

In considerazione del fatto che l'array `tabella` è a tutti gli effetti un array monodimensionale di lunghezza 10, il primo ciclo `for` può utilizzare `tabella.length` invece di `RIGHE`, che vale 10. In aggiunta, ciascuna delle 10 variabili indicizzate (da `tabella[0]` a `tabella[9]`) è a sua volta un array monodimensionale la cui lunghezza è 6 (`COLONNE`). Quindi, dato che `tabella[riga]` è un array monodimensionale la cui lunghezza è 6, il secondo ciclo `for` può utilizzare `tabella[riga].length` anziché `COLONNE`.

La variabile di istanza `length` per un array bidimensionale

Per un array bidimensionale `b`, il valore di `b.length` è il numero di righe, in altre parole l'intero nella prima coppia di parentesi quadre nella dichiarazione dell'array. Il valore di `b[i].length` è il numero di colonne, cioè l'intero nella seconda coppia di parentesi quadre nella dichiarazione dell'array. Per l'array bidimensionale `tabella` del Listato 6.11, il valore di `tabella.length` è il numero di righe (in questo caso 10) e il valore di `tabella[riga].length` è il numero di colonne (in questo caso 6).

MyLab

Video 6.1 Usare array bidimensionali

È possibile sfruttare il fatto che gli array multidimensionali sono array di array per riscrivere il metodo `visualizzaTabella` del Listato 6.11. Si noti che nel Listato 6.11, il metodo `visualizzaTabella` presuppone che l'array passato come argomento abbia `RIGHE (10)` righe e `COLONNE (6)` colonne. Questo risulta vero per questo particolare programma, ma una migliore definizione di `visualizzaTabella` dovrebbe funzionare per qualsiasi array bidimensionale.

6.4.4 Array irregolari (opzionale)

Nei precedenti esempi di array bidimensionali, tutte le righe avevano lo stesso numero di elementi. Per esempio, si consideri l'array bidimensionale creato come segue:

```
int[][] a = new int[3][5];
```

Questa istruzione è equivalente alle seguenti istruzioni:

```
int[][] a;
a = new int[3][];
a[0] = new int[5];
a[1] = new int[5];
a[2] = new int[5];
```

La prima riga:

```
a = new int[3][];
```

dichiara a come un array di lunghezza 3 i cui elementi possono essere array di qualsiasi lunghezza. Ognuna delle successive tre righe di codice crea un array di interi di lunghezza 5 chiamati `a[0]`, `a[1]` e `a[2]`. Il risultato ottenuto è un array bidimensionale di tipo base `int` formato da tre righe e cinque colonne.

Le istruzioni successive:

```
a[0] = new int[5];
a[1] = new int[5];
a[2] = new int[5];
```

suggeriscono la domanda "È proprio necessario che le lunghezze siano tutte pari a 5?" La risposta è "No!" Poiché un array bidimensionale in Java è un array di array, le sue righe possono contenere un numero diverso di elementi. Ovvero, righe diverse possono avere numeri di colonne diversi. Questo tipo di array è chiamato **array irregolare** (*ragged array*). Le seguenti istruzioni definiscono un array irregolare `b` in cui ogni riga ha una lunghezza diversa:

```
int[][] b;
b = new int[3][];
b[0] = new int[5]; //Prima riga, 5 elementi
b[1] = new int[7]; //Seconda riga, 7 elementi
b[2] = new int[4]; //Terza riga, 4 elementi
```

È giusto notare che dopo aver riempito il precedente array `b` con dei valori, non è possibile visualizzare `b` utilizzando il metodo `visualizzaTabella` definito nel Listato 6.11. Tuttavia, è possibile modificare tale metodo come segue:

```
/**
unArray puo' essere un array bidimensionale qualsiasi
*/
public static void visualizzaTabella(int[][] unArray) {
    for (int riga = 0; riga < unArray.length; riga++) {
        System.out.print((riga + 1) + "    ");
        for (int colonna = 0; colonna < unArray[riga].length; colonna++)
            System.out.print("€" + unArray[riga][colonna] + " ");
        System.out.println();
    }
}
```

Questo metodo modificato funziona per qualsiasi array bidimensionale, anche se è irregolare.

È possibile utilizzare in modo proficuo gli array irregolari in alcune situazioni, ma la maggior parte delle applicazioni non li richiede. Tuttavia, se si sono capiti gli array irregolari, si avrà una maggior conoscenza del funzionamento in generale di tutti gli array multidimensionali in Java.

Utilizzare indici che partono da 0 con gli array multidimensionali

Per un array monodimensionale `a`, è stato suggerito di ignorare `a[0]` e di definire un elemento in più nell'array. Sprecare un'area di memoria che contiene un indirizzo non rappresenta un grande problema. Ma con un array bidimensionale, il fatto di ignorare gli elementi i cui indici sono 0, provoca uno spreco di un'intera riga, di un'intera colonna o di una riga e di una colonna. In generale, è una pratica sconsigliabile.

Utilizzare le enumerazioni con gli array

È possibile utilizzare i valori di un'enumerazione come indici di un array. Per esempio, se si deve utilizzare un array che contenga un elemento per ogni giorno della settimana lavorativa, si può definire l'enumerazione

```
enum Giorni {LUN, MAR, MER, GIO, VEN}
```

Va ricordato che, con questa definizione, il valore ordinale di `LUN` è 0, di `MAR` è 1 e così via. Utilizzando un'enumerazione è possibile rendere molto più comprensibile un programma.

6.5 Riepilogo

- Si può pensare a un array come a una collezione di variabili tutte dello stesso tipo.
- Gli array sono creati con l'operatore `new`.
- Gli elementi di un array sono numerati da 0 fino a (*lunghezza dell'array* – 1). Se `a` è un array, `a[i]` è una variabile indicizzata dell'array. L'indice `i` deve avere un valore maggiore o uguale a 0 e strettamente minore di `a.length`, la lunghezza dell'array. Se `i` assume un qualsiasi altro valore durante l'esecuzione del programma, si incorre in un errore di indice fuori dai limiti (*index out of bounds*), che causa un messaggio di errore.
- Quando una variabile indicizzata viene utilizzata come argomento di un metodo, viene trattata come qualsiasi altro argomento il cui tipo di dato sia lo stesso del tipo base dell'array. In particolare, se il tipo base è un tipo primitivo, il metodo non può cambiare il valore della variabile indicizzata, ma se il tipo base è una classe, il metodo può cambiare lo stato dell'oggetto nella variabile indicizzata.

- È possibile passare a un metodo un intero array. Il metodo può cambiare il valore di un array di tipi primitivi o lo stato degli oggetti nell'array.
- Il valore restituito di un metodo può essere un array.
- Quando si utilizza solo una parte di un array, normalmente si memorizzano i valori nel segmento iniziale dell'array e si assegna a una variabile `int` il numero di questi valori. In questo caso, si ha un array parzialmente riempito.
- L'algoritmo *selection sort* ordina un array di valori, per esempio numeri, in senso crescente o decrescente.
- Gli array possono avere più di un indice; in questo caso vengono chiamati array multidimensionali. Gli array multidimensionali vengono implementati in Java come array di array.
- Un array bidimensionale può essere considerato come una tabella bidimensionale di righe e colonne. Una variabile indicizzata di un dato array è un elemento in cui riga e colonna sono indicati dai suoi indici: il primo indice designa la riga e il secondo la colonna.

6.6 Esercizi

1. Scrivere un programma in una classe `NumeriSottoLaMedia` che conti il numero di giorni in cui la temperatura è al di sotto della media. Leggere 10 temperature da tastiera e memorizzarle in un array. Calcolare la temperatura media e contare e visualizzare il numero di giorni in cui la temperatura è al di sotto della media.
2. Scrivere un programma in una classe `ContaFamiglie` che conti il numero di famiglie il cui reddito è al di sotto di un certo valore. Leggere un intero `k` da tastiera e, in seguito, creare un array di valori `double` di dimensione `k`. Leggere dalla tastiera `k` valori che rappresentano i redditi delle famiglie e memorizzarli nell'array. Trovare il più elevato tra questi redditi. Poi contare le famiglie il cui reddito è inferiore fino al 10% rispetto al reddito massimo. Visualizzare questo conteggio e i redditi di queste famiglie.
3. Scrivere un programma in una classe `ConteggioFiori` che calcoli il prezzo di vendita di mazzi di fiori. Ci sono cinque tipi di fiori (petunie, viole del pensiero, rose, violette e garofani) che vengono venduti, rispettivamente, a € 0.50, € 0.75, € 1.50, € 0.50 e € 0.80 per fiore. Creare un array di stringhe che memorizzi il nome di questi fiori. Creare un altro array che memorizzi i prezzi per ogni tipo di fiore. Il programma dovrebbe leggere il nome del fiore e la quantità desiderata dal cliente. Localizzare il nome del fiore nell'array e utilizzare l'indice per trovare il suo prezzo nell'array dei prezzi. Calcolare e visualizzare il prezzo totale del mazzo di fiori.
4. Scrivere un programma in una classe `FrequenzaCarattere` che conti il numero di volte che una cifra appare in un numero telefonico. Il programma deve creare un array di dimensione 10 che memorizzerà il conto di ogni cifra da 0 a 9. Leggere da tastiera un numero di telefono come stringa. Esaminare ogni carattere del numero telefonico e incrementare il conteggio relativo alla cifra nell'array. Visualizzare il contenuto dell'array.

5. Scrivere un metodo statico `strettamenteCrescente(double[] in)` che restituisce `true` se ogni valore dell'array fornito in ingresso è maggiore del valore che lo precede, altrimenti restituisce `false`.

6. Scrivere un metodo statico `rimuoiviDuplicati(char[] in)` che restituisce un nuovo array con i caratteri presenti nell'array passato come argomento, ma senza caratteri duplicati. Mantenere sempre la prima copia del carattere e rimuovere solo le successive. Per esempio, se `in` contiene `b, d, a, b, f, a, g, a, a, f`, il metodo restituirà un array contenente `b, d, a, f, g`. *Suggerimento*: un modo per risolvere questo problema è creare un array booleano della stessa dimensione dell'array `in` da utilizzare per tenere traccia dei caratteri da mantenere. I valori nel nuovo array booleano determineranno la dimensione dell'array da restituire.

7. Scrivere un metodo statico `rimuovi(int v, int[] in)` che restituisce un nuovo array con gli interi presenti nell'array passato come argomento, ma con il valore v rimosso. Per esempio, se v è 3 e `in` contiene 0, 1, 3, 2, 3, 0, 3, 1, il metodo restituirà un array contenente 0, 1, 2, 0, 1.

8. Si supponga di voler vendere scatole di dolci per una raccolta fondi. Si hanno cinque tipi di dolci da vendere: dolci alla menta, cioccolatini alle mandorle, biscotti al cioccolato, dolci al cioccolato e lecca-lecca senza zucchero. Si vuole memorizzare l'ordine di un cliente in un array di cinque interi, i quali rappresentano il numero di scatole per ogni tipo di dolce. Scrivere un metodo statico `combinaOrdini` che prende come argomento due ordini e restituisce un array che rappresenta la somma degli ordini. Per esempio, se `ordine1` contiene 0, 0, 3, 4, 7 e `ordine2` contiene 0, 4, 0, 1, 2, il metodo deve restituire un array contenente 0, 4, 3, 5, 9.

9. Scrivere un metodo statico per il *selection sort* che ordini un array di caratteri.

10. Riscrivere il metodo `selectionSort` che appare nel Listato 6.8 così che ordini un array il cui intervallo di indici vada da `inizio` a fine, dove $0 \leq$ `inizio` $\leq$ fine e fine è minore della lunghezza dell'array. Il nuovo metodo dovrà chiamarsi `selectionSortParziale`.

11. Correggere il metodo `selectionSort` che appare nel Listato 6.8 così che chiami il metodo descritto nell'esercizio precedente.

12. Scrivere una ricerca sequenziale di un array di interi, supponendo che l'array sia ordinato in maniera crescente. *Suggerimento*: si consideri un array che contiene i quattro interi 2, 4, 6, 8. Come si fa a dire che 5 non è presente nell'array senza confrontarlo con tutti gli interi dell'array?

13. Scrivere un metodo statico `cercaFigura(figura, soglia)` dove `figura` è un array bidimensionale di valori `double`. Il metodo dovrebbe restituire un nuovo array bidimensionale i cui elementi sono o 0.0 o 1.0. Il valore 1.0 indica che il valore corrispondente nell'array `figura` eccede `soglia` volte la media di tutti i valori in `figura`. Gli altri elementi sono a 0.0. Per esempio, se i valori in `figura` sono:

1.2	1.3	4.5	6.0	2.7
1.7	3.3	4.4	10.5	17.0
1.1	4.5	2.1	25.3	9.2
1.0	9.5	8.3	2.9	2.1

Il valore medio è 5.93. L'array risultante per una `soglia` pari a 1.4 dovrebbe essere il seguente:

0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	1.0	1.0
0.0	0.0	0.0	1.0	1.0
0.0	1.0	0.0	0.0	0.0

L'array risultante per una `soglia` pari a 0.6 dovrebbe essere il seguente:

0.0	0.0	1.0	1.0	0.0
0.0	0.0	1.0	1.0	1.0
0.0	1.0	0.0	1.0	1.0
0.0	1.0	1.0	0.0	0.0

14. Scrivere un metodo statico `blur(double[][] immagine)` che può essere utilizzato su una porzione di una immagine per oscurare dettagli come il viso di una persona o la targa di un'auto. Questo metodo calcola le medie pesate dei pixel in `immagine` e le restituisce in un nuovo array bidimensionale. Per calcolare la media pesata di un insieme di numeri, alcuni di questi vengono considerati maggiormente rispetto ad altri. Di conseguenza, si moltiplica ogni numero per il suo peso, si sommano i prodotti ottenuti e si divide questo risultato per la somma dei pesi.
Per ogni pixel in `immagine`, si calcoli la media pesata dell'elemento e di quelli adiacenti. Si memorizzi il risultato in un nuovo array bidimensionale nella stessa posizione occupata dal pixel nell'array `immagine`. Il metodo restituisce questo array.
I pixel adiacenti di un pixel di `immagine`, sono quelli che si trovano sopra, sotto, a destra, a sinistra e in diagonale. Di conseguenza, ogni media pesata nel nuovo array sarà una combinazione di (al più) nove pixel dell'array `immagine`. Un pixel che si trova nell'angolo, utilizzerà solo quattro pixel: se stesso e i tre adiacenti. Un pixel che si trova al bordo, utilizzerà solo sei pixel: se stesso e i cinque adiacenti. Un pixel che si trova nel mezzo, utilizzerà tutti e nove i pixel: se stesso e gli otto adiacenti. Di conseguenza, occorre trattare gli angoli e i bordi in maniera appropriata.
I pesi da utilizzare sono:

1 2 1
2 4 2
1 2 1

Il pixel stesso ha il peso più alto: 4. I pixel adiacenti in orizzontale e verticale hanno un peso pari a 2, mentre i pixel adiacenti in diagonale, hanno un peso pari a 1.
Per esempio, si supponga che i valori dei pixel dell'array `immagine` siano quelli mostrati di seguito:

1.2	1.3	4.5	6.0	2.7
1.7	3.3	4.4	10.5	17.0
1.1	4.5	2.1	25.3	9.2
1.0	9.5	8.3	2.9	2.1

e l'array risultante sia chiamato `risultato`. Nell'assegnare i pesi, si inizia con un pixel arbitrario e si considerano i pixel adiacenti in senso orario. Quindi, il valore di `risultato[1][1]` sarà dato da:

$$\frac{4(3.3) + 2(4.4) + 1(2.1) + 2.(4.5) + 1(1.1) + 2(1.7) + 1(1.2) + 2(1.3) + 1(4.5)}{4+2+1+2+1+2+1+2+1}$$

Per ottenere questa formula, si parte dal pixel alla posizione `figura[1][1]` e quindi, iniziando con il pixel adiacente alla sua destra, si considerano i vari pixel in senso orario. Il valore all'angolo in `risultato[0][0]` sarà uguale a:

$$\frac{4(1.2) + 2(1.3) + 1(3.3) + 2(1.7)}{4+2+1+2}$$

Si noti che `figura[0][0]` ha meno pixel adiacenti rispetto ai pixel interni come `figura[1][1]`. Analogamente, anche un pixel al bordo, come `figura[0][1]`, ha meno pixel adiacenti. Di conseguenza, il valore al bordo in `risultato[0][1]` sarà uguale a:

$$\frac{4(1.3) + 2(4.5) + 1(4.4) + 2(3.3) + 1(1.7) + 2(1.2)}{4+2+1+2+1+2}$$

L'array `risultato` sarà il seguente:

1.57	2.44	4.60	6.73	7.48
1.98	2.87	5.97	10.37	12.01
2.63	4.09	7.48	11.40	11.58
3.30	5.73	7.67	7.86	6.43

6.7 Progetti

1. Scrivere un programma che legga degli interi, uno per riga, e visualizzi la loro somma. Deve visualizzare, inoltre, tutti i numeri letti, ognuno con un'annotazione che indichi il contributo percentuale alla somma. Utilizzare un metodo che prende come argomento un intero array e che restituisce la somma dei numeri nell'array. *Suggerimento*: chiedere all'utente il numero di interi che verranno inseriti, creare un array di quella lunghezza e poi riempirlo con gli interi letti. Ecco una possibile interazione tra il programma e l'utente:

```
Quanti numeri verranno inseriti?
4
Inserire 4 interi, uno per riga:
2
1
1
2
```

```
La somma e' 6.
I numeri sono:
2, che e' il 33.3333% della somma.
1, che e' il 16.6666% della somma.
1, che e' il 16.6666% della somma.
2, che e' il 33.3333% della somma.
```

2. Scrivere un programma che legge una riga di testo che termina con un punto, che funge da valore sentinella. Visualizzare tutte le lettere presenti nel testo, una per riga e in ordine alfabetico, indicando il numero di volte che si presenta nel testo. Utilizzare un array di tipo base `int` di lunghezza 21, così che l'elemento all'indice 0 contenga il numero di lettere "a", l'elemento all'indice 1 contenga il numero di lettere "b" e così via. Considerare come input sia le lettere maiuscole, sia quelle minuscole, ma poi, nel conteggio, considerarle uguali. *Suggerimento*: prima di elaborare la riga di testo, utilizzare uno dei metodi `toUpperCase` o `toLowerCase` della classe `String`. Potrebbe essere utile definire un metodo che prende come argomento un carattere e restituisce un valore `int` che corrisponde al corretto indice di quel carattere. Per esempio, l'argomento `'a'` restituirà 0, l'argomento `'b'` restituirà 1, e così via. Si noti che è possibile utilizzare una conversione di tipo, come `(int)lettera`, per trasformare un `char` in `int`. Sicuramente questo non restituirà il valore desiderato, ma se si sottrae `(int)'a'`, si avrà l'indice corretto. Infine, si permetta all'utente di ripetere queste operazioni finché vuole.

3. Una palindroma è una parola o una frase che non cambia se letta da sinistra a destra o viceversa, ignorando gli spazi e senza distinguere fra lettere maiuscole e minuscole. Per esempio, le seguenti frasi e parole sono palindrome:

   a. `Ai lati d'Italia`

   b. `radar`

   c. `I topi non avevano nipoti`

   d. `xyzczyx`

   Scrivere un programma che accetta una sequenza di caratteri terminata da un punto e decide se tale stringa (escluso il punto) è palindroma. Si supponga che l'input contenga solo caratteri e spazi e sia al più lungo 80 caratteri. Includere un ciclo che consenta all'utente di controllare più stringhe. *Suggerimento*: definire un metodo statico chiamato `palindroma` che inizi come segue:

```
/**
Precondizione: L'array contiene lettere
e spazi nelle posizioni da a[0] fino a a[utilizzato - 1].
Restituisce true se la stringa e' palindroma
e false altrimenti.
*/
public static boolean palindroma(char[] a, int utilizzato)
```

Il programma deve leggere i caratteri in input e inserirli in un array il cui tipo base è `char` e in seguito chiamare questo metodo. La variabile `int utilizzato` tiene traccia di quanta parte dell'array è utilizzata, come descritto nel paragrafo "Array parzialmente riempiti".

4. Aggiungere il metodo `bubbleSort` alla classe `OrdinaArray`, fornita nel Listato 6.8, che realizzi un **bubble sort** di un array. L'algoritmo *bubble sort* esamina tutte le coppie di elementi adiacenti nell'array, dall'inizio alla fine, e scambia due elementi se non sono ordinati. Ogni scambio renderà l'array più ordinato, finché non si giungerà all'ordinamento completo. L'algoritmo in pseudocodice è il seguente.

   Algoritmo bubble sort per ordinare un array a

   Ripetere ciò che segue finché l'array a non è ordinato.

   ```
   for (indice = 0; indice < a.length - 1; indice++)
      if (a[indice] > a[indice + 1])
         Scambia i valori di a[indice] e a[indice + 1].
   ```

   L'algoritmo *bubble sort* di norma impiega più tempo degli altri metodi di ordinamento.

5. Aggiungere il metodo `insertionSort` alla classe `OrdinaArray`, fornita nel Listato 6.8, che realizzi un **insertion sort** di un array. Per semplificare il progetto, questo algoritmo *insertion sort* utilizzerà un array aggiuntivo. L'algoritmo deve copiare gli elementi dall'array originale in questo secondo array, inserendo ogni elemento nella sua corretta posizione. Questo richiederà di trasferire un certo numero di elementi dall'array di origine a quello ordinato.

   Algoritmo insertion sort per ordinare un array a

   ```
   for (indice = 0; indice < a.length; indice++)
      Inserire il valore di a[indice] nella corretta posizione dell'array temp,
      in modo che gli elementi copiati nell'array temp finora siano ordinati.
   Copiare tutti gli elementi da temp in a.
   ```

   L'array `temp` è parzialmente riempito ed è una variabile locale nel metodo `ordina`.

6. I sistemi tradizionali di inserimento delle password sono soggetti al cosiddetto *shoulder surfing*, nel quale qualcuno osserva un utente ignaro mentre inserisce una password o un PIN e li utilizza successivamente per accedere all'account. Un modo per combattere questo problema è utilizzare un sistema basato su coppie domanda-risposta generate in modo casuale. In tali sistemi, l'utente deve fornire ogni volta informazioni diverse in risposta a un quesito generato in maniera casuale. Si consideri lo schema seguente, nel quale la password consiste in un PIN a cinque cifre (da 00000 a 99999). Ad ogni cifra è associato un numero casuale scelto tra 1, 2 e 3. L'utente inserirà, al posto del PIN vero e proprio, i numeri casuali corrispondenti alle cifre del PIN.

   Per esempio, si supponga che il vero PIN sia 12345. In fase di autenticazione, all'utente verrà mostrata una schermata come:

   PIN: 0 1 2 3 4 5 6 7 8 9

   NUM: 3 2 3 1 1 3 2 2 1 3

A questo punto, l'utente inserirà il numero 23113 al posto di 12345. In questo modo, la password non viene svelata nemmeno se qualcuno vede cosa viene inserito, perché 23113 potrebbe corrispondere anche a molti altri PIN, come 69440 o 70439. Al successivo accesso verrà generata una sequenza diversa di numeri casuali, come:

```
PIN:    0 1 2 3 4 5 6 7 8 9
NUM:    1 1 2 3 1 2 2 3 3 3
```

Si scriva un programma che simuli la procedura di autenticazione, inserendo un PIN direttamente nel codice. Il programma dovrebbe utilizzare un array per associare i numeri casuali alle cifre da 0 a 9. Inoltre, dovrà mostrare sullo schermo i numeri casuali, leggere la risposta dell'utente e comunicare se la risposta dell'utente è corretta o meno.

Capitolo 7

Ricorsione

OBIETTIVI

- Descrivere il concetto di ricorsione.
- Utilizzare la ricorsione come strumento di programmazione.
- Descrivere e utilizzare la forma ricorsiva dell'algoritmo di ricerca binario per cercare un dato elemento in un array.
- Descrivere e utilizzare l'algoritmo *merge sort* per ordinare un array.

Molti ritengono che non si dovrebbe mai definire un concetto riferendosi al concetto stesso, poiché ciò produrrebbe un riferimento circolare. Tuttavia, vi sono situazioni nelle quali definire un metodo in termini del metodo stesso non solo è possibile, ma persino utile. Se fatto correttamente, questo procedimento non produce nemmeno un riferimento circolare. Il linguaggio Java consente, in un certo senso, di definire un metodo in termini del metodo stesso. Più precisamente, la definizione di un metodo Java può contenere una chiamata al metodo stesso che si sta definendo, cioè il metodo può invocare se stesso. I metodi con questa caratteristica costituiscono l'argomento di questo capitolo.

Prerequisiti

Per comprendere questo capitolo è necessario conoscere il contenuto dei Capitoli da 1 a 5. Solamente gli ultimi due paragrafi del capitolo (il caso di studio "Ricerca binaria" e l'esempio di programmazione "Merge sort – Un metodo di ordinamento ricorsivo") richiedono la conoscenza degli array, presentati nel Capitolo 6.

7.1 Le basi della ricorsione

Accade spesso che il modo più naturale per progettare un algoritmo implichi l'applicazione dell'algoritmo stesso in uno o più casi particolari. Per esempio, un algoritmo per la ricerca di un nome in un elenco telefonico potrebbe essere schematizzato come segue: aprire l'elenco telefonico alla pagina centrale. Se il nome compare in quella pagina, la ricerca è finita. Se il nome, in base all'ordine alfabetico, compare prima di quella pagina,

effettuare la ricerca nella prima metà dell'elenco. Se invece il nome compare dopo quella pagina, effettuare la ricerca nella seconda metà. La ricerca effettuata in una delle due metà dell'elenco è semplicemente una versione ridotta della ricerca nell'elenco intero.

Quando una parte di un algoritmo costituisce una versione ridotta dell'algoritmo completo, quest'ultimo è definito **ricorsivo**. Un algoritmo ricorsivo può essere implementato tramite un **metodo ricorsivo**, cioè un metodo che contenga una chiamata a se stesso, detta **chiamata ricorsiva**.

L'obiettivo di questo capitolo è quello di spiegare come devono essere definiti i metodi ricorsivi affinché il codice funzioni correttamente, un argomento noto in generale come **ricorsione**.

L'argomento verrà introdotto mediante un semplice esempio che illustra la ricorsione in Java. Data una variabile di tipo intero num, si vogliono stampare tutti i numeri da num in giù fino a 1. Se num è minore di 1, si vuole stampare solo un ritorno a capo. Per esempio, se num vale 3, il programma dovrà stampare:

```
321
```

Un programma come questo può ovviamente essere scritto utilizzando un semplice ciclo `for`. L'utilizzo del ciclo `for` sarebbe anzi la soluzione migliore per questo problema, ma verrà comunque presentato il metodo ricorsivo `contoAllaRovescia` per mostrare come funziona la ricorsione in un caso semplice. Più avanti in questo capitolo, verranno presentati altri problemi per i quali la soluzione ricorsiva è la migliore e la più elegante.

Per prima cosa, si consideri il caso più semplice del problema da risolvere. Si tratta solitamente del caso in cui i dati in ingresso sono i più semplici o assumono i valori minori tra quelli ammessi e può generalmente essere risolto immediatamente. Nell'esempio, il dato di partenza più semplice corrisponde a un valore di num minore di 1. In tal caso, l'unica cosa da fare è stampare un ritorno a capo. Questo caso è chiamato **caso base**, o **caso di arresto**, perché non richiederà ulteriori chiamate ricorsive. Come si vedrà in seguito, quando si verifica questo caso la ricorsione termina. Inoltre, se questo caso semplice non funziona correttamente, nessun altro caso potrà funzionare.

Il codice di un metodo che implementi il caso base per l'esempio del conto alla rovescia è semplice:

```
public static void contoAllaRovescia(int num) {
    if (num <= 0) {
        System.out.println();
    }
}
```

Questo metodo gestisce il caso base ma non i casi più interessanti nei quali num è maggiore o uguale a 1. Per risolvere ricorsivamente questi casi, è necessario ridurre il problema a una sua versione più semplice. Ciò significa riformulare il problema in termini di un'altra chiamata al metodo `contoAllaRovescia`. Si supponga che num valga 3, cioè che sia stata effettuata la chiamata `contoAllaRovescia(3)`. Si vuole stampare "321", cioè stampare un "3" seguito da "21". Ma "21" è proprio ciò che dovrebbe essere stampato dalla chiamata `contoAllaRovescia(2)`! L'algoritmo diventa quindi:

- stampare 3;
- chiamare `contoAllaRovescia(2)`, che si occuperà di stampare 21.

A sua volta, la chiamata `contoAllaRovescia(2)` può ripetere lo stesso procedimento:

- stampare 2;
- chiamare `contoAllaRovescia(1)`, che si occuperà di stampare 1.

La chiamata `contoAllaRovescia(1)` ripeterà nuovamente il procedimento:

- stampare 1;
- chiamare `contoAllaRovescia(0)`, che stamperà un ritorno a capo.

La chiamata `contoAllaRovescia(0)` esegue il caso base, che stampa un ritorno a capo e termina la sequenza di chiamate ricorsive.

Per un valore generico di num, la soluzione ricorsiva diventa:

- stampare num;
- chiamare `contoAllaRovescia(num – 1)`, che si occuperà di stampare da num – 1 a 1.

Il Listato 7.1 mostra l'implementazione completa in Java di questo algoritmo.

LISTATO 7.1 Un metodo ricorsivo per il conto alla rovescia.

MyLab

```
public class ContoAllaRovesciaRicorsivo {
    public static void main(String[] args) {
        contoAllaRovescia(3);
    }

    public static void contoAllaRovescia(int num) {
        if (num <= 0) {
            System.out.println();
        } else {
            System.out.print(num);
            contoAllaRovescia(num - 1);
        }
    }
}
```

Esempio di output

```
321
```

La ricorsione è possibile perché a ogni chiamata di `contoAllaRovescia` corrisponde una copia distinta della variabile num che è locale alla chiamata corrispondente. La Figura 7.1 illustra la sequenza di chiamate ricorsive e mostra come la variabile num venga passata da una chiamata del metodo all'altra.

Metodi ricorsivi

Un metodo ricorsivo richiama se stesso, cioè la sua definizione contiene una chiamata al metodo stesso.

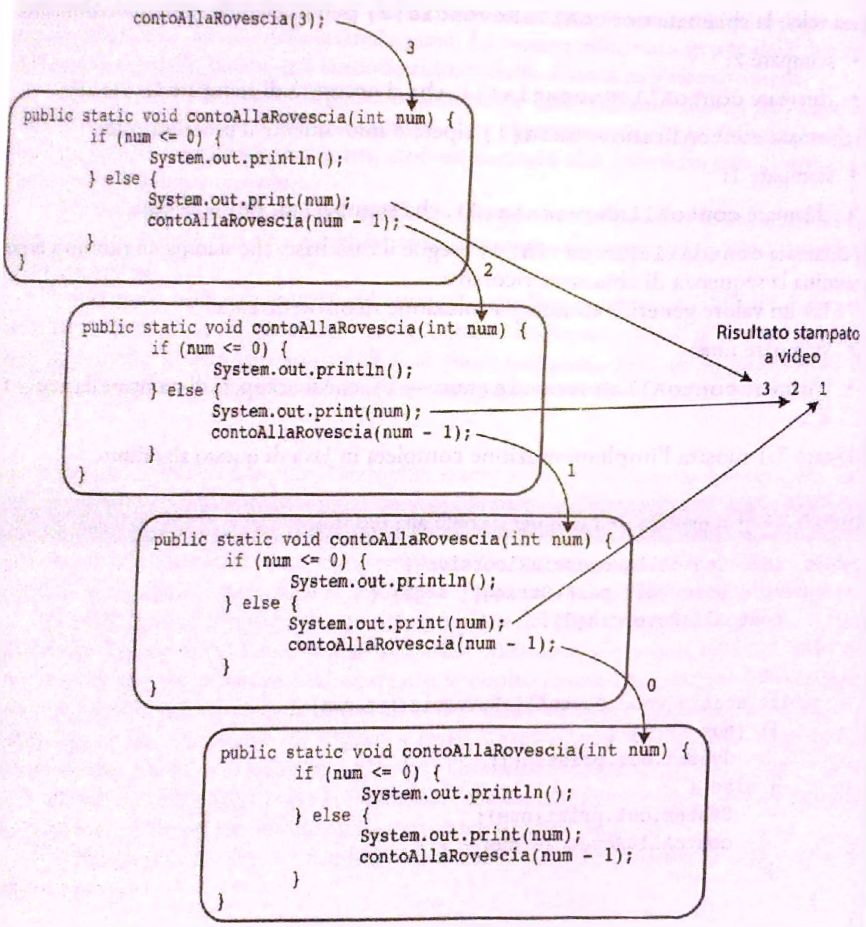


Figura 7.1 Chiamate ricorsive per il metodo `contoAllaRovescia`.

CASO DI STUDIO
CONVERSIONE DA CIFRE A PAROLE

In questo caso di studio verrà definito un metodo che accetta come parametro un intero e scrive "a parole" le cifre che compongono il numero. Per esempio, se l'argomento fosse il numero 223, il metodo dovrebbe mostrare

```
due due tre
```

Una possibile intestazione per il metodo è

```
/**
Precondizione: numero >= 0
Scrive a parole le cifre del numero.
*/
public static void scriviAParole(int numero)
```

Se numero è composto da una sola cifra, si può utilizzare un lungo costrutto `switch` per decidere quale parola utilizzare per quella cifra. Delegheremo questo compito al metodo `daCifraAParola`, che può essere chiamato da `scriviAParole`. Possiamo specificare tale metodo come segue:

```
/**
Precondizione: 0 <= cifra <= 9
Restituisce la parola corrispondente alla cifra passata come parametro.
*/
public static String daCifraAParola(int cifra)
```

Pertanto, quando `cifra` vale 0, `daCifraAParola` restituisce "`zero`", quando `cifra` vale 1 il metodo restituisce "uno" e così via.

Se il numero è formato da più cifre, possiamo suddividere il compito da eseguire in operazioni più semplici in molti modi. Alcuni di questi si prestano a soluzioni che utilizzano la ricorsione, mentre altri no. Un buon modo per suddividere il compito in due compiti più semplici, tali che uno possa essere completato immediatamente mentre il secondo si presta all'uso della ricorsione, è il seguente:

1. Scrivere a parole tutte le cifre tranne l'ultima.
2. Scrivere a parole l'ultima cifra.

Il primo compito è una versione ridotta del problema originale. Si tratta, infatti, dello stesso problema di partenza, a parte il fatto che il numero da considerare contiene una cifra in meno. Pertanto, il primo compito può essere portato a termine chiamando semplicemente il metodo stesso che si sta definendo. Il secondo compito è realizzato tramite una chiamata al metodo `daCifraAParola`. Questa situazione suggerisce il seguente raffinamento dei due passi descritti sopra:

Algoritmo per `scriviAParole(numero)`

1. `scriviAParole(numero senza l'ultima cifra);`
2. `System.out.print(daCifraAParola(ultima cifra di numero) + " ");`

Come si può rimuovere l'ultima cifra da un numero, in modo da completare il passo 1, conservando la cifra per il passo 2? Si consideri per esempio un numero formato da più cifre, come 987, da dividere in due parti, 98 e 7. Ciò può essere realizzato utilizzando la divisione intera per 10: infatti, `987 / 10` è uguale a 98, mentre `987 % 10` vale 7. L'algoritmo considerato porta quindi al seguente codice Java:

```
scriviAParole(numero / 10);
System.out.print(daCifraAParola(numero % 10) + " ");
```

Utilizzando queste istruzioni come implementazione del metodo `scriviAParole`, si ottiene la definizione seguente:

```
public static void scriviAParole(int numero) {
    // Non ancora del tutto corretto
    scriviAParole(numero / 10);
    System.out.print(daCifraAParola(numero % 10) + " ");
}
```

Come indicato dal commento, però, questo metodo non funzionerà correttamente. Il metodo così definito sfrutta l'idea di base, ma presenta un problema serio: la definizione precedente presume, infatti, che il parametro `numero` sia un numero a più cifre. È necessario trattare come caso particolare la situazione nella quale il numero contiene una sola cifra. Come si vedrà in seguito, in nessun caso il metodo funzionerà correttamente se non viene trattato nel modo giusto questo caso particolare. Questa osservazione porta a modificare la definizione del metodo in questo modo:

```
public static void scriviAParole(int numero) {
    if (numero < 10)
        System.out.print(daCifraAParola(numero) + " ");
    else {    // il numero è composto da due o più cifre
        scriviAParole(numero / 10);
        System.out.print(daCifraAParola(numero % 10) + " ");
    }
}
```

Il parametro della chiamata ricorsiva `scriviAParole(numero / 10)` è minore di `numero`, cioè del parametro passato alla precedente chiamata di `scriviAParole`. È importante che una chiamata ricorsiva in un metodo risolva una versione "ridotta" del problema originale (secondo un'interpretazione per il momento intuitiva dell'aggettivo "ridotta" e che sarà resa più precisa prima della fine di questo capitolo).

Come si vedrà nel prossimo paragrafo, l'esecuzione corretta di un metodo ricorsivo come `scriviAParole` richiede che il caso più semplice (caso base) non implichi a sua volta una chiamata ricorsiva. Nella definizione di `scriviAParole`, tale caso base è rappresentato dai numeri a una cifra, che sono elaborati come segue:

```
if (numero < 10)
    System.out.println(daCifraAParola(numero) + " ");
```

Come si può vedere, in questo caso non interviene alcuna chiamata ricorsiva.

La definizione del metodo `scriviAParole` è ora completa. Il Listato 7.2 mostra tale metodo incluso in un programma di esempio.

MyLab

LISTATO 7.2 Un esempio di ricorsione.

```
import java.util.Scanner;

public class RicorsioneDemo1 {
    public static void main(String[] args) {
        System.out.println("Inserire un intero:");
        Scanner tastiera = new Scanner(System.in);
        int numero = tastiera.nextInt();
        System.out.println("Le cifre in questo numero sono:");
        scriviAParole(numero);
        System.out.println();
        System.out.println("Aggiungendo 10 a questo numero, ");
```

```
        System.out.println("le cifre nel nuovo numero sono:");
        numero = numero + 10;
        scriviAParole(numero);
        System.out.println();
    }

    /**
    Precondizione: numero >= 0
    Scrive a parole le cifre del numero.
    */
    public static void scriviAParole(int numero) {
        if (numero < 10)
            System.out.print(daCifraAParola(numero) + " ");
        else { //numero ha due o più cifre
            scriviAParole(numero / 10); <-- Chiamata ricorsiva
            System.out.print(daCifraAParola(numero % 10) + " ");
        }
    }

    // Precondizione: 0 <= cifra <= 9
    // Restituisce la parola corrispondente alla cifra passata come parametro.
    public static String daCifraAParola(int cifra) {
        String risultato = null;
        switch (cifra) {
            case 0: risultato = "zero"; break;
            case 1: risultato = "uno"; break;
            case 2: risultato = "due"; break;
            case 3: risultato = "tre"; break;
            case 4: risultato = "quattro"; break;
            case 5: risultato = "cinque"; break;
            case 6: risultato = "sei"; break;
            case 7: risultato = "sette"; break;
            case 8: risultato = "otto"; break;
            case 9: risultato = "nove"; break;
            default:
                System.out.println("Errore grave.");
                System.exit(0);
            break;
        }
        return risultato;
    }
}
```

Esempio di output

```
Inserire un intero:
987
Le cifre in questo numero sono:
nove otto sette
Aggiungendo 10 a questo numero,
le cifre nel nuovo numero sono:
nove nove sette
```

7.1.1 Come funziona la ricorsione

Come viene gestita esattamente una chiamata ricorsiva da parte di un computer? Per comprendere i dettagli, si consideri la seguente chiamata al metodo `scriviAParole`:

```
scriviAParole(987);
```

Nonostante la definizione di `scriviAParole` riportata nel Listato 7.2 contenga una chiamata ricorsiva, Java non fa niente di speciale per gestire questa o altre chiamate di `scriviAParole`. Il valore dell'argomento `987` viene copiato nel parametro `numero` del metodo e viene eseguito il codice che ne risulta, che equivale a quello che segue:

```
{ // Codice per la chiamata di scriviAParole(987)
   if (987 < 10)
      System.out.print(daCifraAParola(987) + " ");
   else { // 987 ha due o più cifre
      scriviAParole(987 / 10);
      System.out.print(daCifraAParola(987 % 10) + " ");
   }
}
```

Dato che 987 non è minore di 10, viene eseguita l'istruzione composta dopo l'`else`, che inizia con la chiamata ricorsiva `scriviAParole(987 / 10)`. Il resto dell'istruzione composta non può essere eseguito finché non è stata completata questa chiamata ricorsiva. Si noti che questo rimarrebbe vero anche se il metodo chiamasse un metodo differente, anziché chiamare se stesso ricorsivamente. Quindi, l'esecuzione del codice di `scriviAParole(987)` è sospesa in attesa della fine dell'esecuzione di `scriviAParole(987 / 10)`. Una volta che la chiamata ricorsiva termina, l'operazione rimasta in sospeso può riprendere e viene eseguito il resto dell'istruzione composta.

La nuova chiamata ricorsiva, `scriviAParole(987 / 10)`, viene gestita come qualsiasi altra chiamata a metodo: il valore dell'argomento `987 / 10` viene copiato nel parametro `numero` e viene eseguito il codice risultante. Poiché il risultato di `987 / 10` è 98, il codice che si ottiene è equivalente al seguente:

```
{ // Codice per la chiamata di scriviAParole(98)
   if (98 < 10)
      System.out.print(daCifraAParola(98) + " ");
   else { // 98 ha due o più cifre
      scriviAParole(98 / 10);
      System.out.print(daCifraAParola(98 % 10) + " ");
   }
}
```

Di nuovo, l'argomento del metodo (che questa volta è `98`) non è minore di 10, di conseguenza l'esecuzione del nuovo codice implica una chiamata ricorsiva ed esattamente `scriviAParole(98 / 10)`. A questo punto l'esecuzione corrente viene sospesa in attesa del completamento della chiamata ricorsiva. Il valore del parametro `numero` è `98 / 10` (cioè 9) e viene intrapresa l'esecuzione del codice che segue:

```
{ // Codice per la chiamata di scriviAParole (9)
   if (9 < 10)
      System.out.print(daCifraAParola(9) + " ");
```

```
    else { // 9 ha due o più cifre
        scriviAParole(9 / 10);
        System.out.print(daCifraAParola(9 % 10) + " ");
    }
}
```

Dato che 9 è minore di 10, viene eseguita solo la prima parte del blocco `if-else`, cioè

```
System.out.print(daCifraAParola(9) + " ");
```

Questo caso è definito **caso base** o **caso di arresto**, in quanto non implica chiamate ricorsive. Dalla definizione del metodo `daCifraAParola` si vede che viene mostrata a video la stringa "nove". Ciò conclude la chiamata di `scriviAParole(98 / 10)`.

A questo punto l'esecuzione rimasta in sospeso e mostrata qui di seguito può riprendere dopo la posizione indicata dalla freccia:

```
{ //Codice per la chiamata a scriviAParole(98)
    if (98 < 10)
        System.out.print(daCifraAParola(98) + " ");
    else { //98 ha due o più cifre
        scriviAParole(98 / 10); <-- Chiamata ricorsiva.
        System.out.print(daCifraAParola(98 % 10) + " ");
    }
}
```

Quindi viene eseguita l'istruzione

```
System.out.print(daCifraAParola(98 % 10) + " ");
```

che produce a video la stringa "otto" e termina la chiamata ricorsiva `scriviAParole(98)`.

Il procedimento è quasi terminato. È rimasta in sospeso solo un'ultima esecuzione, mostrata nel codice seguente:

```
{ // Codice per la chiamata a scriviAParole(987)
    if (987 < 10)
        System.out.print(daCifraAParola(987) + " ");
    else { // 987 ha due o più cifre
        scriviAParole (987 / 10); <-- Chiamata ricorsiva.
        System.out.print(daCifraAParola(987 % 10) + " ");
    }
}
```

L'esecuzione riprende dopo la posizione indicata dalla freccia, con l'istruzione

```
System.out.print(daCifraAParola(987 % 10) + " ");
```

Questa istruzione produce a video la parola "sette" e conclude l'intero processo. La sequenza di chiamate ricorsive è mostrata in Figura 7.2.

Si noti che quando viene eseguita una chiamata di un metodo ricorsivo non accade nulla di speciale: gli argomenti vengono copiati nei parametri e viene eseguito il codice contenuto nella definizione del metodo, così come accadrebbe per la chiamata di qualunque metodo. Quando si incontra una chiamata ricorsiva, l'elaborazione viene temporaneamente sospesa, dato che per procedere è necessario conoscere il risultato della chiamata ricorsiva. Vengono salvate tutte le informazioni necessarie per poter riprende-

`scriviAParole(987)` è equivalente all'esecuzione di:

```
{ //Codice per la chiamata a scriviAParole(987)
    if (987 < 10)
        System.out.print(daCifraAParola(987) + " ");
    else { //987 ha due o più cifre
        scriviAParole (987 / 10);
        System.out.print(daCifraAParola(987 % 10) + " ");
    }
}
```

L'esecuzione si ferma qui in attesa del completamento della chiamata ricorsiva.

`scriviAParole(987/10)` è equivalente a `scriviAParole(98)`, che a sua volta è equivalente all'esecuzione di:

```
{ //Codice per la chiamata a scriviAParole(98)
    if (98 < 10)
        System.out.print(daCifraAParola(98) + " ");
    else { //98 ha due o più cifre
        scriviAParole(98 / 10);
        System.out.print(daCifraAParola(98 % 10) + " ");
    }
}
```

L'esecuzione si ferma qui in attesa del completamento della chiamata ricorsiva.

`scriviAParole(98/10)` è equivalente a `scriviAParole(9)`, che a sua volta è equivalente all'esecuzione di:

```
{ //Codice per la chiamata a scriviAParole(9)
    if (9 < 10)
        System.out.print(daCifraAParola(9) + " ");
    else { //9 ha due o più cifre
        scriviAParole(9 / 10);
        System.out.print(daCifraAParola(9 % 10) + " ");
    }
}
```

Non si verifica nessun'altra chiamata ricorsiva.

Figura 7.2 Esecuzione di una chiamata ricorsiva.

re l'esecuzione in seguito e viene eseguita la chiamata ricorsiva. Una volta completata quest'ultima, viene ripresa l'elaborazione più esterna.

Il linguaggio Java non pone vincoli all'utilizzo di chiamate ricorsive nella definizione di un metodo. Tuttavia, affinché tale definizione sia utile, deve essere progettata in modo che qualunque chiamata al metodo si concluda necessariamente con una qualche porzione di codice che non dipenda dalla ricorsione. Il metodo può chiamare se stesso e questa chiamata ricorsiva può a sua volta invocare il metodo un'altra volta. Questo processo può ripetersi un numero arbitrario di volte, ma non avrà mai termine, salvo che una delle chiamate ricorsive non dipenda a sua volta dalla ricorsione per produrre un risultato. La struttura generale per una corretta definizione di un metodo ricorsivo è la seguente:

- uno o più casi nei quali il metodo esegue il compito previsto tramite una o più chiamate ricorsive al fine di risolvere una o più versioni ridotte del problema originale;

- uno o più casi nei quali il metodo esegue il compito previsto senza ricorrere ad alcuna chiamata ricorsiva. Questi casi, privi di chiamate ricorsive, sono chiamati **casi base o di arresto.**

Spesso si usa un blocco `if-else` per determinare quale caso debba essere eseguito. Una situazione tipica è quella nella quale il metodo originale esegue un caso che prevede una chiamata ricorsiva. Quest'ultima può a sua volta eseguire un caso che richiede un'altra chiamata ricorsiva. Per un certo numero di volte, ogni chiamata ricorsiva ne genera un'altra, ma alla fine si dovrà ricadere in uno dei casi base. Ogni chiamata al metodo deve portare alla fine a un caso di arresto, altrimenti non terminerà mai, a causa di una sequenza infinita di chiamate *ricorsive* (nella realtà, una chiamata che generi una catena infinita di chiamate ricorsive terminerà tipicamente in modo non corretto anziché continuare all'infinito).

Il sistema più comune per garantire che si arrivi sempre a un caso base consiste nel far utilizzare alle chiamate ricorsive del metodo valori "più piccoli" dell'argomento. Per esempio, si consideri il metodo `scriviAParole` presentato nel Listato 7.2: il suo parametro è `numero`, mentre il parametro passato alla chiamata ricorsiva del metodo è il valore più piccolo `numero / 10`. In questo modo, a ogni successiva chiamata del metodo in una sequenza di chiamate ricorsive viene fornito un parametro il cui valore è inferiore. Dato che la definizione del metodo prevede un caso base per ogni argomento composto da un'unica cifra, si ha la garanzia che un caso base sarà sempre raggiunto.

Linee guida per l'uso corretto della ricorsione

La definizione di un metodo che include una chiamata ricorsiva al metodo stesso funziona solo seguendo alcune linee guida ben precise. Le regole seguenti si applicano nella maggior parte dei casi che implicano la ricorsione:

- il nucleo della definizione del metodo deve essere costituito da un blocco `if-else` o da qualche altro tipo di istruzione condizionale che permetta di gestire casi diversi in base a qualche proprietà del parametro del metodo;
- almeno una delle alternative dovrebbe contenere una chiamata ricorsiva del metodo. Questa deve utilizzare argomenti in un qualche modo "più piccoli" o risolvere versioni "ridotte" del compito realizzato dal metodo;
- almeno una delle alternative non deve contenere alcuna chiamata ricorsiva. Tali alternative costituiscono i casi base o di arresto.

Casi base (o di arresto)

I casi base devono essere progettati in modo da terminare ogni possibile sequenza di chiamate ricorsive. Una chiamata di un metodo può produrre una chiamata ricorsiva dello stesso metodo, la quale può generare a sua volta un'altra chiamata ricorsiva e così via, per un certo numero di volte, ma alla fine qualunque sequenza di questo tipo deve portare a un caso base che termina senza ulteriori chiamate ricorsive. In caso contrario, una chiamata del metodo potrebbe non terminare mai (o non terminare fino all'esaurimento delle risorse disponibili nel computer).

Un tipico metodo ricorsivo comprende un blocco `if-else` o un altro tipo di istruzione condizionale che distingua tra uno o più casi che implicano una chiamata ricorsiva del metodo e uno o più casi che concludono l'esecuzione del metodo senza chiamate ricorsive. Ogni sequenza di chiamate ricorsive deve portare, alla fine, a uno di questi casi non ricorsivi, o base.

Il sistema più comune per garantire che si arrivi sempre a un caso base consiste nel far utilizzare alle chiamate ricorsive del metodo valori "più piccoli" dell'argomento.

7.1.2 Ricorsione infinita

Si consideri il metodo `scriviAParole` analizzato nella sezione precedente; supponiamo di averlo incautamente definito come segue:

```
public static void scriviAParole(int numero) {
    // Non ancora corretto
    scriviAParole(numero / 10);
    System.out.print(daCifraAParola(numero % 10) + " ");
}
```

In effetti, questa era una delle proposte preliminari, prima di comprendere che era necessario considerare anche un ulteriore caso. Si supponga tuttavia di aver tralasciato tale caso e di aver utilizzato questa versione semplificata. È molto semplice delineare l'esecuzione della chiamata ricorsiva `scriviAParole(987)`.

La chiamata del metodo `scriviAParole(987)` genera la chiamata ricorsiva `scriviAParole(987 / 10)`, equivalente a `scriviAParole(98)`. A sua volta, la chiamata `scriviAParole(98)` produce la chiamata ricorsiva `scriviAParole(98 / 10)`, che equivale a `scriviAParole(9)`.

Dato che la versione errata di `scriviAParole` non prevede un caso particolare per numeri a una cifra, la chiamata di `scriviAParole(9)` produce la chiamata ricorsiva `scriviAParole(9 / 10)`, equivalente a `scriviAParole(0)`. La chiamata di `scriviAParole(0)` produce la chiamata ricorsiva `scriviAParole(0 / 10)`, anch'essa equivalente a `scriviAParole(0)`.

A questo punto il problema risulta evidente: la chiamata di `scriviAParole(0)` genera un'altra chiamata di `scriviAParole(0)`, che a sua volta produce un'altra chiamata di `scriviAParole(0)` e così via all'infinito (o finché il computer non esaurisce le risorse disponibili). Questa situazione è detta **ricorsione infinita**.

Si noti che la definizione di `scriviAParole` sopra riportata è errata solo nel senso che produce il risultato sbagliato: non è illegale. Il compilatore Java accetterà questa definizione di `scriviAParole` così come qualunque analoga definizione di un metodo ricorsivo che non preveda un caso particolare per interrompere la serie di chiamate ricorsive. Tuttavia, se una definizione ricorsiva non garantisce che un caso base verrà sempre raggiunto, si otterrà una sequenza infinita di chiamate ricorsive, a causa della quale il programma non terminerà mai o terminerà in modo non corretto.

Affinché la definizione di un metodo ricorsivo funzioni correttamente e non generi una sequenza infinita di chiamate ricorsive, devono essere previsti uno o più casi particolari (i casi base) che, per opportuni valori degli argomenti, arrestino la sequenza di chia-

mate ricorsive. La definizione corretta di `scriviAParole`, presentata nel Listato 7.2, prevede un caso base, evidenziato nel codice seguente:

```
public static void scriviAParole(int numero) {
    if (numero < 10)
        System.out.print(daCifraAParola(numero) + " ");
    else { // il numero ha due o più cifre
        scriviAParole(numero / 10);
        System.out.print(daCifraAParola(numero % 10) + " ");
    }
}
```

← Caso base

7.1.3 Lo stack e la ricorsione

Le chiamate ricorsive sono chiamate a metodi. Quindi, come specificato nel Capitolo 5, l'invocazione di un metodo comporta la creazione di un nuovo record di attivazione e il suo posizionamento in cima allo *stack*.

Grazie allo *stack*, il computer può tenere traccia facilmente delle chiamate ricorsive. Quando viene chiamato un metodo, viene creato un nuovo record di attivazione e i parametri formali del metodo vengono inizializzati con gli argomenti passati in ingresso al metodo. Il computer inizia quindi l'esecuzione del corpo della definizione del metodo. Quando incontra una chiamata ricorsiva, interrompe l'esecuzione in corso su quel record di attivazione per determinare il risultato della chiamata ricorsiva. Prima di fare questo, però, salva le informazioni necessarie per poter continuare l'elaborazione rimasta in sospeso una volta determinato il risultato della chiamata ricorsiva. Queste informazioni vengono scritte su un nuovo record di attivazione che viene posto in cima alla pila. Il computer sostituisce sul nuovo record di attivazione i parametri con gli argomenti passati al metodo e inizia l'esecuzione della chiamata ricorsiva. Quando arriva a una nuova chiamata ricorsiva, ripete il processo di salvataggio delle informazioni sullo *stack* e usa un nuovo record di attivazione per la nuova chiamata ricorsiva.

Questo procedimento prosegue finché qualche chiamata ricorsiva del metodo non completa la propria elaborazione senza produrre ulteriori chiamate ricorsive. A quel punto, il computer esamina il record di attivazione in cima alla pila, il quale contiene l'elaborazione, solo parzialmente completata, che è in attesa di quella ricorsiva appena terminata. È quindi possibile procedere con tale elaborazione. Al termine di quest'ultima, il computer elimina il record di attivazione corrispondente e l'elaborazione sospesa che si trovava sotto di esso nella pila diventa quella in cima. Il computer riprende l'esecuzione dell'elaborazione sospesa che si trova in cima alla pila e così di seguito. Il processo continua finché non viene completata l'elaborazione riportata sul record di attivazione alla base della pila. A seconda di quante chiamate ricorsive si verificano e di come è stato definito il metodo, la pila può crescere o ridursi in qualunque modo. Si noti che in ogni momento l'unico record di attivazione accessibile è l'ultimo che è stato posto sulla pila, ma ciò è esattamente quanto basta per tenere traccia delle chiamate ricorsive. Ogni elaborazione sospesa è in attesa del completamento di quella immediatamente sopra nella pila.

Overflow dello *stack*

C'è sempre un limite alle dimensioni dello *stack*. Se si verifica una lunga catena di chiamate ricorsive di un metodo, ogni chiamata ricorsiva produrrà il salvataggio sullo *stack* di un'altra elaborazione sospesa. Se questa sequenza è troppo lunga, lo *stack* cercherà di estendersi oltre i limiti. Questa condizione di errore è detta stack overflow. Quando si ottiene un messaggio di errore di tipo *stack overflow*, è probabile che qualche metodo abbia generato una sequenza eccessivamente lunga di chiamate ricorsive. Una causa tipica di questo tipo di errore è la ricorsione infinita. Se un metodo scatena una ricorsione infinita, prima o poi tenderà a far crescere lo *stack* oltre i limiti.

7.1.4 Confronto tra metodi ricorsivi e iterativi

Qualunque definizione di un metodo che includa una chiamata ricorsiva può essere riscritta in modo da svolgere lo stesso compito senza l'uso della ricorsione. Per esempio, il Listato 7.3 presenta una revisione del programma del Listato 7.2 nella quale la definizione di `scriviAParole` non utilizza la ricorsione. Entrambe le versioni svolgono lo stesso compito, cioè mostrano a video lo stesso risultato. Come in questo caso, la versione non ricorsiva della definizione di un metodo implica solitamente un ciclo al posto della ricorsione. Un processo ripetitivo e non ricorsivo è definito **iterazione**[1] e un metodo che implementi un processo di questo tipo è detto **metodo iterativo**.

Un metodo ricorsivo utilizza più memoria rispetto a una versione iterativa, a causa del carico aggiuntivo sul sistema che deriva dalla necessità di tenere traccia delle chiamate ricorsive e delle elaborazioni rimaste in sospeso. A causa di questo carico aggiuntivo, l'esecuzione di un metodo ricorsivo può essere più lenta di quella del corrispondente metodo iterativo. In alcuni casi, l'incremento del tempo di esecuzione per un particolare metodo ricorsivo è tale da spingere a evitare la ricorsione. Per esempio, non si dovrebbero mai calcolare ricorsivamente i numeri di Fibonacci, presentati nel Progetto di Programmazione 7 alla fine di questo capitolo. In altre situazioni la ricorsione è invece del tutto accettabile e può contribuire ad aumentare la comprensibilità del codice. Esistono, infatti, casi nei quali la ricorsione può migliorare sensibilmente la comprensibilità di un programma.

MyLab

LISTATO 7.3 Una versione iterativa di `scriviAParole`.

```
import java.util.Scanner;

public class IterativoDemo1 {
    public static void main(String[] args)
    <Il resto di main è lo stesso del Listato 7.2.>

    /**
    Precondizione: numero >= 0
    Scrive a parole le cifre del numero.
    */
```

```
    public static void scriviAParole(int numero) {
        int divisore = calcolaPotenzaDiDieci(numero);
        int prossimo = numero;
        while (divisore >= 10) {
            System.out.print(daCifraAParola(prossimo / divisore) + " ");
            prossimo = prossimo % divisore;
            divisore = divisore / 10;
        }
        System.out.print(daCifraAParola(prossimo / divisore) + " ");
    }

    // Precondizione: n >= 0.
    // Restituisce 10 elevato alla potenza n.
    public static int calcolaPotenzaDiDieci(int n) {
        int risultato = 1;
        while (n >= 10) {
            risultato = risultato * 10;
            n = n / 10;
        }
        return risultato;
    }

    public static String daCifraAParola(int cifra)
    <Il resto di daCifraAParola è lo stesso del Listato 7.2.>
}
```

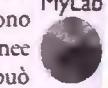
Esempio di output

L'output è esattamente lo stesso del Listato 7.2.

7.1.5 Metodi ricorsivi che restituiscono un valore

Un metodo ricorsivo può essere un metodo `void`, come visto in precedenza, o può restituire un valore. Le modalità di progettazione di un metodo che restituisce un valore sono essenzialmente le stesse valide per i metodi `void` e pertanto rimangono valide le linee guida presentate nel riquadro "DA RICORDARE: linee guida per la ricorsione". Si può integrare la seconda regola per tenere conto dei valori restituiti, aggiungendo la frase in corsivo nella versione della regola riportata qui di seguito.

MyLab



Video 7.2 Scrivere un metodo ricorsivo che restituisce un valore

Almeno una delle alternative dovrebbe contenere una chiamata ricorsiva del metodo *che produce il valore da restituire*. Queste chiamate ricorsive devono utilizzare argomenti in qualche modo "più piccoli" o risolvere versioni "ridotte" del compito realizzato dal metodo.

Il metodo ricorsivo `contaNumeroDiZeri`, definito nel Listato 7.4, restituisce un valore, pari al numero di zeri presenti nell'intero passato come argomento. Per esempio, `contaNumeroDiZeri(2030)` restituisce 2, perché 2030 contiene due cifre uguali a zero. Ecco come funziona questo metodo. La sua definizione si basa sulle seguenti considerazioni:

- se *n* è composto da una sola cifra, il numero di zeri in *n* è 1 se *n* è uguale a zero e 0 se *n* è diverso da zero;

- *se n è composto da due o più cifre, siano d la sua ultima cifra e m l'intero che si ottiene eliminando da n l'ultima cifra. Il numero di zeri in n è uguale al numero di zeri in m, più 1 se d è uguale a zero.*

Per esempio, il numero di zeri in 20030 è uguale al numero di zeri in 2003 più 1 per via dell'ultimo zero. Il numero di zeri in 20035 è pari al numero di zeri in 2003 senza che si debba aggiungere nulla, perché la cifra in più non è uno zero. Tenendo presente questa definizione, si può analizzare una semplice elaborazione che utilizzi `contaNumeroDiZeri`. Si consideri per prima cosa la semplice espressione

```
contaNumeroDiZeri(0)
```

che potrebbe comparire nella parte di destra di un'istruzione di assegnamento. Quando il metodo viene chiamato, il valore del parametro n è posto uguale a 0 e viene eseguito il codice presente nella definizione del metodo. Poiché n è uguale a 0, risulta soddisfatta la prima condizione del blocco `if-else` a più vie e viene restituito il valore 1.

Si consideri quindi un'altra semplice istruzione:

```
contaNumeroDiZeri(5)
```

MyLab

LISTATO 7.4 Un metodo ricorsivo che restituisce un valore.

```
import java.util.Scanner;

public class RicorsioneDemo2 {
    public static void main(String[] args) {
        System.out.println("Inserire un numero non negativo:");
        Scanner tastiera = new Scanner(System.in);
        int numero = tastiera.nextInt();
        System.out.println(numero + " contiene " +
                           contaNumeroDiZeri(numero) + " zeri.");
    }

    /**
    Precondizione: n >= 0
    Restituisce il numero di zeri in n.
    */
    public static int contaNumeroDiZeri(int n) {
        int risultato;
        if (n == 0)
            risultato = 1;
        else if (n < 10)
            risultato = 0; //n ha una sola cifra e questa non è 0
        else if (n % 10 == 0)
            risultato = contaNumeroDiZeri(n / 10) + 1;
        else //n % 10 != 0
            risultato = contaNumeroDiZeri(n / 10);
        return risultato;
    }
}
```

Esempio di output

```
Inserire un numero non negativo:
2008
2008 contiene 2 zeri.
```

Quando il metodo viene chiamato, il parametro n è posto uguale a 5 e viene eseguito il codice del metodo. Dato che n è diverso da 0, la prima condizione del blocco `if-else` a più vie non è soddisfatta. Tuttavia il valore di n è minore di 10, quindi è soddisfatta la seconda condizione del blocco `if-else` e il valore restituito è 0. Come si può vedere, questi due casi semplici vengono gestiti correttamente.

Viene ora affrontato un esempio che coinvolge una chiamata ricorsiva. Si consideri l'espressione

```
contaNumeroDiZeri(50)
```

Quando il metodo viene chiamato, il valore di n è posto uguale a 50 e viene eseguito il codice del metodo. Dato che n non è né uguale a 0, né minore di 10, nessuna delle prime due condizioni del blocco `if-else` è soddisfatta. Poiché `50 % 10` è uguale a 0, è invece soddisfatta la terza condizione. Quindi viene restituito il valore

```
contaNumeroDiZeri(n / 10) + 1
```

che in questo caso è equivalente a

```
contaNumeroDiZeri(50 / 10) + 1
```

equivalente a sua volta a

```
contaNumeroDiZeri(5) + 1
```

Si è visto prima che `contaNumeroDiZeri(5)` restituisce 0, pertanto il valore restituito da `contaNumeroDiZeri(50)` è 0 + 1, cioè 1, che è la risposta corretta.

Numeri più grandi produrranno sequenze di chiamate ricorsive più lunghe. Si consideri per esempio l'espressione

```
contaNumeroDiZeri(2008)
```

Il suo valore è calcolato come segue:

```
contaNumeroDiZeri(2008) è uguale a contaNumeroDiZeri(200) + 0
contaNumeroDiZeri(200) è uguale a contaNumeroDiZeri(20) + 1
contaNumeroDiZeri(20) è uguale a contaNumeroDiZeri(2) + 1
contaNumeroDiZeri(2) è uguale a 0 (caso base)
contaNumeroDiZeri(20) è uguale a contaNumeroDiZeri(2) + 1,
                          che vale 0 + 1, cioè 1
contaNumeroDiZeri(200) è uguale a contaNumeroDiZeri(20) + 1,
                          che vale 1 + 1, cioè 2
contaNumeroDiZeri(2008) è uguale a contaNumeroDiZeri(200) + 0,
                          che vale 2 + 0, cioè 2
```

Si noti che quando Java raggiunge il caso base `contaNumeroDiZeri(2)`, sono rimaste in sospeso tre elaborazioni. Dopo aver calcolato il valore da restituire per il caso base, viene ripristinata l'elaborazione sospesa più di recente, che produce il valore `contaNumeroDiZeri(20)`. Dopodiché vengono ripristinate una dopo l'altra tutte le altre elaborazioni rimaste in sospeso. Ogni risultato viene utilizzato in un'altra elaborazione sospesa, finché non viene completata l'elaborazione per la chiamata originale `contaNumeroDiZeri(2008)`. Le elaborazioni sono completate in ordine inverso rispetto a quello nel quale erano state sospese. Il valore restituito alla fine è 2, che è corretto perché 2008 contiene due zeri.

CASO DI STUDIO
UN ALTRO METODO PER IL CALCOLO DELLE POTENZE

Il Listato 7.5 presenta la definizione ricorsiva di un metodo per il calcolo delle potenze. Questo metodo è stato chiamato `potenza`. Per esempio, l'istruzione seguente imposta il valore di `risultato2` a 8, perché 2^3 è uguale a 8:

```
int risultato2 = potenza(2, 3);
```

Al di fuori della classe nella quale il metodo è inserito, l'istruzione andrebbe scritta come

```
int risultato2 = RicorsioneDemo3.potenza(2, 3);
```

La definizione del metodo potenza si basa sulla formula seguente:

x^n è uguale a $x^{n-1} * x$

La traduzione in Java di questa formula deve essere tale che il valore restituito da `potenza(x, n)` coincida con il valore dell'espressione

```
potenza(x, n - 1) * x
```

La definizione del metodo `potenza` presentata nel Listato 7.5 restituisce effettivamente questo valore per `potenza(x, n)`, a patto che sia `n > 0`.

Il caso in cui n è uguale a 0 è quello di arresto. Se n vale 0, `potenza(x, n)` restituisce semplicemente 1 (perché x^0 è uguale a 1).

Si può verificare cosa accade quando il metodo potenza viene chiamato passandogli alcuni valori di esempio. Si consideri per prima cosa l'espressione

```
potenza(2, 0)
```

Quando si chiama il metodo, il valore di `x` è 2, quello di `n` è 0 e viene eseguito il codice del metodo. Dato che il valore di `n` è tra quelli ammessi, viene eseguito il blocco `if-else`. Inoltre, poiché il valore di `n` non è maggiore di 0, viene eseguita l'istruzione `return` associata all'`else` e la chiamata al metodo restituisce 1. Pertanto, la seguente espressione imposterebbe a 1 il valore di `risultato3`:

```
int risultato3 = potenza(2, 0);
```

Si vedrà ora un esempio che implica una chiamata ricorsiva. Si consideri l'espressione

```
potenza(2, 1)
```

In corrispondenza della chiamata del metodo, il valore di x è 2, quello di n è 1 e viene eseguito il codice del metodo. Dato che ora il valore di n è maggiore di 0, si utilizza la seguente istruzione di `return` per determinare il valore da restituire:

```
return (potenza(x, n-1) * x);
```

che in questo caso è equivalente a

```
return (potenza(2, 0) * 2);
```

A questo punto l'elaborazione di `potenza(2, 1)` viene sospesa, una sua copia è salvata sullo *stack* e il computer inizia una nuova chiamata a metodo per calcolare il valore di `potenza(2, 0)`. Come si è già visto, il computer sostituisce l'espressione `potenza(2, 0)` con il suo valore 1 e ripristina l'elaborazione sospesa, che determina il valore finale di `potenza(2, 1)` nel modo seguente:

`potenza(2, 0) * 2` è uguale a `1 * 2`, cioè `2`

Quindi il valore finale restituito da `potenza(2, 1)` è 2. Di conseguenza, la seguente istruzione imposterebbe il valore di `risultato4` a 2:

```
int risultato4 = potenza(2, 1);
```

Numeri più grandi passati come secondo argomento genereranno sequenze più lunghe di chiamate ricorsive. Per esempio, si consideri l'istruzione

```
System.out.println(potenza(2, 3));
```

Il valore di `potenza(2, 3)` è calcolato come segue:

```
potenza(2, 3) è uguale a potenza(2, 2) * 2
    potenza(2, 2) è uguale a potenza(2, 1) * 2
        potenza(2, 1) è uguale a potenza(2, 0) * 2
            potenza(2, 0) è uguale a 1 (caso di arresto)
```

Quando il computer raggiunge il caso di arresto `potenza(2, 0)`, sono rimaste in sospeso tre elaborazioni. Dopo aver calcolato il valore da restituire nel caso di arresto, ripristina l'elaborazione sospesa più di recente per determinare il valore di `potenza(2, 1)`. Una volta fatto questo, il computer completa ognuna delle altre elaborazioni rimaste in sospeso, utilizzando ogni volta il valore così calcolato in una successiva elaborazione sospesa, finché non raggiunge e completa l'elaborazione per la chiamata originale `potenza(2, 3)`. I dettagli dell'elaborazione completa sono illustrati in Figura 7.3.

LISTATO 7.5 Il metodo ricorsivo potenza.

MyLab

```
public class RicorsioneDemo3 {
    public static void main(String args[]) {
        for (int n = 0; n < 4; n++)
```

```
    }

    public static int potenza(int x, int n) {
        if (n < 0) {
            System.out.println("Argomento non ammesso per potenza.");
            System.exit(0);
        }

        if (n > 0)
            return (potenza(x, n - 1) * x);
        else // n == 0
            return (1);
        }
}
```

Esempio di output

```
3 alla 0 e' uguale a 1
3 alla 1 e' uguale a 3
3 alla 2 e' uguale a 9
3 alla 3 e' uguale a 27
```

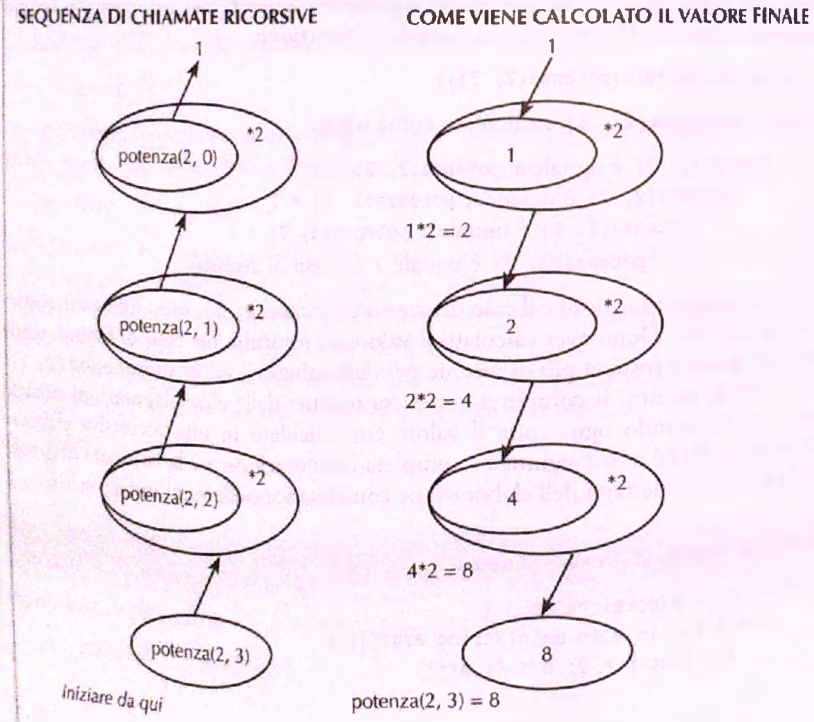


Figura 7.3 Elaborazione della chiamata ricorsiva `potenza(2, 3)`.

Non confondere ricorsione e *overloading*

Non bisogna confondere i termini ricorsione e *overloading*. L'*overloading* sarà trattato approfonditamente nel Capitolo 9. Al momento è sufficiente sapere che si effettua *overloading* di un metodo quando si definiscono due (o più) metodi che hanno lo stesso nome, ma differiscono per tipo, ordine e numero di parametri in ingresso. Se la definizione di uno dei due metodi include una chiamata all'altro, non si verifica alcuna ricorsione. La definizione di un metodo ricorsivo comprende al contrario una chiamata a quello stesso metodo, con la stessa precisa definizione, numero e tipo dei parametri inclusi.

7.2 Programmare utilizzando la ricorsione

In questo paragrafo vengono presentati alcuni programmi che illustrano l'uso della ricorsione.

7.2.1 Tecniche di progettazione ricorsiva

Quando si definiscono e si utilizzano i metodi ricorsivi, non si vogliono di certo tenere sempre presenti la gestione dello *stack* e le esecuzioni sospese. La potenza della ricorsione deriva proprio dal fatto che è possibile ignorare questi dettagli e lasciare che sia il computer a occuparsene. Si consideri l'esempio del metodo potenza del Listato 7.5. Il modo corretto di pensare alla definizione di potenza è il seguente:

`potenza(x, n)` restituisce `potenza(x, n – 1) * x`

Poiché x^n è uguale a x^{n-1} `*` `x`, questo è il valore corretto da restituire, a patto che l'elaborazione raggiunga sempre un caso di arresto e lo gestisca correttamente. Quindi, dopo aver verificato la correttezza della parte ricorsiva della definizione, è necessario controllare che la sequenza di chiamate ricorsive raggiunga sempre un caso di arresto e che esso restituisca sempre il valore giusto. In altre parole, tutto ciò che è necessario fare è assicurarsi che le seguenti tre proprietà siano soddisfatte:

- non si verifica ricorsione infinita (una chiamata ricorsiva potrebbe generare un'altra chiamata ricorsiva, la quale a sua volta potrebbe generarne un'altra e così via, ma qualunque sequenza di questo tipo raggiungerà alla fine un caso di arresto);
- ogni caso di arresto restituisce il valore corretto per quel caso;
- per i casi che coinvolgono la ricorsione: se tutte le chiamate ricorsive restituiscono il valore corretto, allora il valore finale restituito dal metodo è quello corretto.

Per esempio, consideriamo il metodo `potenza` del Listato 7.5.

1. **Non si verifica ricorsione infinita:** il secondo argomento di `potenza(x, n)` è decrementato di un'unità a ogni chiamata ricorsiva, quindi qualunque sequenza di chiamate ricorsive dovrà necessariamente raggiungere alla fine il caso `potenza(x, 0)`, che è il caso di arresto. Pertanto, non c'è possibilità di ricorsione infinita.
2. **Ogni caso di arresto restituisce il valore corretto per quel caso:** l'unico caso di arresto è `potenza(x, 0)`. Una chiamata a `potenza(x, 0)` restituisce sempre 1, che è il valore corretto per x^0. Quindi il caso di arresto restituisce il valore corretto.

3. **Per i casi che coinvolgono la ricorsione: se tutte le chiamate ricorsive restituiscono il valore corretto, allora il valore finale restituito dal metodo è quello corretto.** L'unico caso che coinvolge la ricorsione è quello nel quale n > 1. Quando n > 1, `potenza(x, n)` restituisce

   `potenza(x, n - 1) * x`

   Per vedere che questo è il valore corretto da restituire, si noti che: se `potenza(x, n - 1)` restituisce il valore corretto, allora `potenza(x, n - 1)` restituisce x^{n-1} e quindi `potenza(x, n)` restituisce

   x^{n-1} `* x`, che vale x^n

   e questo è il valore corretto per `potenza(x, n)`.

Questo è tutto ciò che serve verificare per essere sicuri che la definizione di `potenza` sia corretta (la tecnica precedentemente descritta è nota come *induzione matematica*, un concetto che potrebbe essere stato già incontrato in un corso di matematica. Tuttavia, non è necessario avere familiarità con l'espressione *induzione matematica* per utilizzare questa tecnica).

Sono stati forniti tre criteri da usare nella verifica di un metodo ricorsivo che restituisce un valore. Di fatto, le stesse regole possono essere applicate a un metodo ricorsivo `void`. Se si verifica che la definizione di un metodo ricorsivo `void` soddisfa i tre criteri seguenti, si avrà la garanzia che il metodo funzioni correttamente:

1. non si verifica ricorsione infinita;
2. ogni caso di arresto esegue le operazioni corrette per quel caso;
3. per i casi che coinvolgono la ricorsione: *se* tutte le chiamate ricorsive eseguono correttamente le loro azioni, *allora* l'intera elaborazione si svolge correttamente.

ESEMPIO DI PROGRAMMAZIONE
COSTRINGERE L'UTENTE A FORNIRE DATI CORRETTI

Il programma nel Listato 7.6 richiede semplicemente un numero intero positivo e conta alla rovescia dal numero fornito fino a 0 (zero). Il metodo `leggiNumero` legge l'intero fornito dall'utente. Si noti che se l'utente inserisce un numero non positivo, il metodo `leggiNumero` chiama se stesso. Questa chiamata fa ripartire dall'inizio la procedura di inserimento del dato. Se l'utente fornisce un altro numero non positivo, si verifica un'altra chiamata ricorsiva e la procedura di inserimento dati ripartirà dall'inizio un'altra volta. Questa sequenza si ripete finché l'utente non inserisce un intero positivo. Nell'utilizzo reale, naturalmente, non dovrebbero essere necessarie molte chiamate ricorsive, ma ne avverrà comunque una ogni volta che l'utente inserisce un dato non accettabile.

MyLab

LISTATO 7.6 Uso della ricorsione per ripartire dall'inizio.

```
import java.util.Scanner;

public class ContoAllaRovescia {
    public static void main(String[] args) {
```

```
        int valore = leggiNumero();
        mostraContoAllaRovescia(valore);
    }

    public static void leggiNumero() {
        System.out.println("Inserire un intero positivo:");
        Scanner tastiera = new Scanner(System.in);
        int conteggio = tastiera.nextInt();
        if (conteggio <= 0) {
            System.out.println("Il dato deve essere positivo.");
            System.out.println("Riprovare.");
            conteggio = leggiNumero();  //ripartenza
        }
        return conteggio;
    }

    public static void mostraContoAllaRovescia(int conteggio) {
        System.out.println("Conto alla rovescia:");
        for (int rimanenti = conteggio; rimanenti >= 0; rimanenti --)
            System.out.print(rimanenti + ", ");
        System.out.println("Decollo!");
    }
}
```

Esempio di output

```
Inserire un intero positivo:
0
Il dato deve essere positivo.
Riprovare.
Il dato deve essere positivo.
3
Conto alla rovescia:
3, 2, 1, 0, Decollo!
```

CASO DI STUDIO
RICERCA BINARIA

MyLab

Video 7.3 Usare la ricorsione per ricercare e ordinare gli elementi di un array

Questo caso di studio presume che si siano già studiate le basi sugli array presentate nel Capitolo 6. Verrà progettato un metodo ricorsivo che verifica se un numero dato sia presente o meno in un array di interi. Se il numero cercato è nell'array, il metodo fornirà anche l'indice corrispondente alla posizione del numero nell'array.

Per esempio, si supponga che l'array contenga l'elenco dei biglietti vincenti di una lotteria e che si voglia controllare tale elenco per verificare se si è vinto qualcosa. Si supponga inoltre che un altro array contenga l'ammontare dei premi per ogni biglietto, posti nello stesso ordine del primo array. Di conseguenza, se si conosce l'indice di un biglietto vincente, lo si può utilizzare per individuare l'ammontare del premio nel secondo array.

Nel paragrafo "Ricerca negli array" del Capitolo 6 è stato trattato un metodo di ricerca in un array che prevedeva semplicemente di controllare ogni posizione dell'array. Il metodo sviluppato in questo caso di studio risulterà molto più veloce della semplice ricerca sequenziale. Tuttavia, il corretto funzionamento del metodo più veloce richiede che l'array sia ordinato. Si presumerà che i numeri nell'array siano già stati disposti in ordine crescente e che l'array sia interamente popolato. Quindi, se `a` è il nome dell'array, si avrà che

$$a[0] \leq a[1] \leq a[2] \leq \ldots \leq a[a.length - 1]$$

Si tenga presente che la ricerca sequenziale non richiede questo prerequisito.

Si vuole progettare il metodo in modo che restituisca un intero corrispondente all'indice del numero cercato. Se il numero non è nell'array, sarà restituito il valore -1. Prima di passare alla costruzione completa della classe e dei metodi e di affrontare la questione di come collegare il metodo a un array, è utile progettare dello pseudocodice per risolvere il problema della ricerca.

Dato che l'array è ordinato, se ne possono escludere intere sezioni che sicuramente non possono contenere il numero cercato. Per esempio, se si sta cercando il numero 7 e si sa che `a[5]` contiene 9, se ne deduce, ovviamente, che 7 non è uguale al contenuto di `a[5]`. Ma in realtà se ne può dedurre molto di più: essendo l'array ordinato, si sa anche che

$$7 < a[5] \leq a[i] \text{ per } i \geq 5$$

Si sa quindi che 7 non può essere uguale a `a[i]` per nessun valore di `i` maggiore o uguale a 5. Pertanto non è necessario effettuare la ricerca tra gli elementi `a[i]` con $i \geq 5$: si sa già che il valore cercato 7 non è tra questi senza nemmeno doverli controllare.

Allo stesso modo, se 7, il numero da cercare, fosse maggiore di `a[5]` (per esempio se `a[5]` fosse 3 invece di 9), si potrebbero escludere subito tutti gli elementi `a[i]` con $i \leq 5$.

Queste osservazioni verranno ora tradotte in una versione preliminare dell'algoritmo, sostituendo l'indice 5 utilizzato nell'esempio con il generico `m`.

Algoritmo per la ricerca dell'elemento obiettivo in un array ordinato (versione preliminare n. 1)

```
1.  m = un indice tra 0 e (a.length - 1)
2.  if (obiettivo == a[m])
3.      return m;
4.  else if (obiettivo < a[m])
5.      return il risultato della ricerca tra gli elementi da a[0] a a[m - 1]
6.  else if (obiettivo > a[m])
7.      return il risultato della ricerca tra gli elementi da a[m + 1] a a[a.length - 1]
```

Se `a[m]` non è l'elemento cercato, si può ignorare una parte dell'array e cercare solo nell'altra. La scelta di `m` determina le dimensioni delle due parti. Dato che non si sa a priori quale delle due parti contenga `a[m]`, tanto vale considerare due parti di dimensioni il più possibile simili. Per fare questo, si sceglie `m` in modo che `a[m]` si trovi all'incirca a metà dell'array. Si sostituisce quindi il punto 1 dello pseudocodice riportato sopra con

```
m = punto medio approssimato tra 0 e (a.length - 1)
```

Per rendere più chiaro l'algoritmo, è utile rinominare l'indice `m` in `med`.

Si noti che ognuna delle due alternative `if-else` ricerca in un segmento dell'array. Tale ricerca è una versione ridotta dell'intera operazione che si sta progettando, il che suggerisce di utilizzare la ricorsione. La ricerca nei due segmenti dell'array può, infatti, essere realizzata tramite chiamate ricorsive dell'algoritmo stesso. Nonostante sia necessario prevedere due chiamate ricorsive nell'algoritmo, una sola delle due verrà eseguita nell'ambito di una data ricerca, perché verrà effettuata solo una delle due ricerche. Quindi, se `a[med]` non è l'elemento cercato, si cercherà tra gli elementi

da `a[0]` a `a[med - 1]`

oppure

da `a[med + 1]` a `a[a.length - 1]`

C'è però una complicazione. Per implementare queste chiamate ricorsive servono dei parametri aggiuntivi: è necessario specificare che la ricerca deve essere eseguita solo in una parte dell'array. Nel primo caso si tratta della parte costituita dagli elementi con indice tra `0` e `med - 1`, mentre il secondo caso coinvolge gli elementi con indice tra `med + 1` e `a.length - 1`. Servono pertanto altri due parametri per specificare il primo e l'ultimo indice della parte di array nella quale compiere la ricerca. Siano `primo` e `ultimo` i nomi di questi due nuovi parametri. A questo punto si può rendere più preciso lo pseudocodice come segue.

Algoritmo per la ricerca dell'elemento obiettivo in un array ordinato (versione preliminare n. 2)

```
1.  med = punto medio approssimato tra primo e ultimo
2.  if (obiettivo == a[med])
3.      return med;
4.  else if (obiettivo < a[med])
5.      return il risultato della ricerca tra gli elementi da a[primo] a a[med - 1]
6.  else if (obiettivo > a[med])
7.      return il risultato della ricerca tra gli elementi da a[med + 1] a a[ultimo]
```

Per effettuare la ricerca nell'array completo basterà impostare all'inizio `primo` a `0` e `ultimo` a `a.length - 1`. Ogni chiamata ricorsiva utilizzerà invece valori diversi di `primo` e `ultimo`. Per esempio, se la prima chiamata ricorsiva avviene al passo 5, imposterà `primo` a `0` e `ultimo` a `med - 1`.

Bisogna sempre assicurarsi che un algoritmo ricorsivo non generi una ricorsione infinita verificando che ogni possibile chiamata dell'algoritmo porti a un caso base. Si considerino le tre alternative nel blocco `if-else` annidato. Nel primo caso, il numero cercato si trova in `a[med]` e non è richiesta alcuna chiamata ricorsiva, per cui il procedimento termina, essendo stato raggiunto il caso base. In ognuna delle altre due alternative, una chiamata ricorsiva effettua la ricerca in un sottoinsieme più piccolo dell'array. Se il numero cercato è nell'array, l'algoritmo restringerà sempre più l'ambito della ricerca finché il numero non viene trovato. Ma cosa accade se il numero non è nell'array? La serie di chiamate ricorsive porterà a un caso base anche se il numero non è nell'array? Sfortunatamente, no. Il problema non è difficile da individuare. Si noti che, in ogni

chiamata ricorsiva, o viene incrementato il valore di `primo` o viene decrementato il valore di `ultimo`. Se a un certo punto uno di questi due indici oltrepassa l'altro, così che `primo` diventa di fatto maggiore di `ultimo`, si può concludere che non sono rimasti più indici da controllare e quindi il numero `obiettivo` non è nell'array. Aggiungendo questo controllo allo pseudocodice, si ottiene la seguente versione, più completa.

Algoritmo per la ricerca dell'elemento obiettivo in un array ordinato (versione finale)

```
1.  med = punto medio approssimato tra primo e ultimo
2.  if (primo > ultimo)
3.      return -1;
4.  else if (obiettivo == a[med])
5.      return med;
6.  else if (obiettivo < a[med])
7.      return il risultato della ricerca tra gli elementi da a[primo] a a[med - 1]
8.  else if (obiettivo > a[med])
9.      return il risultato della ricerca tra gli elementi da a[med + 1] a a[ultimo]
```

Questa procedura per la ricerca in un array è detta **ricerca binaria**. Nel corso della ricerca, l'algoritmo esclude dapprima metà dell'array, poi metà di ciò che è rimasto e così via. La Figura 7.4 mostra un esempio di come funziona questo algoritmo.

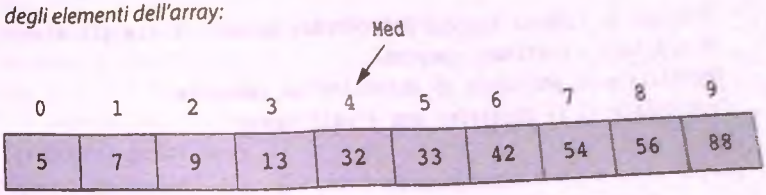
A questo punto è necessario tradurre lo pseudocodice in codice Java. Sia `ricercaBinaria` il nome del metodo che implementa l'algoritmo. Poiché il metodo `ricercaBinaria` richiede, oltre all'array su cui effettuare la ricerca e il vaore obiettivo da trovare, anche dei parametri aggiuntivi, l'utente dovrebbe impostarli esplicitamente a `0` e `a.length - 1` per specificare che la ricerca va effettuata nell'array completo. Per evitare di dover tenere a mente questo dettaglio, si aggiunge il metodo `trova`, che si limita a chiamare il metodo `ricercaBinaria`, permettendo all'utente di specificare il valore `obiettivo` senza doversi preoccupare degli indici.

Il codice completo di questi due metodi è presentato nel Listato 7.7. Nel Listato 7.8 è riportato un semplice programma che mostra come funziona questo metodo.

L'algoritmo di ricerca binaria è estremamente veloce. Esclude già in partenza circa metà dell'array dalla ricerca, dopodiché ne elimina un altro quarto, poi un altro ottavo e così di seguito. Come conseguenza di questo processo di esclusione, la maggior parte dell'array non deve nemmeno essere presa in considerazione, con un grande risparmio di tempo. Per esempio, per un array da 1000 elementi la ricerca binaria dovrà confrontare il valore obiettivo solo con 10 elementi circa dell'array. Una semplice ricerca sequenziale potrebbe essere costretta a confrontare il valore obiettivo con tutti i 1000 elementi e, in media, dovrà effettuare 500 confronti. La ricerca in un array di un milione di elementi costituisce un esempio ancora più efficace: la ricerca binaria richiederà al massimo 20 confronti, mentre una ricerca sequenziale comporterebbe, in media, 500.000 confronti, ma potrebbe anche richiederne un milione.

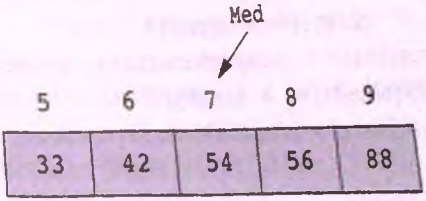
obiettivo è uguale a 33

Elimina metà degli elementi dell'array:



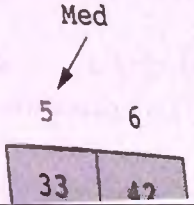
1. med = (0 + 9)/2 (uguale a 4)
2. 33 > a[med] (cioè 33 > a[4])
3. Quindi se 33 è nell'array, 33 è uno tra a[5], a[6], a[7], a[8], a[9].

Elimina metà degli elementi rimanenti dell'array:



1. med = (5 + 9)/2 (uguale a 7)
2. 33 < a[med] (cioè 33 < a[7])
3. Quindi se 33 è nell'array, 33 è uno tra a[5], a[6].

Elimina metà degli elementi rimanenti dell'array:



LISTATO 7.7 Metodi per la ricerca binaria.

```
/**
Utilizza la ricerca binaria per cercare obiettivo tra gli elementi
da a[primo] a a[ultimo] compresi.
Restituisce la posizione di obiettivo se presente.
Restituisce -1 se obiettivo non è nell'array.
*/
public static int ricercaBinaria (int[] a, int obiettivo, int primo,
        int ultimo) {
    int risultato;
    if (primo > ultimo)
        risultato = -1;
    else {
        int med = (primo + ultimo) / 2;
        if (obiettivo == a[med])
            risultato = med;
        else if (obiettivo < a[med])
            risultato = ricercaBinaria(a, obiettivo, primo, med - 1);
        else //(obiettivo > a[med])
            risultato = ricercaBinaria(a, obiettivo, med + 1, ultimo);
    }
    return risultato;
}

/**
Precondizione: array è pieno e ordinato
in senso crescente.
Se obiettivo è nell'array, restituisce l'indice corrispondente.
Restituisce -1 se obiettivo non è nell'array.
*/
public static int trova(int[] a, int obiettivo) {
    return ricercaBinaria(a, obiettivo, 0, a.length - 1);
}
```

MyLab

LISTATO 7.8 Un esempio di ricerca binaria.

```
import java.util.Scanner;

public class EsempioRicercaArray {

    < Metodi del Listato 7.7 >

    public static void main(String[] args) {
        int [] unArray = new int[10];
        Scanner tastiera = new Scanner(System.in);
```

```
        System.out.println("Inserire 10 interi in ordine crescente,");
        System.out.println("uno per riga.");
        for (int i = 0; i < 10; i++)
            unArray [i] = tastiera.nextInt();
        System.out.println();
        for (int i = 0; i < 10; i++)
            System.out.print("a[" + i + "]=" + unArray [i] + " ");
        System.out.println();
        System.out.println();

        String risposta;
        do {
            System.out.println("Inserire un valore da cercare:");
            int obiettivo = tastiera.nextInt();
            int risultato = trova(unArray, obiettivo);
            if (risultato < 0)
                System.out.println(obiettivo +
                            " non è nell'array.");
            else
                System.out.println(obiettivo + " è alla posizione " +
                                            risultato);
            System.out.println("Nuova ricerca?");
            risposta = tastiera.next();
        } while (risposta.equalsIgnoreCase("si"));
        System.out.println(
                "Possa tu trovare ciò che stai cercando.");
    }
}
```

Esempio di output

```
Inserire 10 interi in ordine crescente,
uno per riga.
0
2
4
6
8
10
12
14
16
18
a[0]=0 a[1]=2 a[2]=4 a[3]=6 a[4]=8 a[5]=10 a[6]=12 a[7]=14 a[8]=16 a[9]=18
Inserire un valore da cercare:
14
14 è alla posizione 7
Nuova ricerca?
si
```

```
Inserire un valore da cercare:
0
0 è alla posizione 0
Nuova ricerca?
si
Inserire un valore da cercare:
2
2 è alla posizione 1
Nuova ricerca?
si
Inserire un valore da cercare:
13
13 non è nell'array.
Nuova ricerca?
no
Possa tu trovare ciò che stai cercando.
```

Generalizzare il problema

Quando si progetta un algoritmo ricorsivo, risulta spesso necessario risolvere un problema più generale di quello di partenza. Per esempio, si consideri il metodo `ricercaBinaria` nel caso di studio precedente: è stato progettato per poter effettuare la ricerca non solo nell'array completo, ma anche in una sua parte. Ciò si è rivelato necessario per permettere di trattare i casi ricorsivi. Accade spesso, nel progetto di un algoritmo ricorsivo, che si debba considerare un problema un po' più generale per poter trattare più facilmente i casi ricorsivi.

ESEMPIO DI PROGRAMMAZIONE
MERGE SORT – UN METODO DI ORDINAMENTO RICORSIVO

Il modo più semplice di descrivere alcuni degli algoritmi di ordinamento più efficienti è sotto forma di algoritmi ricorsivi. Un esempio è rappresentato dall'algoritmo di **merge sort** (ordinamento con fusione) per l'ordinamento di un array. In questo esempio verrà presentato un metodo che implementa l'algoritmo di *merge sort* per ordinare un array di valori `int` in senso crescente. Da questo, tramite modifiche minime, si possono ottenere metodi per ordinare array di elementi di qualunque tipo permetta un ordinamento.
Il *merge sort* è un esempio di algoritmo *divide et impera*. L'array da ordinare viene diviso a metà e le due metà vengono ordinate tramite chiamate ricorsive, ottenendo due array ordinati più corti. Questi due array vengono quindi fusi per formare un singolo array ordinato. In forma schematica, l'algoritmo ha la seguente struttura.

Algoritmo di merge sort per l'ordinamento dell'array a:

1. Se l'array a ha solo un elemento, non c'è niente da fare (caso base). Altrimenti, procedere come segue (caso ricorsivo):
2. Copiare la prima metà degli elementi di a in un array più corto chiamato `primaMeta`.
3. Copiare il resto degli elementi di a in un altro array più corto `ultimaMeta`.
4. Ordinare l'array `primaMeta` usando una chiamata ricorsiva.
5. Ordinare l'array `ultimaMeta` usando una chiamata ricorsiva.
6. Unire gli elementi degli array `primaMeta` e `ultimaMeta` nell'array a.

L'implementazione della maggior parte dei dettagli dell'algoritmo in codice Java è immediata. La fusione dei due array `primaMeta` e `ultimaMeta` in un unico array ordinato richiede però qualche spiegazione.

L'idea di base dell'algoritmo di fusione è la seguente: gli array `primaMeta` e `ultimaMeta` sono entrambi ordinati in senso crescente. Quindi, il minore tra gli elementi di `primaMeta` è `primaMeta[0]` e il minore tra gli elementi di `ultimaMeta` è `ultimaMeta[0]`. Di conseguenza, il minore tra tutti gli elementi dei due array è il minore tra `primaMeta[0]` e `ultimaMeta[0]`. Si copia quell'elemento minore di tutti gli altri in `a[0]`.

Per esempio, si supponga che il minore dei due elementi sia `ultimaMeta[0]`, che viene quindi copiato in `a[0]`. Il minore tra gli elementi che rimangono da copiare in a sarà il minore tra `primaMeta[0]` e `ultimaMeta[1]`. Si copia quell'elemento in `a[1]` e si continua con la procedura.

Il codice Java per il procedimento di fusione avrà all'incirca la forma seguente:

```
int indicePrimaMeta = 0, indiceUltimaMeta = 0, indiceA = 0;
while (qualche_condizione) {
    if (primaMeta[indicePrimaMeta] < ultimaMeta[indiceUltimaMeta]) {
        a[indiceA] = primaMeta[indicePrimaMeta];
        indiceA++;
        indicePrimaMeta++;
    } else {
        a[indiceA] = ultimaMeta[indiceUltimaMeta];
        indiceA++;
        indiceUltimaMeta++;
    }
}
```

Ora, qual è la condizione da utilizzare nel ciclo `while`? Si noti che il ciclo non ha senso, a meno che gli array `primaMeta` e `ultimaMeta` non contengano ancora entrambi altri elementi da copiare. Quindi, anziché eseguire il ciclo fino a quando a non è pieno, basta eseguirlo fino a quando tutti gli elementi di uno dei due array `primaMeta` e

`ultimaMeta` sono stati copiati in a. Di conseguenza, la condizione per il ciclo `while` può essere:

```
while ((indicePrimaMeta < primaMeta.length) &&
    (indiceUltimaMeta < ultimaMeta.length))
```

Quando questo ciclo `while` termina, tutti gli elementi di uno degli array `primaMeta` e `ultimaMeta` sono stati copiati nell'array a, mentre l'altro array potrebbe ancora contenere (ed è anzi probabile che sia così) altri elementi da copiare in a. Questi elementi sono già ordinati e sono tutti maggiori di tutti gli elementi già in a, per cui è sufficiente copiare in a tutti gli elementi rimasti in `primaMeta` o in `secondaMeta`.

Il Listato 7.9 mostra l'implementazione Java completa dell'algoritmo di *merge sort*, incluso il metodo `merge`. Il Listato 7.10 contiene un programma di esempio che usa i metodi del Listato 7.9.

L'algoritmo di *merge sort* è molto più efficiente del *selection sort* presentato nel Capitolo 6. In effetti, non esiste alcun algoritmo di ordinamento la cui efficienza sia "ordini di grandezza" migliore del *merge sort*. La discussione del significato preciso di ciò che si intende con "ordini di grandezza" esulerebbe dall'argomento di questo libro, ma se ne può comunque dare un'idea intuitiva. Esistono in realtà algoritmi che sono, di fatto, più efficienti del *merge sort*. Tuttavia, per array molto grandi, tutti questi algoritmi (*merge sort* compreso) sono talmente più veloci del *selection sort* che il guadagno rispetto a quest'ultimo è molto più consistente delle minime differenze esistenti tra i vari algoritmi più veloci.

LISTATO 7.9 Metodi per il *merge sort*.

```
/**
Precondizione: ogni posizione dell'array a contiene un valore.
Postcondizione: a[0] <= a[1] <= ... <= a[a.length - 1].
*/
public static void ordina(int[] a) {
    if (a.length >= 2) {
        int metaLunghezza = a.length / 2;
        int[] primaMeta = new int[metaLunghezza];
        int[] ultimaMeta = new int[a.length - metaLunghezza];
        dividi(a, primaMeta, ultimaMeta);
        ordina(primaMeta);
        ordina(ultimaMeta);
        merge(a, primaMeta, ultimaMeta);
    }
    //altrimenti non fare niente. a.length == 1, quindi a è
    //già ordinato.
}

/**
Precondizione: a.length = primaMeta.length + ultimaMeta.length.
Postcondizione: Gli elementi di a sono divisi
tra gli arrays primaMeta e ultimaMeta.
*/
```

```
public static void dividi(int[] a, int[] primaMeta, int[] ultimaMeta) {
    for (int i = 0; i < primaMeta.length; i++)
        primaMeta[i] = a[i];
    for (int i = 0; i < ultimaMeta.length; i++)
        ultimaMeta[i] = a[primaMeta.length + i];
}

/**
Precondizione: Gli array primaMeta e ultimaMeta sono ordinate in
senso crescente; a.length = primaMeta.length + ultimaMeta.length.
Postcondizione: L'array a contiene tutti gli elementi di primaMeta
e ultimaMeta ed è ordinato in senso crescente.
*/
public static void merge(int[] a, int[] primaMeta, int[] ultimaMeta) {
    int indicePrimaMeta = 0, indiceUltimaMeta = 0, indiceA = 0;
    while ((indicePrimaMeta < primaMeta.length) &&
           (indiceUltimaMeta < ultimaMeta.length)) {
        if (primaMeta [indicePrimaMeta] < ultimaMeta [indiceUltimaMeta]) {
            a[indiceA] = primaMeta[indicePrimaMeta];
            indicePrimaMeta++;
        } else {
            a[indiceA] = ultimaMeta[indiceUltimaMeta];
            indiceUltimaMeta++;
        }
        indiceA ++;
    }

    /*
    Almeno uno tra primaMeta e ultimaMeta è stato copiato
    completamente in a.
    Copiare il resto di primaMeta, se ne è rimasto.
    */
    while (indicePrimaMeta < primaMeta.length) {
        a[indiceA] = primaMeta[indicePrimaMeta];
        indiceA++;
        indicePrimaMeta ++;
    }
    // Copiare il resto di ultimaMeta, se ne è rimasto.
    while (indiceUltimaMeta < ultimaMeta.length) {
        a[indiceA] = ultimaMeta[indiceUltimaMeta];
        indiceA++;
        indiceUltimaMeta++;
    }
}
```

MyLab

LISTATO 7.10 Esempio d'uso dei metodi per il *merge sort*.

```
public class EsempioMergeSort {

    < Metodi del Listato 7.9 >

    public static void main(String[] args) {
        int[] unArray = {7, 5, 11, 2, 16, 4, 18, 14, 12, 30};
        System.out.println("Valori dell'array prima dell'ordinamento");
        for (int i = 0; i < unArray.length; i++)
            System.out.print(unArray [i] + " ");
        System.out.println();
        ordina(unArray);
        System.out.println("Valori dell'array dopo l'ordinamento");
        for (int i = 0; i < unArray.length; i++)
            System.out.print(unArray [i] + " ");
        System.out.println();
    }
}
```

Esempio di output

```
Valori dell'array prima dell'ordinamento
7 5 11 2 16 4 18 14 12 30
Valori dell'array dopo l'ordinamento
2 4 5 7 11 12 14 16 18 30
```

7.3 Riepilogo

- Se la definizione di un metodo comprende una chiamata del metodo stesso, tale chiamata è detta ricorsiva. Le chiamate ricorsive sono ammesse in Java e in alcuni casi possono rendere più chiara la definizione di un metodo.
- Ogni volta che un algoritmo prevede un compito parziale che costituisce una versione ridotta del compito svolto dall'algoritmo completo, esso può essere implementato tramite un metodo Java ricorsivo.
- Al fine di evitare ricorsioni infinite, la definizione di un metodo ricorsivo dovrebbe comprendere due tipi di casi: uno o più casi che includano chiamate ricorsive e uno o più casi base (o di arresto) che non implichino chiamate ricorsive.
- Due ottimi esempi di algoritmi ricorsivi sono quello per la ricerca binaria e il *merge sort*.

7.4 Esercizi

1. Quale risultato sarà prodotto dal codice seguente?

```
public class Esempio {
    public static void main(String args[]) {
        System.out.println("Il risultato e':");
        metodo(23);
    }
    public static void metodo(int numero) {
        if (numero > 0) {
            metodo(numero / 2);
            System.out.println(numero % 2);
        }
    }
}
```

2. Quale risultato sarà prodotto dal codice seguente?

```
public class Esempio {
    public static void main(String args[]) {
        System.out.println("Il risultato e':");
        metodo(11156);
        System.out.println();
    }
    public static void metodo(int numero) {
        if (numero > 0) {
            int d = numero % 10;
            boolean dispari = (numero / 10) % 2 == 1;
            metodo(numero / 10);
            if (dispari)
                System.out.print(d / 2 + 5);
            else
                System.out.print(d / 2);
        }
    }
}
```

3. Scrivere un metodo ricorsivo che conti il numero di cifre dispari in un numero.
4. Scrivere un metodo ricorsivo che calcoli la somma delle cifre di un numero positivo.
5. Completare la definizione ricorsiva del metodo seguente:

```
/**
   Precondizione: n >= 0
   Restituisce 10 elevato alla potenza n.
*/
public static int calcolaDieciAlla(int n);
```

Sfruttare le seguenti proprietà di x^n:

$x^n=(x^{n/2})^2$ quando n è un numero positivo pari

$x^n=x(x^{(n-1)/2})^2$ quando n è un numero positivo dispari

$x^0=1$

6. Scrivere un metodo ricorsivo che calcoli la somma di tutti gli elementi di un array.
7. Scrivere un metodo ricorsivo che trovi e restituisca il valore più grande in un array di interi. *Suggerimento:* dividere l'array a metà e cercare ricorsivamente il valore massimo in ogni metà. Restituire il maggiore tra i due valori.
8. Scrivere un algoritmo ricorsivo di ricerca ternaria che divida un array in tre parti anziché le due utilizzate dalla ricerca binaria.
9. Scrivere un algoritmo ricorsivo che calcoli somme cumulative in un array. Per trovare le somme cumulative, si sommi a ogni elemento dell'array la somma degli elementi che lo precedono. Per esempio, se gli elementi dell'array fossero [2, 3, 1, 5, 6, 2, 7], il risultato dovrebbe essere [2, (2) + 3, (2 + 3) + 1, (2 + 3 + 1) + 5, (2 + 3 + 1 + 5) + 6, (2 + 3 + 1 + 5 + 6) + 2, (2 + 3 + 1 + 5 + 6 + 2) + 7], cioè [2, 5, 6, 11, 17, 19, 26]. *Suggerimento:* le somme riportate tra parentesi nell'esempio sono i risultati di chiamate ricorsive.
10. Si supponga di voler calcolare l'ammontare del denaro depositato su un conto bancario a interesse composto. Se m è la somma depositata sul conto, la somma disponibile alla fine del mese sarà 1.005 m. Scrivere un metodo ricorsivo che calcoli la somma presente sul conto dopo t mesi data una somma di partenza m.
11. Si supponga di avere un satellite in orbita. Per comunicare con il satellite possono essere inviati messaggi composti da due segnali: punto e linea. La trasmissione di un punto richiede 2 microsecondi, mentre quella della linea ne richiede 3. Si immagini di voler conoscere il numero $M(k)$ di messaggi distinti che possono essere inviati in k microsecondi.
    - Se k vale 0 o 1, si può trasmettere un messaggio (il messaggio vuoto)
    - Se k vale 2 o 3, si può trasmettere un messaggio (punto o linea, rispettivamente)
    - Se k è maggiore di 3, si sa che il messaggio può iniziare con un punto o con una linea. Se inizia con un punto, il numero di messaggi possibili è $M(k-2)$. Se il messaggio inizia con una linea, il numero di messaggi è $M(k-3)$. Quindi il numero di messaggi distinti che possono essere inviati in k microsecondi è $M(k-2) + M(k-3)$.

    Si scriva un programma che legga da tastiera un numero k e mostri il valore di $M(k)$, calcolato tramite un metodo ricorsivo.
12. Si scriva un metodo ricorsivo che conti il numero di vocali in una stringa. *Suggerimento:* ogni volta che si effettua una chiamata ricorsiva, si utilizzi il metodo `substring` della classe `String` per ottenere una nuova stringa formata dai caratteri compresi tra il secondo e l'ultimo della stringa originale. L'ultima chiamata ricorsiva avverrà quando la stringa non contiene caratteri.

13. Si scriva un metodo ricorsivo che elimini tutte le vocali da una stringa data e restituisca sotto forma di una nuova stringa tutto quello che rimane. *Suggerimento:* utilizzare l'operatore + per realizzare la concatenazione di stringhe al fine di costruire la stringa da restituire.
14. Si scriva un metodo ricorsivo che duplichi ogni carattere in una stringa e restituisca il risultato sotto forma di una nuova stringa. Per esempio, se l'argomento è `"libro"`, il risultato dovrebbe essere `"lliibbrroo"`.
15. Si scriva un metodo ricorsivo che inverta l'ordine dei caratteri in una stringa data e restituisca il risultato sotto forma di una nuova stringa. Per esempio, se l'argomento è `"libro"` il risultato dovrebbe essere `"orbil"`.

7.5 Progetti

1. Scrivere un metodo statico ricorsivo che restituisca il numero di cifre del numero passatogli come argomento di tipo `int`. Sono permessi sia valori positivi che negativi dell'argomento. Per esempio, -120 ha tre cifre. Non si tenga conto di eventuali zeri non significativi. Includere il metodo in un programma e verificarne il funzionamento.
2. Scrivere un metodo statico ricorsivo che restituisca la somma degli interi contenuti nell'array passatogli come unico argomento. Si presuma che ogni posizione dell'array contenga un valore. Includere il metodo in un programma e verificarne il funzionamento.
3. Uno degli esempi tipici di uso della ricorsione è rappresentato dall'algoritmo per il calcolo del **fattoriale** di un numero intero. La notazione per il fattoriale del numero intero n è $n!$ e la sua definizione è la seguente:

   0! è uguale a 1

   1! è uguale a 1

   2! è uguale a $2 \times 1 = 2$

   3! è uguale a $3 \times 2 \times 1 = 6$

   4! è uguale a $4 \times 3 \times 2 \times 1 = 24$

   ...

   $n!$ è uguale a $n \times (n - 1) \times (n - 2) \times \ldots \times 3 \times 2 \times 1$

   Un modo alternativo per descrivere il calcolo di $n!$ è la formula ricorsiva $n \times (n - 1)!$, più il caso base 0!, che vale 1. Scrivere un metodo statico che implementi questa definizione ricorsiva del fattoriale. Includere il metodo in un programma di prova che consenta all'utente di inserire valori di n finché non richiede la fine dell'esecuzione.
4. Un esempio tipico di formula ricorsiva è quella per calcolare la somma dei primi n numeri interi: $1 + 2 + 3 + \ldots + n$. La formula ricorsiva può essere scritta come

   $1 + 2 + 3 + \ldots + n = n + (1 + 2 + 3 + \ldots + (n - 1))$

Scrivere un metodo statico che implementi questa formula ricorsiva per calcolare la somma dei primi n interi. Includere il metodo in un programma di prova che consenta all'utente di inserire valori di n finché non segnala la fine dell'esecuzione. Non si usi un ciclo per sommare i primi n interi.

5. Un palindromo è una stringa che non cambia anche se letta al contrario, come `"radar"`. Si scriva un metodo statico ricorsivo che accetti un parametro di tipo `String` e restituisca `true` se l'argomento è un palindromo e `false` altrimenti. Non si tenga conto di spazi e segni di punteggiatura nella stringa, e non si faccia distinzione tra lettere maiuscole e minuscole. Per esempio, il metodo dovrebbe considerare palindrome le stringhe seguenti:

```
"I topi non avevano nipoti"
"A Roma trasalì la sarta mora"
```

Non è necessario che il metodo controlli che la stringa contenga una parola o frase corretta in italiano. Includere il metodo in un programma e verificarne il funzionamento.

6. Una **progressione geometrica** è definita come il prodotto dei primi n numeri interi e si indica con la notazione

$$\text{geometrica}(n) = \prod_{i=1}^{n} i$$

dove questa notazione indica che bisogna moltiplicare tra loro gli interi da 1 a n. Una **progressione armonica** è definita come il prodotto dei reciproci dei primi n interi e si indica con la notazione

$$\text{armonica}(n) = \prod_{i=1}^{n} \frac{1}{i}$$

Entrambe le espressioni ammettono una definizione ricorsiva equivalente:

$$\text{geometrica}(n) = n \times \prod_{i=1}^{n-1} i$$

$$\text{armonica}(n) = \frac{1}{n} \times \prod_{i=1}^{n-1} \frac{1}{i}$$

Scrivere dei metodi statici che implementino queste formule ricorsive per calcolare `geometrica(n)` e `armonica(n)`. Non si dimentichi il caso base, non incluso in queste formule, ma da determinare. Si inseriscano i metodi in un programma di prova che consenta all'utente di calcolare sia `geometrica(n)` che `armonica(n)` dato un intero n. Il programma dovrebbe permettere all'utente di inserire un altro valore di n dopo il primo e ripetere il calcolo finché non viene chiesto di interrompere il programma. Nessuno dei due metodi dovrebbe utilizzare cicli per moltiplicare gli n numeri.

7. La **sequenza di Fibonacci** compare spesso in natura sotto forma di tasso di crescita per certe popolazioni animali idealizzate. La sequenza di Fibonacci inizia con 0 e 1 e ogni numero successivo è la somma dei due precedenti. I primi dieci numeri di Fibonacci sono quindi 0, 1, 1, 2, 3, 5, 8, 13, 21 e 34. Il terzo numero della sequenza è 0 + 1, cioè 1; il quarto è 1 + 1, cioè 2; il quinto è 1 + 2, cioè 3 e così via.

Oltre a descrivere la crescita di certe popolazioni, la sequenza può essere utilizzata per definire la forma di una spirale. Inoltre, i rapporti di numeri di Fibonacci successivi tendono a una costante, pari a circa 1,618, detta "sezione aurea". Questo rapporto è così piacevole esteticamente che viene usato per stabilire i rapporti tra lunghezza e larghezza di stanze o cartoline.

Utilizzare una formula ricorsiva per definire un metodo statico per il calcolo dell'*n*-esimo numero di Fibonacci, dato *n*. Il metodo non deve utilizzare un ciclo per calcolare tutti i numeri di Fibonacci fino a quello richiesto, ma dovrebbe utilizzare la ricorsione. Includere questo metodo in un programma che mostri la convergenza della successione dei rapporti tra numeri di Fibonacci successivi. Il programma chiederà all'utente di specificare quanti numeri di Fibonacci debbano essere calcolati e li mostrerà uno per riga. Dopo le prime due righe, mostrerà su ciascuna riga anche il rapporto tra il numero di Fibonacci corrente e quello precedente (i due rapporti iniziali non sono significativi). Il risultato dovrebbe essere qualcosa di simile a quanto segue, se l'utente inserisce il valore 5:

```
Fibonacci #1 = 0
Fibonacci #2 = 1
Fibonacci #3 = 1; 1 / 1 = 1
Fibonacci #4 = 2; 2 / 1 = 2
Fibonacci #5 = 3; 3 / 2 = 1.5
```

3. Si consideri una barretta di cioccolato che può essere tagliata in k punti e si supponga di voler sapere quante sono le possibili sequenze di tagli che la dividono nei singoli pezzi. Per esempio, se k vale 3 si potrebbe tagliare la barretta nel punto 1, poi nel punto 2 e infine nel punto 3. Indichiamo questa sequenza di tagli come 123. Quindi, se k è uguale a 3, esistono sei modi di tagliare la barretta: 123, 132, 213, 231, 312 e 321. Si noti che ci sono k possibilità per effettuare il primo taglio. Una volta fatto il primo taglio, rimangono $k - 1$ punti in cui è possibile tagliare. Ricorsivamente, ciò si può esprimere come

$$C(k) = kC(k-1)$$

Per rendere il problema più interessante, si aggiunga un vincolo: bisogna sempre tagliare i pezzi più a sinistra che possono ancora essere tagliati. Ora, se k vale 3, si può tagliare la barretta secondo le sequenze 123, 132, 213, 312 o 321. La sequenza 231 non è ammessa perché dopo il taglio alla posizione 2 si può solo effettuarne uno alla posizione 1, essendo questo il punto più a sinistra. Ci sono ancora k possibilità per il primo taglio, ma ora è necessario contare i numeri di modi di tagliare due pezzi della barretta e moltiplicarli tra loro. Ricorsivamente, questo si può esprimere come:

$$D(k) = \sum_{i=1}^{k} D(i-1)D(k-1)$$

Quando k è uguale a 3, bisogna calcolare

$$D(3) = \sum_{i=1}^{3} D(i-1)D(3-i) = D(0)D(2) + D(1)D(1) + D(2)D(0)$$

$$D(2) = \sum_{i=1}^{2} D(i-1)D(2-i) = D(0)D(1) + D(1)D(0)$$

$$D(1) = \sum_{i=1}^{1} D(i-1)D(1-i) = D(0)D(0)$$

Per entrambe le formule ricorsive, se $k = 0$ esiste solo un modo di dividere la barretta.

Si sviluppi un programma che legga da tastiera un valore per k e mostri $C(k)$ e $D(k)$. La quantità $D(k)$ è interessante perché risulta essere il numero di modi in cui si possono inserire coppie di parentesi in un'espressione matematica con k operatori binari.

9. Un tempo, in un regno lontano, il re accumulava scorte di cibo e il popolo soffriva la fame. Il suo consigliere suggerì di utilizzare le riserve di cibo per aiutare il popolo, ma il re rifiutò. Un giorno, un piccolo gruppo di ribelli cercò di uccidere il re, ma fu fermato dal consigliere. Come ricompensa, il re volle fare un regalo al consigliere. Questi chiese qualche chicco di grano preso dalle riserve reali da distribuire al popolo. Il numero di chicchi sarebbe stato determinato piazzandoli su una scacchiera. Pose un chicco di grano sulla prima casella, due sulla seconda, quattro sulla terza, otto sulla quarta e così via.

   Calcolare il numero complessivo di chicchi di grano sistemati su k caselle scrivendo un metodo ricorsivo `calcolaChicchiTotali(k, chicchi)`. A ogni chiamata, il metodo "posiziona" dei chicchi su una singola casella; `chicchi` è il numero di chicchi di grano da sistemare su quella casella. Se k è uguale a 1, il metodo restituisce `chicchi`. Altrimenti, effettua una chiamata ricorsiva nella quale il valore di k è diminuito di 1 e `chicchi` è raddoppiato. La chiamata ricorsiva calcola il numero di chicchi da sistemare nelle altre k - 1 caselle. Per trovare il numero totale di chicchi su tutte le k caselle, sommare a `chicchi` il risultato della chiamata ricorsiva e restituire la somma.

10. In una stanza ci sono n persone, dove n è un intero maggiore o uguale a 2. Ogni persona stringe la mano una volta a tutte le altre. Qual è il numero totale di strette di mano? Si scriva un metodo ricorsivo per la soluzione di questo problema con la seguente intestazione:

```
public static int stretteDiMano(int n)
```

dove `stretteDiMano(n)` restituisce il numero totale di strette di mano tra n persone nella stanza. Per cominciare, se ci sono solo una o due persone nella stanza,

```
stretteDiMano(1) = 0
stretteDiMano(2) = 1
```

11. Data la seguente definizione di un array bidimensionale:

```
String[][] data = {
   {"A","B"},
   {"1","2"},
   {"XX","YY","ZZ"}
};
```

si scriva un programma ricorsivo che stampi tutte le combinazioni costruite prendendo un elemento da ogni array unidimensionale contenuto nell'array bidimensionale. Nell'esempio precedente, il risultato da ottenere (anche se non necessariamente con quest'ordine) è:

```
A 1 XX
A 1 YY
A 1 ZZ
A 2 XX
A 2 YY
A 2 ZZ
B 1 XX
B 1 YY
B 1 ZZ
B 2 XX
B 2 YY
B 2 ZZ
```

Il programma deve funzionare con array di lunghezza qualunque in ogni dimensione. Per esempio, dato l'array

```
String[][] data = {
   {"A"},
   {"1"},
   {"2"},
   {"XX","YY"}
};
```

il programma dovrà stampare

```
A 1 2 XX
A 1 2 YY
```

Capitolo 8

Definire classi e creare oggetti

OBIETTIVI

- Descrivere i concetti di classe e di oggetto di una classe.
- Creare un oggetto di una certa classe.
- Definire una nuova classe Java.
- Usare i modificatori di accesso `public` e `private`.
- Definire metodi *get* e *set* di una classe.
- Definire e utilizzare i metodi privati di una classe.
- Descrivere i concetti di information hiding e incapsulamento.
- Scrivere precondizioni e postcondizioni per un metodo.
- Descrivere lo scopo di `javadoc`.
- Disegnare semplici diagrammi delle classi UML.
- Descrivere riferimenti (*reference*), variabili e parametri di un certo tipo classe.
- Definire il metodo `equals` e altri metodi che producono risultati booleani.

Questo capitolo riprende un concetto già introdotto nei capitoli precedenti: gli oggetti. Un oggetto viene assegnato a variabili di tipo classe. Gli oggetti hanno associati dei dati e possono effettuare delle azioni. Queste azioni sono definite da metodi. I metodi sono già stati introdotti nel Capitolo 5. In questo capitolo si vedrà la differenza tra metodi di classe e metodi di istanza. Nei capitoli precedenti sono già stati utilizzati alcuni oggetti e sono stati invocati i loro metodi. Per esempio, sono stati creati e utilizzati oggetti di tipo `String`. Se `nome` è un oggetto di tipo `String`, la sua lunghezza viene restituita dal metodo `length`. Quindi, la lunghezza di `nome` corrisponde al valore restituito dall'espressione `nome.length()`. La classe `String` è già definita nella Java Class Library. Questo capitolo spiega come definire nuove classi e come usare gli oggetti e i metodi di queste nuove classi.

Prerequisiti

Prima di leggere questo capitolo, si deve aver acquisito familiarità con quanto descritto nei primi cinque capitoli. Potrebbe inoltre essere utile rileggere il paragrafo "Programmazione a oggetti" nel Capitolo 1.

8.1 Definizione di classi

Un programma Java è costituito da oggetti di vario tipo che interagiscono tra loro. Prima di entrare nel dettaglio della definizione delle classi e degli oggetti in Java, si rivedrà e rielaborerà quanto decritto nei capitoli precedenti a proposito delle classi e degli oggetti.

Gli **oggetti** di un programma possono rappresentare oggetti del mondo reale (come automobili o case) oppure astrazioni (come colori, forme o parole). Una classe è la definizione di un tipo di oggetto. È come uno stampo (*blueprint*) per la costruzione di oggetti di un certo tipo. Per esempio, la Figura 8.1 descrive una classe chiamata `Automobile`: una descrizione generale di un'automobile e delle sue caratteristiche e funzionalità.

Gli oggetti di questa classe rappresentano automobili specifiche. La figura mostra tre oggetti di tipo `Automobile`. Ciascuno di questi oggetti è un'**istanza** (o **oggetto**)

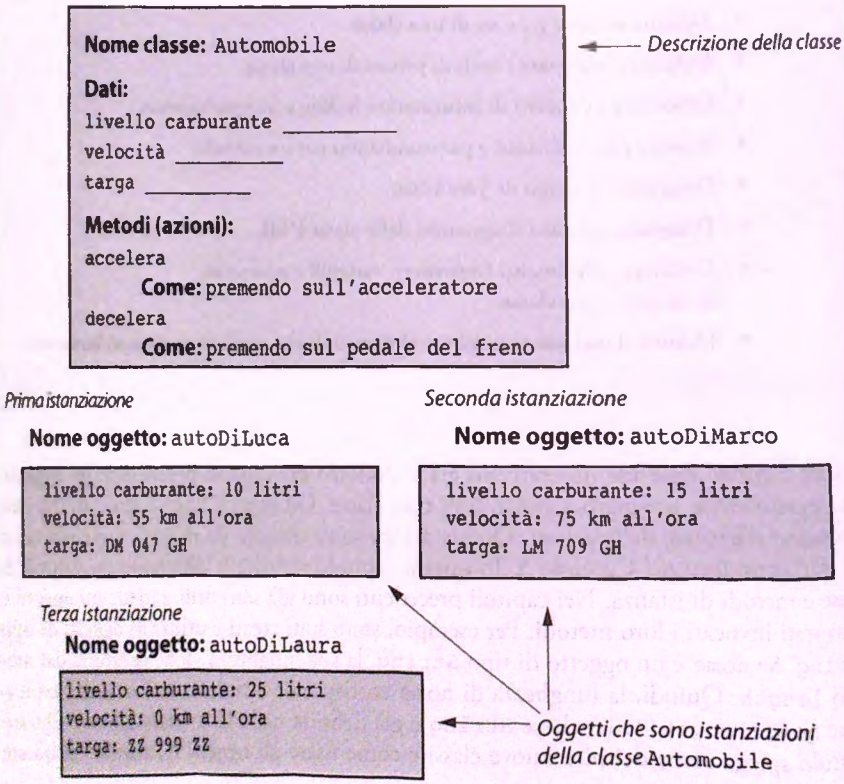


Figura 8.1 Una classe come *blueprint*.

della classe `Automobile` e quindi soddisfa la definizione data dalla classe `Automobile`. Si possono creare, o meglio **istanziare**, più oggetti della stessa classe. In questo caso, gli oggetti rappresentano le singole automobili, mentre la classe `Automobile` è una descrizione generica di un'automobile e delle sue caratteristiche. Questo esempio fornisce una visione molto semplificata di un'automobile, tuttavia permette di comprendere che cosa si intende per classe e istanza.

Una classe specifica gli attributi, o dati, degli oggetti della classe. La definizione della classe `Automobile` indica che un oggetto di tale classe ha tre attributi: un numero che indica quanti litri di benzina sono presenti nel serbatoio, un altro numero che indica la velocità dell'automobile e una stringa che indica la targa. La definizione di una classe non specifica il valore degli attributi, non include, quindi, né numeri, né stringhe. I valori degli attributi sono specifici dei singoli oggetti; la classe specifica solamente il tipo (di dato) di questi attributi.

Una classe, inoltre, specifica le azioni che possono essere svolte dagli oggetti e come queste azioni vengono svolte. Per esempio, la classe `Automobile` specifica due azioni: `accelera` e `decelera`, le due sole azioni che un oggetto `Automobile` può compiere. Queste azioni sono descritte all'interno della classe per mezzo di metodi. Tutti gli oggetti di una classe hanno gli stessi metodi. Per esempio, tutti gli oggetti della classe `Automobile` hanno gli stessi metodi, che sono `accelera` e `decelera`. Le definizioni dei metodi fanno parte della definizione della classe; essi descrivono il modo in cui gli oggetti svolgono le azioni.

La notazione presentata nella Figura 8.1 non è molto comoda, poiché è pesante da leggere e troppo ricca di dettagli superflui. Per questo motivo, progettisti e sviluppatori software usano una notazione molto più sintetica per descrivere le proprietà di una classe. Questa notazione, mostrata in Figura 8.2, è detta **diagramma delle classi UML** o più semplicemente **diagramma delle classi**. UML è un'abbreviazione per **Universal Modeling Language** (letteralmente "linguaggio di modellazione universale"). La classe descritta nella Figura 8.2 è la stessa descritta nella Figura 8.1. Tutte le notazioni usate nella Figura 8.2 saranno descritte nelle pagine di questo capitolo.

Nella Figura 8.1 si può notare un'ultima caratteristica degli oggetti `Automobile`: ciascun oggetto ha un nome. Nella Figura 8.1, i nomi usati sono `autoDiLuca`, `autoDiMarco` e `autoDiLaura`. In un programma Java, questi nomi corrisponderebbero a variabili di tipo `Automobile`. Cioè, il tipo di queste variabili è il tipo classe `Automobile`.

Prima di procedere con la definizione di una semplice classe, nei prossimi paragrafi saranno riassunti alcuni concetti inerenti il salvataggio delle classi e la compilazione, già descritti, comunque, nel Capitolo 1.

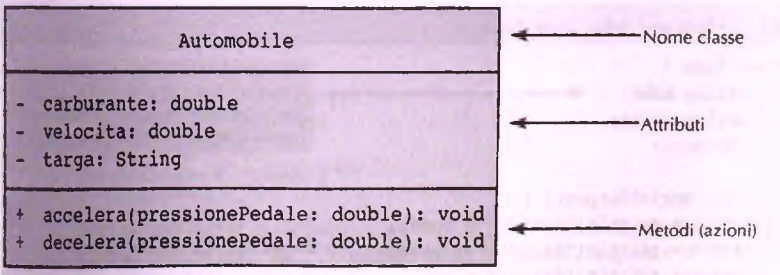


Figura 8.2 La classe rappresentata sotto forma di diagramma delle classi UML.

8.1.1 File delle classi e compilazione

Indipendentemente dal fatto che si utilizzino le classi descritte in questo testo o classi scritte in altro modo, è necessario salvare ciascuna definizione di classe in un file distinto. Ci siano alcune rare eccezioni a questa regola, ma non è necessario considerarle al momento. Un file contenente la definizione di una classe ha lo stesso nome della classe stessa e usa come estensione, cioè come ultima parte del nome, la stringa `.java`. Quindi la definizione della classe `Automobile` sarà contenuta nel file `Automobile.java`.

È possibile compilare una classe Java prima di avere un programma che la utilizzi. Il bytecode generato a partire dalla definizione di una classe e per effetto della compilazione, viene memorizzato in un file con lo stesso nome della classe, ma con l'estensione `.class`. Quindi, la compilazione del file `Automobile.java` genera il file `Automobile.class`. Una volta che il file `Automobile.class` è stato generato, non è più necessario ricompilare la classe per usare la sua definizione. Questa convenzione per i nomi si applica a interi programmi così come alle singole classi. Si noti che ogni programma che definisce un metodo `main`, ha un nome di classe all'inizio del file. Tale nome di classe deve essere usato per assegnare un nome al file contenente il programma. Per esempio, il programma del Listato 8.1 è contenuto nel file `Cane.java`. Fintantoché le classi usate da un programma sono memorizzate nella stessa directory che contiene il file del programma che le usa, non occorre preoccuparsi della posizione delle classi. Il Capitolo 9 descrive come gestire classi memorizzate in directory differenti.

8.1.2 Variabili di istanza

Il Listato 8.1 contiene la definizione di una semplice classe. Sebbene la semplicità di questa classe agevoli la descrizione di questo primo esempio, tale classe viola diversi principi di progettazione molto importanti. Man mano che si procederà nel capitolo, tali principi saranno introdotti e applicati negli esempi successivi. Il nome di questa prima classe è `Cane`.

Questa classe è stata progettata per mantenere alcune informazioni relative ai cani. Ciascun oggetto di questa classe contiene tre dati: un nome, la razza e l'età. Gli oggetti presentano due comportamenti definiti dai metodi `scriviOutput` e `getEtaInAnniUmani`. Il metodo `scriviOutput` visualizza i valori dei dati memorizzati di un cane, mentre `getEtaInAnniUmani` rapporta l'età di un cane a quella umana. Sia i dati sia i metodi vengono spesso chiamati **membri** dell'oggetto, in quanto appartengono all'oggetto. Questo testo chiamerà i dati con il termine **variabili di istanza**, mentre i metodi con il termine **metodi di istanza** per differenziarli da quelli di classe introdotti nel Capitolo 5.

MyLab

LISTATO 8.1 Definizione della classe `Cane`.

```
public class Cane {
    public String nome;
    public String razza;
    public int anni;

    public void scriviOutput() {
        System.out.println("Nome: " + nome);
        System.out.println("Razza: " + razza);
        System.out.println("Eta': " + anni);
    }
```

Più avanti nel capitolo si vedrà che il modificatore di visibilità `public` per le variabili di istanza dovrebbe essere sostituito con `private`.

```
    public int getEtaInAnniUmani() {
        int etaUmana = 0;
        if (anni <= 2) {
            etaUmana = anni * 11;
        } else {
            etaUmana = 22 + ((anni-2) * 5);
        }
        return etaUmana;
    }
}
```

Le tre righe che seguono sono estratte dalla definizione della classe `Cane` e definiscono tre variabili di istanza:

```
public String nome;
public String razza;
public int anni;
```

La parola chiave `public` indica semplicemente che non ci sono restrizioni su come e dove queste variabili di istanza possono essere usate. Presto si capirà che usare `public` in questo caso non è una buona idea, ma per ora si ignori questo aspetto. Ciascuna delle righe precedenti dichiara una variabile di istanza. Si noti che a ciascuna variabile è associato un tipo. Per esempio, la variabile `nome` è di tipo `String`.

Si può pensare a un oggetto di una classe come a un elemento complesso contenente al proprio interno una serie di variabili di istanza. In questo specifico caso, le variabili di istanza sono `nome`, `razza` e `anni`. Ciascuna istanza, cioè ciascun oggetto della classe `Cane`, possiede una propria copia di queste tre variabili ed è proprio per questo che sono chiamate variabili di istanza. Il programma nel Listato 8.2 mostra come usare la classe `Cane` e gestire queste variabili di istanza.

LISTATO 8.2 Utilizzo della classe `Cane`.

MyLab

```
public class CaneDemo {
    public static void main(String[] args) {
        Cane balto = new Cane();
        balto.nome = "Balto";
        balto.anni = 8;
        balto.razza = "Husky Siberiano";

        balto.scriviOutput();

        Cane scooby = new Cane();
        scooby.nome = "Scooby";
        scooby.anni = 9;
        scooby.razza = "Alano";

        System.out.println(scooby.nome + " e' un " + scooby.razza + ".");
        System.out.print("Ha " + scooby.anni + " anni, oppure ");
        int anniUmani = scooby.getEtaInAnniUmani();
```

```
        System.out.println(anniUmani + " in anni umani.");
    }
}
```

Esempio di output

```
Nome: Balto
Razza: Husky Siberiano
Eta': 8
Scooby e' un Alano.
Ha 9 anni, oppure 57 in anni umani.
```

La riga seguente, estratta dal Listato 8.2, crea un oggetto di tipo `Cane` e associa a questo oggetto la variabile `balto`:

```
Cane balto = new Cane();
```

Le variabili `balto` e `scooby` fanno riferimento a due ben distinti oggetti di tipo `Cane`, ognuno con le proprie variabili di istanza `nome`, `razza` e `anni`. Si può far riferimento a una di esse scrivendo il nome dell'oggetto seguito da un punto e quindi il nome della variabile di istanza. Per esempio, l'istruzione:

```
balto.nome
```

denota la variabile di istanza `nome` appartenente all'oggetto `balto`.

Dato che `nome` è di tipo `String`, `balto.nome` è una variabile di tipo `String` e può essere usata ovunque si possa usare una variabile di tipo `String`. Per esempio, tutte quelle che seguono sono istruzioni valide:

```
balto.nome = "Balto";
System.out.println("Il nome del cane e' " + balto.nome);
String nelNome = balto.nome;
```

Poiché ogni oggetto di tipo `Cane` ha le proprie tre variabili di istanza, se il programma contiene anche l'istruzione seguente

```
Cane scooby = new Cane();
```

allora `balto.nome` e `scooby.nome` sono due variabili di istanza diverse e possono quindi avere due valori diversi. Nel Listato 8.2 hanno due valori diversi poiché `balto.nome` ha valore "`Balto`" e `scooby.nome` ha valore "`Scooby`".

FAQ Perché serve la parola chiave new?

In Java, `new` è un operatore unario usato per creare oggetti di una classe. Quando l'operatore `new` viene usato in un'espressione come:

```
Cane scooby = new Cane();
```

crea un oggetto della classe `Cane` e restituisce il suo indirizzo di memoria. L'istruzione Java sopra indicata assegna questo indirizzo alla variabile `scooby`. Un oggetto può avere al proprio interno più variabili, le sue variabili di istanza. L'operatore `new` predispone queste variabili di istanza all'interno dell'oggetto nel momento in cui lo crea.

FAQ **Se il programma presentato nel Listato 8.2 è una classe, perché non ha variabili di istanza?**

Un programma è semplicemente una classe che include il metodo `main`. Tuttavia, un programma può avere anche altri metodi e variabili, sebbene in nessuno degli esempi presentati finora ci sia una variabile di istanza o un metodo al di fuori del metodo `main`.

8.1.3 Metodi di istanza

Come introdotto nel Capitolo 5, un metodo è un insieme di istruzioni con un nome la cui invocazione comporta l'esecuzione delle istruzioni in esso definite. Nel Capitolo 5 sono stati introdotti i metodi di classe (o statici) e si è visto come è possibile definirli e invocarli. Un **metodo di istanza** è un metodo che viene invocato su un oggetto e che può manipolare lo stato dell'oggetto stesso. Nel Listato 8.2, l'istruzione

```
balto.scriviOutput();
```

invoca il metodo `scriviOutput` usando l'oggetto `balto` che è di tipo `Cane`.

Si consideri questa invocazione più nel dettaglio. Un metodo di istanza definito in una classe viene invocato usando un oggetto di quella classe[1]. L'oggetto prende il nome di oggetto **chiamante** (*calling object*) o ***oggetto ricevente*** (*receiving object*). Quindi, espressioni come *l'oggetto che ha invocato (o chiamato) il metodo* e *l'oggetto che riceve l'invocazione (o chiamata) di un metodo* sono analoghe. L'invocazione viene effettuata scrivendo il nome dell'oggetto, per esempio `balto`, seguito da un punto e quindi dal nome del metodo, per esempio `scriviOutput()`, e infine da una coppia di parentesi che possono contenere gli argomenti per il metodo.

Si osservi ora la definizione del metodo `void scriviOutput` presente nel Listato 8.1 e di seguito riportato

```
public void scriviOutput() {
    System.out.println("Nome: " + nome);
    System.out.println("Razza: " + razza);
    System.out.println("Eta': " + anni);
}
```

Come è possibile notare, l'intestazione del metodo `void scriviOutput` è uguale a un qualsiasi altro metodo `void` definito nel Capitolo 5, ad eccezione del fatto che non compare il modificatore `static`. Analoga osservazione si può fare sull'intestazione del metodo che restituisce un valore `getEtaInAnniUmani()` definito anch'esso nel Listato 8.1: è uguale a un qualsiasi altro metodo introdotto nel Capitolo 5 che restituisce un valore di tipo `int` ad eccezione del modificatore `static`.

La differenza nell'intestazione dei metodi visti nel Capitolo 5 risiede quindi nell'assenza del modificatore `static`: se non presente, il metodo dichiarato è un metodo di istanza.

Si ricorda che il modificatore d'accesso `public` indica che non ci sono particolari restrizioni sull'uso del metodo. I prossimi paragrafi di questo capitolo mostreranno come la parola chiave `public` possa essere sostituita da altre parole chiave che limitano l'uso del metodo.

[1] Si vedrà nel Capitolo 10 che in realtà non è sempre vero grazie all'ereditarietà.

La differenza sostanziale tra metodi visti nel Capitolo 5 e i metodi di istanza risiede nel corpo del metodo. Si osservi il corpo del metodo `scriviOutput`. È possibile notare come esso contenga istruzioni che fanno riferimento alle variabili di istanza definite nella classe `Cane`. Per esempio, la prima istruzione presente nel metodo `scriviOutput`

```
System.out.println("Nome: " + nome);
```

fa riferimento alla variabile di istanza nome. Un metodo di istanza si differenzia da un metodo trattato nel Capitolo 5 dal fatto che può contenere istruzioni che fanno riferimento alle variabili di istanza.

Quando viene eseguita l'istruzione

```
balto.scriviOutput();
```

presente nel Listato 8.2, le variabili di istanza a cui il metodo fa riferimento sono quelle proprie dell'oggetto che ha invocato il metodo. Nel caso specifico, le variabili di istanza `nome`, `razza` e `anni` a cui il metodo `scriviOutput` si riferisce sono quelle proprie dell'oggetto `balto`. Di conseguenza i valori saranno `Balto`, `8` e `Husky Siberiano`.

Quando invece viene eseguita l'istruzione:

```
int anniUmani = scooby.getEtaInAnniUmani();
```

presente nel Listato 8.2, la variabile di istanza `anni` a cui il metodo `getEtaInAnniUmani` fa riferimento è quella propria dell'oggetto `scooby`. Di conseguenza il valore è 9.

Chi può invocare un metodo

Tutte le definizioni di metodo di istanza compaiono nella definizione della classe alla quale appartengono. Come conseguenza, questi metodi possono essere usati esclusivamente con oggetti della classe in cui il metodo è definito. Se si considera il metodo `scriviOutput`, questo è definito all'interno della classe `Cane` e può quindi essere usato solo con oggetti della classe `Cane`.

Invocare (o chiamare) un metodo

Si invoca un metodo scrivendo il nome dell'oggetto che riceve l'invocazione, seguito da un punto, dal nome del metodo e da una coppia di parentesi che possono contenere gli argomenti, i quali passano informazioni al metodo.

Esempi

```
balto.scriviOutput();
int anniUmani = scooby.getEtaInAnniUmani();
```

Definire i metodi di istanza

Ciascun metodo appartiene a una certa classe ed è accessibile a tutti gli oggetti creati da quella classe. La definizione di un metodo si trova all'interno della classe cui appartiene. Ciascun metodo restituisce un singolo valore oppure non restituisce alcun valore (metodo `void`).

Sintassi

```
public tipo_valore_restituito nome_metodo(parametri) {
    istruzioni

}
```

Esempi

```
public void scriviOutput() {
    System.out.println("Nome: " + nome);
    System.out.println("Razza: " + razza);
    System.out.println("Eta': " + anni);
}

public int getEtaInAnniUmani() {
    int etaUmana = 0;
    if (anni <= 2) {
        etaUmana = anni * 11;
    } else {
        etaUmana = 22 + ((anni-2) * 5);
    }
    return etaUmana;
}
```


Le variabili di istanza sono inizializzate a valori di deafult

Quando viene istanziato un oggetto, i valori delle sue variabili di istanza sono automaticamente inizializzate a valori di default che dipendono dal tipo con cui la variabile è stata dichiarata. Per esempio, una variabile di tipo `int` assumerà valore `0`, una di tipo `boolean` valore `false`. Una variabile di istanza di tipo `String`, che è a tutti gli effetti una classe, assumerà, invece, il valore di default per il tipo classe `null`. Il valore `null` sarà approfondito nel capitolo successivo.

Si ricorda, invece, che il valore di una variabile locale a un metodo è indefinito fino a quando non viene effettuato un esplicito assegnamento.

Se per esempio si eseguono le seguenti istruzioni

```
Cane pippo = new Cane();
System.out.println(pippo.anni);
System.out.println(pippo.nome);
```

si avrà a video il seguente output:

```
0
null
```

ESEMPIO DI PROGRAMMAZIONE
PRIMO TENTATIVO DI DESCRIZIONE DELLA CLASSE `Specie`

La classe presentata nel Listato 8.3 è stata progettata per registrare informazioni riguardanti alcune specie di animali in via d'estinzione. Questa classe è leggermente più complessa della classe `Cane` e viola anch'essa diversi principi di progettazione molto importanti. A mano a mano che si procederà nel capitolo l'esempio sarà migliorato e verranno descritti questi importanti principi. Il nome della classe è `SeciePrimaProva`.

MyLab

LISTATO 8.3 Definizione della classe `SeciePrimaProva` – Prima Prova.

```
import java.util.Scanner;

public class SeciePrimaProva {
    public String nome;
    public int popolazione;
    public double tassoCrescita;

    public void leggiInput() {
        Scanner tastiera = new Scanner(System.in);
        System.out.println("Qual e' il nome della specie?");
        nome = tastiera.nextLine();
        System.out.println("A quanto ammonta la popolazione?");
        popolazione = tastiera.nextInt();
        System.out.println("Inserisci il tasso di crescita " +
                           "(% crescita per anno):");
        tassoCrescita = tastiera.nextDouble();
    }

    public void scriviOutput() {
        System.out.println("Nome = " + nome);
        System.out.println("Popolazione = " + popolazione);
        System.out.println("Tasso crescita = " + tassoCrescita + "%");
    }

    public int prediciPopolazione(int anni) {
        int risultato = 0;
        double totalePopolazione = popolazione;
        int contatore = anni;

        while ((contatore > 0) && (totalePopolazione > 0)) {
            totalePopolazione = (totalePopolazione +
                (tassoCrescita / 100) * totalePopolazione);
            contatore--;
        }
```

Più avanti verrà presentata una implementazione migliore di questa classe.

Più avanti nel capitolo, si vedrà che il modificatore di visibilità `public` per le variabili di istanza dovrebbe essere sostituito con `private`.

```
        if (totalePopolazione > 0)
            risultato = (int) totalePopolazione;
        return risultato;
    }
}
```

Ciascun oggetto di questa classe contiene tre dati: un nome, l'entità della popolazione e il tasso di crescita. Tali dati sono memorizzati in tre variabili di istanza: nome, popolazione e `tassoDiCrescita`.

Gli oggetti presentano tre comportamenti definiti dai metodi `leggiInput`, `scriviOutput` e `prediciPopolazione`. Il metodo `leggiInput` richiede all'utente di immettere i valori per i dati della specie, `scriviOutput` visualizza i valori dei dati della specie, mentre `prediciPopolazione` restituisce la numerosità della specie tra il numero di anni passato come argomento del metodo.

Il codice nel Listato 8.4, crea un oggetto di tipo `SeciePrimaProva` e associa a questo oggetto la variabile `specieDelMese`. Sulla base della popolazione e del tasso di crescita inseriti, stampa a video la popolazione attesa fra 10 anni.

LISTATO 8.4 Usare la classe `SeciePrimaProva` e i suoi metodi.

MyLab

```
public class SeciePrimaProvaDemo {

  public static void main(String[] args) {

    SeciePrimaProva specieDelMese = new SeciePrimaProva();

    System.out.println("Inserisci i dati della specie del mese:");
    specieDelMese.leggiInput();
    specieDelMese.scriviOutput();

    int popolazioneFutura = specieDelMese.prediciPopolazione(10);
    System.out.println("Tra dieci anni la popolazione sara' di " +
                       popolazioneFutura + " individui.");

    //Cambia la specie per verificare come
    //si modificano i valori delle variabili di istanza:
    specieDelMese.nome = "Panthera tigris tigris";
    specieDelMese.popolazione = 3750;
    specieDelMese.tassoCrescita = 30;

    System.out.println("La nuova specie del mese:");
    specieDelMese.scriviOutput();
    System.out.println("Tra dieci anni la popolazione sara' di " +
                  specieDelMese.prediciPopolazione(10) + " individui.");
  }
}
```

Esempio di output

```
Inserisci i dati della specie del mese:
Qual e' il nome della specie?
Panda
A quanto ammonta la popolazione?
3000
Inserisci il tasso di crescita (% crescita per anno):
10
Nome = Panda
Popolazione = 3000
Tasso crescita = 10.0%
Tra dieci anni la popolazione sara' di 7781 individui
La nuova specie del mese:
Nome = Panthera tigris tigris
Popolazione = 3750
Tasso crescita = 30.0%
Tra dieci anni la popolazione sara' di 51696 individui.
```

Così come tutti gli oggetti di tipo `SeciePrimaProva`, anche l'oggetto `specieDelMese` ha tre variabili di istanza chiamate `nome`, `popolazione` e `tassoCrescita`. Invece di impostare i valori per le tre variabili di istanza come fatto nel metodo `main` del Listato 8.2, si invoca il metodo `leggiInput` per permettere all'utente di inserire tali valori da tastiera. I valori letti sono memorizzati nelle variabili di istanza appartenenti all'oggetto `specieDelMese` poiché è lui che ha invocato il metodo.

Le istruzioni del metodo `prediciPopolazione` fanno riferimento alla variabile di istanza `popolazione`. Tale variabile di istanza si riferisce a quella dell'oggetto che ha ricevuto l'invocazione del metodo, che in questo caso è `specieDelMese`.

8.1.4 La parola chiave `this`

Si osservi la definizione della classe `SeciePrimaProva` nel Listato 8.3 e poi l'uso della classe nel Listato 8.4. Si noti che le variabili di istanza sono scritte in maniera diversa, a seconda che siano all'interno della definizione della classe o all'esterno, come in un programma che usa la classe. Al di fuori della definizione della classe, il nome delle variabili di istanza è composto dal nome dell'oggetto della classe, seguito da un punto e dal nome della variabile di istanza, come nell'esempio che segue estratto dal Listato 8.4:

```
specieDelMese.nome = "Panthera Tigris Tigris"
```

Al contrario, all'interno della definizione di un metodo che risiede nella stessa classe della variabile di istanza, si utilizza semplicemente il nome della variabile di istanza senza alcun nome di oggetto o punto. Per esempio, nella definizione del metodo `leggiInput` della classe `SeciePrimaProva` nel Listato 8.3 è presente la seguente riga:

```
nome = tastiera.nextLine();
```

Ogni variabile di istanza, come `nome`, appartiene a un oggetto. Nei casi come questo, si sottintende l'oggetto omettendone il nome. Il nome sottinteso di questo oggetto è `this` (letteralmente "questo"). Sebbene questo nome sia spesso omesso, lo si può includere se si vuole. Per esempio, l'assegnamento precedente è equivalente a quello che segue:

```
this.nome = tastiera.nextLine();
```

Il seguente metodo è equivalente, invece, alla versione di `scriviOutput` usata nel Listato 8.3:

```
public void scriviOutput() {
    System.out.println("Nome = " + this.nome);
    System.out.println("Popolazione = " + this.popolazione);
    System.out.println("Tasso crescita = " + this.tassoCrescita + "%");
}
```

La parola chiave `this` rappresenta l'oggetto che riceve l'invocazione del metodo. Per esempio, nell'invocazione

```
specieDelMese.scriviOutput();
```

l'oggetto ricevente è `specieDelMese`. Di conseguenza, l'invocazione del metodo `scriviOutput` è equivalente a

```
System.out.println("Nome = " + specieDelMese.nome);
System.out.println("Popolazione = " + specieDelMese.popolazione);
System.out.println("Tasso crescita = "+ specieDelMese.tassoCrescita + "%");
```

ottenuto sostituendo `specieDelMese` a `this`. La parola chiave `this` è come uno spazio vuoto che aspetta di essere riempito con il nome dell'oggetto che riceve l'invocazione del metodo. Java permette di omettere `this` dato che sarebbe usato molto di frequente. In alcuni casi, invece, è proprio necessario usarlo.

La parola chiave `this`

Si può usare la parola chiave `this` all'interno della definizione di un metodo come un nome per l'oggetto che riceve l'invocazione del metodo.

Definizione di classe

Sintassi

```
public class nome_classe {
    dichiarazioni_variabili_di_istanza
    ...
    dichiarazioni_di_metodi
}
```

Sebbene questa sia la forma più usata, è possibile anche mischiare definizioni di metodi e dichiarazioni di variabili di istanza.

8.2 Information hiding e incapsulamento

Information hiding letteralmente vuol dire nascondere le informazioni. Sebbene questa frase sembri avere una connotazione negativa, rappresenta invece un grosso vantaggio. In un programma, la possibilità di nascondere alcune informazioni viene vista come una qualità del linguaggio che semplifica il lavoro dei programmatori e rende più comprensibile il codice sviluppato.

mente, tutte le variabili di istanza dovrebbero essere private. Una variabile di istanza viene resa privata usando la parola chiave `private` al posto della parola chiave `public`. Le parole chiave `public` e `private` sono esempi di **modificatori d'accesso** o **modificatori di visibilità**.

Si supponga di cambiare da `public` a `private` il modificatore d'accesso che precede la variabile di istanza `nome` nella definizione della classe `SeziePrimaProva` nel Listato 8.3. La definizione della classe comincerebbe così:

```
public class SeziePrimaProva {
    private String nome;    //Privata!
    public int popolazione;
    public double tassoCrescita;
```

Questa nuova definizione renderebbe non più valida la seguente istruzione nel Listato 8.4:

```
specieDelMese.nome = "Panthera Tigris Tigris";    //Non valida se privata.
```

Le seguenti istruzioni resterebbero invece valide, in quanto i modificatori d'accesso delle variabili `popolazione` e `tassoCrescita` sono rimasti `public`.

```
specieDelMese.popolazione = 3750;
specieDelMese.tassoCrescita = 30;
```

Quando una variabile di istanza è privata, il suo *nome* non è accessibile *al di fuori* della definizione della sua classe. La variabile può però essere usata all'interno di uno qualsiasi dei metodi definiti nella sua stessa classe. In particolare, si potrebbe modificare direttamente il valore della variabile di istanza. Al di fuori della definizione della classe, tuttavia, non si può far alcun riferimento diretto a questa variabile.

Si rendano, per esempio, private le variabili di istanza della classe `SeziePrimaProva`, ma lasciando invariate le definizioni dei metodi. Il risultato è mostrato nel Listato 8.5, nella classe `SezieSecondaProva`. Dato che tutte le variabili sono private, le ultime tre istruzioni che seguono sono considerate non valide all'interno di una classe diversa da `SezieSecondaProva`:

```
SezieSecondaProva specieSegreta = new SezieSecondaProva();  //Valida
specieSegreta.leggiInput();                                 //Valida
specieSegreta.nome = "Aardvark";                            //Non valida
System.out.println(specieSegreta.popolazione);              //Non valida
System.out.println(specieSegreta.tassoCrescita);            //Non valida
```

Si noti, tuttavia, che l'invocazione del metodo `leggiInput` è valida. Questo vuol dire che esiste ancora un modo per assegnare un valore alle variabili di istanza di un oggetto, anche se le sue variabili di istanza sono private. All'interno della definizione del metodo `leggiInput` (mostrato nel Listato 8.3) sono definite istruzioni di assegnamento come:

```
nome = tastiera.nextLine();
```

e

```
popolazione = tastiera.nextInt();
```

che assegnano un valore alle variabili di istanza. Di conseguenza, rendere privata una variabile di istanza, non significa che non se ne può modificare il valore. Significa, invece, che non è possibile utilizzare il *nome* della variabile di istanza per far riferimento direttamente alla variabile al di fuori della classe.

Anche i metodi possono essere privati. Se a un metodo è assegnato il modificatore d'accesso `private`, il metodo non può essere invocato al di fuori della definizione della classe. Tuttavia, può essere invocato all'interno di un altro metodo appartenente alla stessa classe. La maggior parte dei metodi è pubblica, ma se un metodo deve essere utilizzato solo dagli altri metodi della sua classe, dovrebbe essere reso privato. Usare metodi privati è un altro modo per nascondere i dettagli implementativi di una classe.

Anche le stesse classi possono essere private.

LISTATO 8.5 Una classe con variabili di istanza `private`.

MyLab

```
import java.util.Scanner;

public class SpecieSecondaProva {

    private String nome;
    private int popolazione;
    private double tassoCrescita;

        <Le definizioni dei metodi leggiInput, scriviOutput
         e prediciPolazione sono le stesse del Listato 8.3.>
}
```

Più avanti verrà presentata una versione ulteriormente migliorata di questa classe.

I modificatori d'accesso `public` e `private`

All'interno della definizione di una classe, ogni dichiarazione di variabile di istanza e ogni definizione di metodo, così come la definizione della classe stessa, possono essere preceduti dai modificatori d'accesso `public` o `private`. I modificatori d'accesso specificano in quali classi è possibile utilizzate una classe, una variabile di istanza o un metodo. Se una variabile di istanza è privata, il suo nome non può essere usato per accedere al suo valore al di fuori della definizione della classe. Tuttavia, il nome può essere usato dai metodi definiti nella classe di appartenenza. Se una variabile di istanza è pubblica, il suo nome può essere utilizzato ovunque, senza alcuna limitazione.

Se la definizione di un metodo è privata, il metodo non può essere invocato al di fuori della definizione della classe. Tuttavia, può essere invocato dai metodi della sua stessa classe. Se il metodo è pubblico, può essere invocato ovunque, senza alcuna limitazione.

Normalmente tutte le variabili di istanza sono private e la maggior parte dei metodi sono pubblici.

Le variabili di istanza dovrebbero essere `private`

Tutte le variabili di istanza di una classe dovrebbero essere dichiarate come `private`. In questo modo si costringe un programmatore che debba usare la classe ad accedere alla variabile di istanza solo attraverso i metodi della classe. Questo permette alla classe di controllare tutte le attività di lettura e scrittura dei valori delle variabili di istanza. L'esempio che segue mostra perché è importante rendere private le variabili di istanza.

ESEMPIO DI PROGRAMMAZIONE
PERCHÉ LE VARIABILI DI ISTANZA DOVREBBERO ESSERE DICHIARATE `private`

Il Listato 8.6 mostra una semplice classe che rappresenta un rettangolo. Questa classe ha tre variabili di istanza private che ne rappresentano la larghezza, l'altezza e l'area. Il metodo `setDimensioni` assegna la larghezza e l'altezza, mentre il metodo `getArea` restituisce l'area del rettangolo.

La classe `Rettangolo` potrebbe essere usata nel seguente modo:

```
Rettangolo box = new Rettangolo();
box.setDimensioni(10, 5);
System.out.println("L'area del rettangolo e' " + box.getArea());
```

L'output di queste semplici istruzioni è:

```
L'area del rettangolo e' 50
```

Se le tre variabili di istanza della classe `Rettangolo` fossero state pubbliche invece di private, dopo aver creato un rettangolo 10 × 5 si sarebbe stati in grado di modificare il valore di una qualsiasi variabile di istanza. Perciò, mentre `area` ha valore 50, si sarebbe potuto modificare il valore della variabile `larghezza` assegnandole valore 6 con l'istruzione che segue:

```
box.larghezza = 6;    //Questo può essere fatto se larghezza è public
```

Se quindi si fosse invocato `getArea`, per il modo in cui è stato implementato tale metodo, si sarebbe ottenuta un'area di 50 invece della nuova area, 30. Rendendo pubbliche le variabili di istanza della classe `Rettangolo`, si lascia aperta la possibilità che l'area del rettangolo possa non essere uguale a larghezza * altezza. Rendendo invece le variabili di istanza private, si restringono i modi con cui queste possono essere utilizzate.

MyLab

LISTATO 8.6 La classe `Rettangolo`.

```
/**
Classe che rappresenta un generico rettangolo
*/
public class Rettangolo {

    private int larghezza;
    private int altezza;
    private int area;

    public void setDimensioni(int nuovaLarghezza, int nuovaAltezza) {
        larghezza = nuovaLarghezza;
        altezza = nuovaAltezza;
        area = larghezza * altezza;
    }
```

```
    public int getArea() {
        return area;
    }
}
```

Le variabili di istanza pubbliche possono causare il danneggiamento dei dati della classe

MyLab

Video 8.2 Accesso pubblico e privato

L'esempio di programmazione precedente ha mostrato come la possibilità di cambiare i valori delle variabili di istanza di una classe possa portare a dati incoerenti all'interno di un oggetto. Dato che le variabili di istanza pubbliche rendono possibile questa situazione, è sempre bene renderle private.

ESEMPIO DI PROGRAMMAZIONE
UN ALTRO ESEMPIO DI IMPLEMENTAZIONE DELLA CLASSE RETTANGOLO

Si consideri la classe `Rettangolo2` presentata nel Listato 8.7. Questa classe ha gli stessi metodi della classe `Rettangolo`, ma li implementa in una maniera leggermente differente. La nuova classe calcola l'area del rettangolo solo quando è invocato il metodo `getArea`. In più, l'area non viene salvata in una variabile di istanza.

LISTATO 8.7 Un'altra classe `Rettangolo`.

MyLab

```
/**
Un'altra classe che rappresenta un rettangolo generico
*/
public class Rettangolo2 {

    private int larghezza;
    private int altezza;

    public void setDimensioni(int nuovaLarghezza, int nuovaAltezza) {
        larghezza = nuovaLarghezza;
        altezza = nuovaAltezza;
    }

    public int getArea() {
        return larghezza * altezza;
    }
}
```

Si noti che la classe `Rettangolo2` si può utilizzare nello stesso modo con cui è stata utilizzata la classe `Rettangolo`. Si possono, quindi, riutilizzare le istruzioni precedentemente usate per `Rettangolo`, sostituendo solo `Rettangolo` con `Rettangolo2`. Si ottiene, quindi:

```
Rettangolo2 box = new Rettangolo2();
box.setDimensioni(10, 5);
System.out.println("L'area del rettangolo e' " + box.getArea());
```

Il motivo per cui si sono potute utilizzare le stesse istruzioni è che i metodi hanno lo stesso comportamento in entrambe le classi, anche se sono implementati in maniera differente.

L'implementazione non dovrebbe influenzare il comportamento

Due classi possono presentare lo stesso comportamento, ma avere implementazioni diverse.

Le classi `Rettangolo` e `Rettangolo2` si comportano allo stesso modo. Più in dettaglio, le due classi fanno la stessa *cosa*, cambia il *modo* in cui lo fanno. Nella classe `Rettangolo2`, il metodo `getArea` calcola e quindi restituisce l'area senza salvarla. La classe `Rettangolo`, invece, ha una variabile di istanza, `area`, che contiene l'area del rettangolo. Il metodo `setDimensioni` della classe `Rettangolo` calcola l'area e la memorizza nella variabile di istanza `area`. Il metodo `getArea` restituisce semplicemente il valore contenuto nella variabile di istanza `area`.

È difficile stabilire quale tra queste due classi sia migliore rispetto all'altra. La risposta dipende da ciò che si intende per "migliore". Tuttavia si possono fare alcune osservazioni.

- La classe `Rettangolo` usa più memoria della classe `Rettangolo2`, in quanto ha una variabile di istanza in più.
- La classe `Rettangolo` calcola sempre l'area anche quando questo non è necessario.

La seconda osservazione suggerisce che l'uso della classe `Rettangolo` potrebbe richiedere più tempo di computazione rispetto a `Rettangolo2`. Per capire meglio questo caso bisognerebbe supporre che entrambe le classi abbiano molti più metodi e, di conseguenza, rendano molto probabile il fatto che il metodo `getArea` venga invocato raramente. Se l'area del rettangolo viene richiesta raramente, allora è meglio usare la classe `Rettangolo2` in quanto permette di risparmiare tempo di esecuzione e memoria. Al contrario, se si invoca ripetutamente il metodo `getArea` per uno stesso rettangolo, è meglio usare la classe `Rettangolo`, in quanto farebbe risparmiare tempo di computazione dal momento che il calcolo dell'area viene effettuato una sola volta.

L'implementazione può influenzare l'efficienza

Il modo in cui è implementata una classe può influenzare il suo tempo di esecuzione e la sua occupazione di memoria.

8.2.4 Metodi get e set

Rendere private tutte le variabili di istanza di una classe permette di avere un controllo totale su di esse. A volte, però, sussistono delle ragioni più che legittime per accedere a tali variabili di istanza. In questo caso è necessario fornire dei metodi di accesso. Un **metodo d'accesso**, detto anche **metodo get** (letteralmente "prendi") o **getter**, è semplicemente un metodo che permette di osservare quali sono i dati contenuti in una variabile di istanza. Il Listato 8.8 contiene una nuova versione della classe per le specie animali riscritta in modo che abbia dei metodi d'accesso per ottenere il valore di ciascuna variabile di istanza. Il nome dei metodi d'accesso, per convenzione, inizia con la parola `get`, come in `getNome`.

I metodi d'accesso permettono di osservare i dati contenuti in una variabile di istanza. Altri metodi, conosciuti come **metodi di modifica** o **metodi set** (letteralmente "assegna") o **setter**, permettono di modificare i dati memorizzati nelle variabili di istanza private. Il nome dei metodi di modifica, per convenzione, inizia con la parola `set`. La definizione della classe ha un metodo di modifica, chiamato `setSpecie`, che permette di assegnare nuovi valori alle variabili di istanza. Il programma presentato nel Listato 8.9 illustra l'uso del metodo di modifica `setSpecie`. Questo programma è simile a quello mostrato nel Listato 8.4; ma, dato che la nuova versione della classe `Specie` possiede variabili di istanza private, in esso occorre utilizzare il metodo di modifica `setSpecie` per reimpostare il valore delle variabili di istanza.

D'ora in avanti si utilizzeranno i termini metodi *set* e metodi *get* per far riferimento rispettivamente ai metodi di modifica e ai metodi d'accesso.

Definire metodi *set* e *get* sembra annullare lo scopo per cui le variabili di istanza sono dichiarate private. In realtà non è così. Un metodo *set*, infatti, può verificare se un cambiamento è appropriato e notificare l'utente in caso di problemi. Per esempio, il metodo *set* `setSpecie` può controllare se il programma tenta inavvertitamente di assegnare un valore negativo alla variabile `popolazione`.

Il nome di questi metodi non deve necessariamente contenere i termini `get` e `set`. Per esempio, si potrebbero assegnare nomi come `restituisciValore`, `resetta` o `daiNuovoValore`. Tuttavia, per convenzione, si usano rispettivamente il termine `get` come prefisso dei metodi d'accesso e il termine `set` come prefisso dei metodi di modifica.

LISTATO 8.8 Una classe con metodi *get* e *set*.

MyLab

```
import java.util.Scanner;

public class SpecieTerzaProva {

    private String nome;
    private int popolazione;
    private double tassoCrescita;

    <Le definizioni dei metodi leggiInput, scriviOutput e
    prediciPopolazione vanno qui. Sono le stesse del Listato 8.3.>
```

Più avanti verrà presentata una versione ulteriormente migliorata di questa classe.

```
    public void setSpecie(String nuovoNome, int nuovaPopolazione,
                          double nuovoTassoCrescita) {
        nome = nuovoNome;
        if (nuovaPopolazione >= 0)
            popolazione = nuovaPopolazione;
        else {
            System.out.println("ERRORE: si sta usando un numero negativo " +
                               "per la popolazione.");
            System.exit(0);
        }
        tassoCrescita = nuovoTassoCrescita;
    }

    public String getNome() {
        return nome;
    }

    public int getPopolazione() {
        return popolazione;
    }

    public double getTassoCrescita() {
        return tassoCrescita;
    }
}
```

Un metodo *set* può controllare che alle variabili di istanza vengano assegnati valori corretti.

LISTATO 8.9 **Usare un metodo *set*.**

```
import java.util.Scanner;

public class SpecieTerzaProvaDemo {

    public static void main(String[] args) {

        SpecieTerzaProva specieDelMese = new SpecieTerzaProva();

        System.out.println("Inserisci i dati sulla specie del mese:");
        specieDelMese.leggiInput();
        specieDelMese.scriviOutput();

        System.out.println("Inserisci il numero di anni da predire:");
        Scanner tastiera = new Scanner(System.in);
        int numeroAnni = tastiera.nextInt();

        int popolazioneFutura = specieDelMese.prediciPopolazione(numeroAnni);
        System.out.println("Tra " + numeroAnni + " anni la popolazione sara' di " +
                           popolazioneFutura + " individui.");
```

```
        //Cambia la specie per verificare come
        //si modificano i valori delle variabili di istanza:
        specieDelMese.setSpecie("Panthera tigris tigris", 3750, 30);
        System.out.println("La nuova specie del mese:");
        specieDelMese.scriviOutput();
        popolazioneFutura = specieDelMese.prediciPopolazione(numeroAnni);
        System.out.println("Tra " + numeroAnni + " anni la popolazione sara' di " +
                           popolazioneFutura + " individui.");
    }
}
```

Esempio di output

```
Inserisci i dati sulla specie del mese:
Qual e' il nome della specie?
Panda
A quanto ammonta la popolazione?
3000
Inserisci il tasso di crescita (% crescita per anno):
10
Nome = Panda
Popolazione = 3000
Tasso crescita = 10.0%
Inserisci il numero di anni da predire:
10
Tra 10 anni la popolazione sara' di 7781 individui.
La nuova specie del mese:
Nome = Panthera tigris tigris
Popolazione = 3750
Tasso crescita = 30.0%
Tra 10 anni la popolazione sara' di 51696 individui.
```

Metodi *set* e *get*

Un metodo pubblico che restituisce dati memorizzati in una variabile di istanza privata è detto metodo d'accesso o metodo *get* o *getter*. Il nome dei metodi d'accesso tipicamente inizia con `get`. Un metodo pubblico che modifica i dati memorizzati in una o più variabili di istanza private è detto metodo di modifica o metodo *set* o *setter*. Il nome dei metodi di modifica inizia per convenzione con `set`.

ESEMPIO DI PROGRAMMAZIONE
LA CLASSE `Acquisto`

Il Listato 8.10 contiene una classe che rappresenta un singolo acquisto di molteplici articoli tutti dello stesso tipo, per esempio 12 mele o 2 litri di latte. Questa classe è stata

progettata per essere usata alla cassa di un supermercato. In questo supermercato gli articoli vengono venduti a gruppi, quindi il prezzo si riferisce a un gruppo di articoli. Per esempio, 5 mele a 1,25 Euro oppure 3 scatole di biscotti a 2,50 Euro.

Le righe seguenti mostrano le variabili di istanza:

```
private String nome;
private int dimGruppo;          //Parte del prezzo, come il 5
                                //in 5 per E 1.25.
private double prezzoGruppo;    //Parte del prezzo, come il 1.25
                                //in 5 per E 1.25.
private int articoliAcquistati; //Numero di articoli acquistati.
```

Di seguito viene presentato un esempio per spiegare meglio il significato di queste variabili di istanza. Se si acquistano 12 mele al prezzo di 5 per 1,25 Euro, la variabile `nome` ha il valore `"mele"`, `dimGruppo` ha il valore 5, `prezzoGruppo` il valore 1.25 e `articoliAcquistati` il valore 12. Si noti che il prezzo di 5 per 1,25 Euro è memorizzato in due diverse variabili di istanza: `dimGruppo` per il 5 e `prezzoGruppo` per 1,25 Euro.

Si consideri, per esempio, il metodo `getCostoTotale`. Il costo totale di un acquisto è calcolato come:

```
(prezzoGruppo / dimGruppo) * articoliAcquistati
```

Se l'acquisto è di 12 mele al prezzo di 5 per 1,25 Euro il costo totale è di (1,25 / 5) * 12

Si notino, inoltre, i metodi `leggiInput`, `setPrezzo` e `setArticoliAcquistati`. Tutti questi metodi controllano se vengono passati numeri negativi palesemente errati, per esempio quando l'utente inserisce il numero di articoli acquistati. Una semplice dimostrazione di un programma che usa questa classe è data nel Listato 8.11.

yLab

LISTATO 8.10 La classe `Acquisto`.

```
import java.util.Scanner;

/**
Classe per l'acquisto di molteplici quantità di
un solo tipo di articolo, come 3 arance.
I prezzi sono memorizzati nello stile delle offerte
di un supermercato, come 5 per 1.25 Euro.
*/
public class Acquisto {

    private String nome;
    private int dimGruppo;          //Parte del prezzo, come il 5
                                    //in 5 per E 1.25.
    private double prezzoGruppo;    //Parte del prezzo, come il 1.25
                                    //in 5 per E 1.25.
    private int articoliAcquistati; //Numero di articoli acquistati.

    public void setNome(String nuovoNome) {
        nome = nuovoNome;
    }
```

```
/**
Imposta il costo come numeroArticoli per prezzo.
Per esempio, 2 (numeroArticoli) per 1.99 (prezzo)
*/
public void setPrezzo(int numeroArticoli, double prezzo) {
   if ((numeroArticoli <= 0) || (prezzo <= 0)) {
      System.out.println("Errore: parametro errato in setPrezzo.");
      System.exit(0);
   } else {
      dimGruppo = numeroArticoli;
      prezzoGruppo = prezzo;
   }
}

public void setArticoliAcquistati(int numero) {
   if (numero <= 0) {
      System.out.println("Errore: parametro non corretto in " +
                         "setArticoliAcquistati.");
      System.exit(0);
   } else
      articoliAcquistati = numero;
}

/**
Legge dalla tastiera il prezzo e il numero di acquisti
*/
public void leggiInput() {
   Scanner tastiera = new Scanner(System.in);
   System.out.println("Inserisci il nome dell'articolo che intendi " +
                      "acquistare:");
   nome = tastiera.nextLine();
   System.out.println("Inserisci il prezzo dell'articolo usando " +
                      "due cifre.");
   System.out.println("Per esempio 3 per 2.99 Euro viene scritto come");
   System.out.println("3 2.99");
   System.out.println("Inserisci il prezzo dell'articolo usando " +
                      "due cifre:");
   dimGruppo = tastiera.nextInt();
   prezzoGruppo = tastiera.nextDouble();

   while ((dimGruppo <= 0) || (prezzoGruppo <= 0)) { //Prova ancora
      System.out.println(
         "Entrambi i numeri devono essere positivi, riprova.");
      System.out.println("Inserisci il prezzo dell'articolo usando " +
                         "due cifre.");
      System.out.println("Per esempio 3 per 2.99 Euro viene scritto come");
      System.out.println("3 2.99");
      System.out.println("Inserisci il prezzo dell'articolo usando " +
                         "due cifre:");
      dimGruppo = tastiera.nextInt();
      prezzoGruppo = tastiera.nextDouble();
   }
```

```
        System.out.println("Inserisci il numero di articoli acquistati:");
        articoliAcquistati = tastiera.nextInt();
        while (articoliAcquistati <= 0) { //Prova ancora:
            System.out.println("Il numero deve essere positivo, prova ancora.");
            System.out.println("Inserisci il numero di articoli acquistati: ");
            articoliAcquistati = tastiera.nextInt();
        }
    }

    /**
    Mostra il prezzo e il numero di articoli acquistati
    */
    public void scriviOutput () {
        System.out.println(articoliAcquistati + " " + nome);
        System.out.println("al prezzo di " + dimGruppo +
                           " per " + prezzoGruppo + " Euro.");
    }

    public String getNome() {
        return nome;
    }

    public double getCostoTotale() {
        return (prezzoGruppo / dimGruppo) * articoliAcquistati;
    }

    public double getCostoUnitario() {
        return prezzoGruppo / dimGruppo;
    }

    public int getArticoliAcquistati(){
        return articoliAcquistati;
    }
}
```

LISTATO 8.11 Usare la classe `Acquisto`.

```
public class AcquistoDemo {

    public static void main(String[] args) {
        Acquisto unAcquisto = new Acquisto();
        unAcquisto.leggiInput();
        unAcquisto.scriviOutput();
        System.out.println("Costo unitario: " +
            unAcquisto.getCostoUnitario() + " Euro.");
        System.out.println("Costo totale: " +
            unAcquisto.getCostoTotale() + " Euro.");
    }
}
```

Esempio di output

```
Inserisci il nome dell'articolo che intendi acquistare:
Biscotti
Inserisci il prezzo dell'articolo usando due cifre.
Per esempio 3 per 2.99 Euro viene scritto come
3 2.99
Inserisci il prezzo dell'articolo usando due cifre:
0 2.80
Inserisci il numero di articoli acquistati:
10
10 Biscotti
al prezzo di 8 per 2.8 Euro.
Costo unitario: 0.35 Euro.
Costo totale: 3.5 Euro.
```

8.2.5 La parola chiave `this` applicata alle variabili di istanza

Quando si è introdotta la visibilità delle variabili, si è affermato che all'interno di un blocco non possono coesistere due variabili che hanno lo stesso nome. Di conseguenza, anche all'interno di un metodo non possono esserci due variabili con lo stesso nome. Esiste però un'eccezione a questa regola. Se una delle due variabili è una variabile di istanza, questo è possibile.

Si consideri la seguente semplice classe che rappresenta una persona. Una persona è caratterizzata esclusivamente da un'età.

```
public class Persona {
    private int eta;

    public void setEta(int nuovaEta){
        eta = nuovaEta;
    }

    public int getEta(){
        return eta;
    }
}
```

L'intestazione del metodo `setEta` dichiara il parametro di tipo `int` con identificativo `nuovaEta`. Tale intestazione si sarebbe anche potuta scrivere come segue:

```
public void setEta(int eta)
```

cioè, utilizzare l'identificativo `eta` al posto di `nuovaEta` e modificare di conseguenza il corpo del metodo come segue:

```
public void setEta(int eta) {
    eta = eta;
}
```

Chiaramente l'intenzione è sempre quella di assegnare a `eta`, variabile di istanza, il valore di `eta` parametro del metodo.

A prima vista può sembrare illegale, poiché all'interno del corpo del metodo si hanno due variabili con lo stesso identificativo: la variabile di istanza e la variabile locale al metodo. In realtà il compilatore non produce nessun messaggio di errore. Infatti, all'interno di un blocco è possibile avere due variabili con lo stesso identificativo, a patto che una sia una variabile di istanza e l'altra una variabile locale al metodo, anche un parametro.

Se però si tenta di eseguire il seguente codice:

```
Persona p = new Persona();
p.setEta(10);
System.out.println(p.getEta());
```

l'output prodotto sarà 0. In altre parole, non viene assegnato il valore 10 alla variabile di istanza `eta` e il suo valore rimane 0, il valore assegnato per default ai tipi primitivi di tipo `int`.

Tale comportamento è dovuto al fatto che nell'istruzione di assegnamento:

```
eta = eta
```

entrambe le variabili vengono considerate come la variabile locale al metodo. Praticamente il codice non fa altro che riassegnare alla variabile locale il suo valore. Poiché l'intento era quello di assegnare alla variabile di istanza dell'oggetto invocante il metodo (`setEta`) il valore passato come argomento al metodo, occorre esplicitare che la prima variabile è in realtà la variabile di istanza e non la variabile locale.

Per riferirsi a una variabile di istanza occorre utilizzare la seguente sintassi:

`this.`*nome_della_variabile_di_istanza*

Tale sintassi è la stessa introdotta nel paragrafo "La parola chiave this" e il significato rimane quindi invariato: la parola chiave `this` rappresenta l'oggetto che riceve l'invocazione del metodo. Di conseguenza, per indicare che la prima variabile dell'assegnamento è la variabile di istanza si modifica il codice del metodo come segue:

```
public void setEta(int eta) {
    this.eta = eta;
}
```

Con questa modifica, a fronte dell'esecuzione delle istruzioni:

```
Persona p = new Persona();
p.setEta(10);
System.out.println(p.getEta());
```

l'output sarà 10, che è esattamente quello che si voleva ottenere.

8.2.6 Metodi che invocano altri metodi

Il corpo di un metodo può contenere l'invocazione di un altro metodo. Questa situazione si è già presentata più volte quando in un metodo `main` si sono invocati metodi attraverso degli oggetti. Per esempio, nel Listato 8.11 il metodo `main` ha la seguente invocazione di metodo: `unAcquisto.leggiInput`.

Tuttavia, se il metodo invocato si trova nella stessa classe, l'invocazione viene effettuata senza scrivere il nome dell'oggetto. Questa regola vale nel caso di metodi sia pubblici sia privati. Il Listato 8.12 contiene la definizione di una classe chiamata `Oracolo`.

Il metodo `parla` di questa classe sostiene un dialogo con l'utente, rispondendo a una serie di domande. Si noti che la definizione del metodo `parla` contiene un'invocazione al metodo `rispondi`, che è un metodo della classe `Oracolo`. Se si osserva la definizione del metodo `rispondi`, si può notare che contiene le invocazioni ad altri due metodi, `cercaSuggerimento` e `aggiorna`, entrambi definiti nella classe `Oracolo`.

Si consideri l'invocazione del metodo `rispondi` all'interno del metodo `parla`. Si noti che il metodo `rispondi` non è preceduto dal nome di un oggetto e dal punto. In questo caso l'oggetto ricevente la chiamata `rispondi` è esattamente lo stesso oggetto che ha ricevuto la chiamata al metodo `parla`. Il programma presentato nel Listato 8.13 crea un oggetto della classe `Oracolo` e lo assegna alla variabile `delphi`, quindi usa questo oggetto per invocare il metodo `parla` come segue:

```
delphi.parla();
```

Quindi, quando il metodo `parla` invoca il metodo `rispondi`, l'invocazione

```
rispondi();
```

nella definizione del metodo `parla` viene utilizzata da Java come se fosse

```
delphi.rispondi();
```

LISTATO 8.12 Metodi che invocano altri metodi.

MyLab

```
import java.util.Scanner;

public class Oracolo {

  private String vecchiaRisposta = "La risposta e' nel tuo cuore.";
  private String nuovaRisposta;
  private String domanda;

  public void parla() {
     System.out.print("Sono l'oracolo. ");
     System.out.println("Rispondero' a qualsiasi domanda che " +
                        "digiterai su una riga.");
     Scanner tastiera = new Scanner(System.in);
     String risposta;
     do {
        rispondi();
        System.out.println("Vuoi pormi un'altra domanda?");
        risposta = tastiera.next();
     } while (risposta.equalsIgnoreCase("si"));
     System.out.println("L'oracolo ora riposa.");
  }

  private void rispondi() {
     System.out.println("Qual e' la tua domanda?");
     Scanner tastiera = new Scanner(System.in);
     domanda = tastiera.nextLine();
     cercaSuggerimento();
```

```
        System.out.println("Hai posto la domanda:");
        System.out.println(" " + domanda);
        System.out.println("Ora ecco la mia risposta:");
        System.out.println(" " + vecchiaRisposta);
        aggiorna();
    }

    private void cercaSuggerimento() {
        System.out.println("Hmm, ho bisogno di aiuto su questo.");
        System.out.println("Scrivimi una riga di aiuto.");
        Scanner tastiera = new Scanner(System.in);
        nuovaRisposta = tastiera.nextLine();
        System.out.println("Grazie. Mi ha aiutato molto");
    }

    private void aggiorna() {
        vecchiaRisposta = nuovaRisposta;
    }
}
```

Quando si scrive la definizione di un metodo come `parla` non si conosce il nome dell'oggetto che riceverà la chiamata al metodo. Potrebbe essere diverso ogni volta. Dato che non si può conoscere in anticipo quale oggetto riceverà la chiamata al metodo, questo nome viene omesso. Perciò, nella definizione della classe `Oracolo` nel Listato 8.12 quando si scrive

```
rispondi();
```

all'interno della definizione del metodo `parla`, questa istruzione vuol dire:

```
oggetto_ricevente.rispondi();
```

Dato che la parola chiave `this` rappresenta proprio l'oggetto che riceve la chiamata corrente, si potrebbe scrivere l'invocazione del metodo `rispondi` come

```
this.rispondi();
```

Omettere la parola chiave `this` e il punto quando si invoca un metodo non è un concetto nuovo. Questa stessa operazione è stata fatta per le variabili di istanza e funziona anche per i metodi della stessa classe. Se si invoca un metodo di una classe all'interno della definizione di un metodo di un'altra classe si deve, invece, includere il nome dell'oggetto e il punto.

Si noti, inoltre, che omettere il nome dell'oggetto che riceve l'invocazione al metodo è possibile solo se l'oggetto può essere espresso con la parola chiave `this`. Se l'oggetto che deve ricevere la chiamata è diverso dall'oggetto rappresentabile con la parola chiave `this`, è necessario includere il nome dell'oggetto e il punto.

LISTATO 8.13 Programma di dimostrazione della classe `Oracolo`.

MyLab

```
public class OracoloDemo {

   public static void main(String[] args) {
      Oracolo delphi = new Oracolo();
      delphi.parla();
   }
}
```

Esempio di output

```
Sono l'oracolo. Rispondero' a qualsiasi domanda che digiterai su una riga.
Qual e' la tua domanda?
Che ore sono?
Hmm, ho bisogno di aiuto su questo.
Scrivimi una riga di aiuto.
Guardami e dovresti trovar risposta
Grazie. Mi ha aiutato molto
Hai posto la domanda:
 Che ore sono?
Ora ecco la mia risposta:
 La risposta e' nel tuo cuore.
Vuoi pormi un'altra domanda?
si
Qual e' la tua domanda?
Qual e' il senso della vita?
Hmm, ho bisogno di aiuto su questo.
Scrivimi una riga di aiuto.
Chiedi al rivenditore d'auto
Grazie. Mi ha aiutato molto
Hai posto la domanda:
 Qual e' il senso della vita?
Ora ecco la mia risposta:
 Guardami e dovresti trovar risposta
Vuoi pormi un'altra domanda?
no
L'oracolo ora riposa.
```

Si possono avere, inoltre, metodi che invocano altri metodi i quali a loro volta invocano ulteriori metodi. Nella classe `Oracolo`, la definizione del metodo `rispondi` include anche chiamate ai metodi `cercaSuggerimento` e `aggiorna`, entrambi definiti nella classe `Oracolo`. Queste invocazioni non sono precedute dal nome di un oggetto per il motivo descritto sopra.

Si consideri la seguente invocazione di metodo tratta dal Listato 8.13:

```
delphi.parla();
```

La definizione del metodo `parla` include l'invocazione

```
rispondi();
```

che è equivalente a

```
this.rispondi();
```

Dato che l'oggetto che riceve la chiamata è `delphi`, questa istruzione è anche equivalente all'invocazione

```
delphi.rispondi();
```

La definizione di `rispondi` include anche le invocazioni

```
cercaSuggerimento();
```

e

```
aggiorna();
```

che sono equivalenti a

```
this.cercaSuggerimento();
```

e

```
this.aggiorna();
```

Dato che l'oggetto che riceve la chiamata è `delphi` queste invocazioni sono anche equivalenti a

```
delphi.cercaSuggerimento();
```

e

```
delphi.aggiorna();
```

Si possono avere metodi che invocano altri metodi in cascata. Le diverse invocazioni vengono gestite proprio come sono state descritte in questo paragrafo.

Omettere l'oggetto ricevente

Quando l'oggetto che riceve la chiamata in un metodo è rappresentato dalla parola chiave `this`, si può omettere la parola chiave `this` e il punto. Per esempio, il metodo:

```
public void rispondi() {
    ...
    this.cercaSuggerimento();
    ...
    this.aggiorna();
}
```

equivale a:

```
public void rispondi() {
    ...
    cercaSuggerimento();
    ...
    aggiorna();
}
```

Rendere privati i metodi ausiliari

Si osservi nuovamente il Listato 8.12. I metodi `rispondi`, `cercaSuggerimento` e `aggiorna` sono etichettati come `private` invece che `public`. Si ricordi che se un metodo è `private`, può essere invocato solo dai metodi della sua stessa classe. Perciò, in altre classi o programmi, la seguente invocazione di metodo non sarebbe valida e produrrebbe un errore di compilazione:

```
Oracolo mioOracolo = new Oracolo();
mioOracolo.rispondi();     //Non valida: rispondi è privato
```

La seguente invocazione di metodo sarebbe, invece, valida:

```
mioOracolo.parla();     //Valida
```

I metodi `rispondi`, `cercaSuggerimento` e `aggiorna` sono stati resi privati in quanto sono metodi ausiliari all'interno della definizione del metodo `parla`. Questo vuol dire che i metodi `rispondi`, `cercaSuggerimento` e `aggiorna` sono specifici di questa implementazione della classe e non dovrebbero essere disponibili agli utenti della classe stessa. Come verrà enfatizzato nel paragrafo "Incapsulamento", è bene mantenere privati tutti quegli elementi della classe che rappresentano la specifica implementazione della classe.

8.2.7 Incapsulamento

Nel Capitolo 1 si è detto che con il termine incapsulamento si intende il fatto di nascondere i dettagli di un componente software. A questo punto del testo si hanno le conoscenze sufficienti per comprendere più nel dettaglio cosa sia l'incapsulamento. L'incapsulamento consiste nel nascondere tutti i dettagli della definizione di una classe che non sono necessari per usare le istanze create da quella classe. Si consideri il seguente esempio.

Si supponga di voler guidare un'automobile. Dato questo compito, qual è il modo più utile per descrivere il funzionamento di un'automobile? Chiaramente dettagli come il numero di cilindri del motore, le percentuali in base alle quali si miscelano aria e benzina, il momento in cui deve esplodere questa miscela e il diametro dei condotti attraverso i quali vengono espulsi i gas risultanti, non sono utili per descrivere come guidare un'automobile. Per una persona che desidera guidare un'automobile, le informazioni più utili sono le seguenti.

- Se premi il pedale dell'acceleratore, l'automobile accelera.
- Se premi il pedale del freno, l'automobile rallenta fino a fermarsi.
- Se giri lo sterzo a destra o a sinistra, l'automobile curva di conseguenza.

Il principio di incapsulamento dice che quando si descrive un'automobile a qualcuno che vuole imparare a guidare, è bene fornire informazioni come quelle dell'elenco precedente. Nel contesto della programmazione, il significato di incapsulamento resta lo stesso. Per usare un certo componente software, un programmatore non ha bisogno di conoscere tutti i dettagli della sua definizione. In particolare, se il componente software è codificato in ben dieci pagine di codice, la descrizione che deve essere fornita a un programmatore

che intenda usarlo, dovrebbe essere molto più corta di dieci pagine, magari solo mezza pagina. Chiaramente questo è possibile solo se si scrive il software in modo che sia possibile descriverlo in così poco spazio.

Un'altra analogia che potrebbe aiutare nella comprensione: un'automobile ha alcuni elementi visibili, come i pedali e lo sterzo, e altri nascosti. L'automobile è "incapsulata": sono visibili solo i controlli necessari per guidarla, mentre i dettagli sono nascosti. Analogamente, un elemento software dovrebbe essere incapsulato in modo che siano visibili solo i controlli, mentre i dettagli devono essere nascosti. Il programmatore che usa il software non deve perciò preoccuparsi dei dettagli del software che usa. L'incapsulamento è molto importante perché semplifica il lavoro del programmatore che sfrutta il software incapsulato per scrivere altro software.

All'inizio di questo capitolo sono già state descritte alcune tecniche di incapsulamento quando si è introdotto l'*information hiding*. L'incapsulamento è una forma di *information hiding*. Affinché l'incapsulamento sia utile, la definizione di una classe deve essere tale per cui un programmatore possa usarla senza conoscerne i dettagli. L'incapsulamento deve separare la definizione di una classe in due parti: l'interfaccia[2] e l'implementazione. L'**interfaccia di una classe** (*class interface*) indica ai programmatori ciò di cui hanno bisogno per usare la classe nei loro programmi. L'interfaccia della classe consiste nell'intestazione dei suoi metodi pubblici e delle sue costanti pubbliche, insieme ai commenti che indicano al programmatore come usare i metodi e le costanti.

L'**implementazione di una classe** consiste di tutti gli elementi privati della classe, principalmente le variabili di istanza private e le definizioni dei metodi pubblici e privati. Si noti che l'interfaccia e l'implementazione di una classe non sono separate nel codice Java. Per esempio, nel Listato 8.10 è evidenziata l'interfaccia della classe `Acquisto`.

Quando si definisce una classe usando il principio di incapsulamento, occorre separare concettualmente l'interfaccia della classe dalla sua implementazione in modo che l'interfaccia sia una descrizione semplificata della classe. Un modo per pensare a questa separazione è quello di immaginare una parete tra l'implementazione e l'interfaccia; per attraversare questa parete si possono usare solo canali di comunicazione ben definiti e regolati. La Figura 8.3 illustra graficamente questa parete. Quando si usa l'incapsulamento per definire una classe in questo modo, si dice che la classe è **ben incapsulata**. Di seguito vengono riportate alcune importanti linee guida per definire una classe ben incapsulata.

- Si predisponga un commento prima della definizione della classe che descriva al programmatore cosa rappresenta la classe senza descrivere come lo fa. Se, per esempio, la classe rappresenta una somma di denaro, nel commento devono apparire termini quali Euro e centesimi e non il modo in cui questi sono rappresentati nella classe.
- Si dichiarino tutte le variabili di istanza della classe come private.
- Si forniscano metodi *get* pubblici per recuperare i dati in un oggetto. Si forniscano, inoltre, metodi pubblici per qualsiasi altra necessità di base di cui un programmatore potrebbe aver bisogno per gestire i dati della classe. Questi metodi potrebbero includere, per esempio, i metodi *set*.
- Si predisponga un commento prima di ogni intestazione di metodo pubblico che specifichi chiaramente come usare il metodo.

[2] La parola *interface* (letteralmente "interfaccia") ha anche un significato tecnico in Java come sarà illustrato nel Capitolo 11. Tuttavia, quando si utilizza il termine *class interface* (letteralmente "interfaccia della classe"), si intende qualcosa di diverso così come di seguito descritto.

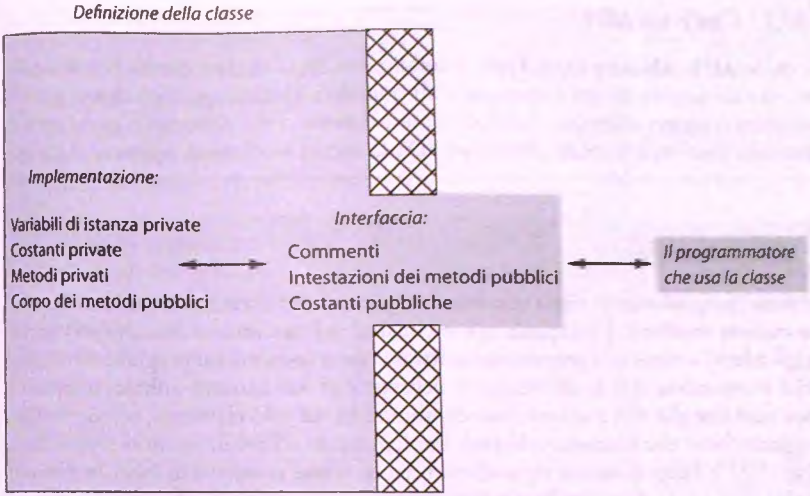


Figura 8.3 La definizione di una classe ben incapsulata.

- Si rendano privati i metodi ausiliari.
- Si scrivano commenti all'interno della classe per descrivere i dettagli implementativi.

I commenti inseriti nella definizione di una classe che descrivono come usare la classe e i suoi metodi pubblici, sono parte dall'interfaccia. Come indicato, questi commenti si trovano di solito prima della definizione della classe e prima di ogni definizione di metodo. Altri commenti chiariscono l'implementazione. Una buona regola da seguire quando si scrivono i commenti è di usare lo stile `/** ... */` per i commenti delle interfacce della classe e lo stile `//` per i commenti sull'implementazione.

Quando si usa l'incapsulamento per definire una classe, si dovrebbe essere in grado di poter modificare i dettagli implementativi della classe senza dover modificare alcun programma che usa la classe. Questo è un buon modo per verificare se la classe è ben incapsulata. Spesso si possono avere buone ragioni per modificare i dettagli implementativi di una classe. Per esempio, si potrebbe identificare un modo più efficiente per implementare un metodo in modo che la sua invocazione venga eseguita più velocemente. Si potrebbero voler modificare alcuni dettagli dell'implementazione senza modificare il modo in cui i metodi sono invocati e le attività di base che svolgono. Per esempio, se una classe rappresenta un conto in banca, si potrebbe cambiare la regola per assegnare una penalità per un conto in rosso.

FAQ L'interfaccia di una classe ha qualche relazione con il termine Application Programming Interface?

Il termine API o Application Programming Interface, è stato presentato nel Capitolo 1 quando è stata introdotta l'API di Java. L'API di una classe è essenzialmente la stessa cosa dell'interfaccia di una classe. Il termine API si incontra spesso quando si legge la documentazione delle librerie.

FAQ Cos'è un ADT

Il termine **ADT** (**Abstract Data Type**, letteralmente "tipo di dato astratto") è una specifica per un insieme di dati e operazioni su quei dati. Questa specifica descrive quali operazioni vengono effettuate, ma non come i dati sono memorizzati o come queste operazioni sono implementate. Perciò un ADT fa uso di tecniche di *information hiding*.

Incapsulamento

Il termine incapsulamento viene usato spesso quando si descrivono le tecniche di programmazione moderne. L'incapsulamento consiste nel nascondere (incapsulare) tutti i dettagli relativi a come un componente software opera fornendo al programmatore che userà il componente solo le informazioni necessarie al suo corretto utilizzo. Incapsulamento vuol dire che dati e azioni sono combinati in un solo elemento, in questo caso un oggetto classe, che nasconde i dettagli implementativi. Perciò, i termini *information hiding*, *ADT* e *incapsulamento* riguardano tutti lo stesso concetto di base. In termini operativi, l'idea è quella di fare in modo che il programmatore che sfrutta la classe non debba preoccuparsi del modo in cui è stata implementata.

8.2.8 Documentazione automatica con `javadoc`

I sistemi Java, incluso quello fornito da Oracle, di solito forniscono un programma chiamato `javadoc` che genera in modo automatico la documentazione per le interfacce delle classi. La documentazione generata indica agli altri programmatori tutto ciò che hanno bisogno di conoscere per poter usare le classi. Per generare documenti `javadoc`, è necessario scrivere i commenti in un modo particolare. Le classi definite in questo testo sono commentate in modo da poter usare `javadoc`, anche se, per motivi di spazio, sono stati specificati meno commenti del necessario. Se si commenta correttamente una classe, `javadoc` prenderà in input i commenti e produrrà un documento ben formattato che potrà essere letto comodamente per comprendere l'interfaccia delle classi. Per esempio, eseguendo `javadoc` sulla definizione della classe presente nel Listato 8.10, l'output sarebbe costituito unicamente dai testi inseriti nei commenti `/** ... */` e dalle intestazione dei metodi pubblici, con eventuale formattazione di fine riga e spaziature. Per leggere i documenti prodotti da `javadoc` è necessario un browser o un visualizzatore di pagine HTML.

Non è necessario usare `javadoc` per comprendere questo testo, né è necessario usare `javadoc` per scrivere programmi Java. Tuttavia `javadoc` è semplice e utile.

8.2.9 Diagrammi di classe UML

La Figura 8.2 presentata all'inizio del capitolo, mostra un esempio di diagramma UML. A questo punto del capitolo, è possibile comprendere tutti gli elementi e la notazione di quel diagramma. Tuttavia, invece di considerare quel diagramma, se ne introdurrà un altro. La Figura 8.4 contiene un diagramma UML per la classe `Acquisto` del Listato 8.10. I dettagli sono autoesplicativi, a parte per i segni + e -. Un segno più (+) prima di un nome

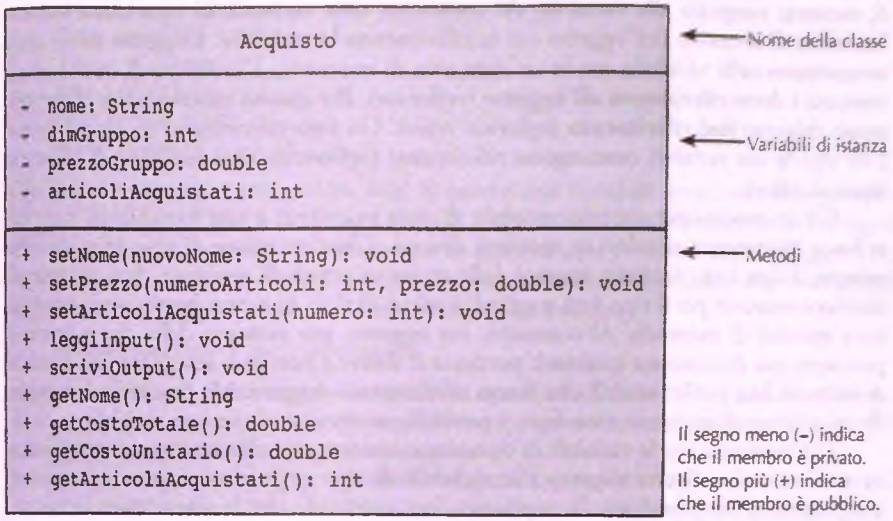


Figura 8.4 Diagramma delle classi UML per la classe `Acquisto` (Listato 8.10).

di una variabile di istanza o di metodo significa che l'attributo o il metodo sono pubblici. Un segno meno (-) significa che sono privati.

Si noti che il diagramma della classe contiene molti più elementi di quelli dell'interfaccia della classe, ma meno rispetto all'implementazione. Normalmente si scrive un diagramma delle classi prima di definire la classe in Java. Un diagramma delle classi è una sorta di schema sia dell'interfaccia della classe, sia dell'implementazione. È rivolto soprattutto al programmatore che deve implementare la classe. L'interfaccia della classe , invece, è rivolta al programmatore che utilizzerà la classe per produrre altro software.

8.3 Oggetti e riferimenti

Le variabili di tipo classe, come la variabile `unAcquisto` del Listato 8.11, si comportano in maniera molto diversa rispetto alle variabili di tipo primitivo. Le variabili di tipo classe sono nomi di oggetti. Però, contrariamente a quanto accade per le variabili di tipo primitivo, gli oggetti non sono i valori delle variabili di tipo classe. Il modo con cui si fa riferimento a un oggetto è più complicato rispetto a quello utilizzato per i valori di tipo primitivo. In questo paragrafo si parlerà del modo in cui una variabile di tipo classe fa riferimento al proprio oggetto e del comportamento dei parametri di tipo classe nei metodi.

MyLab

Video 8.3 Riferimenti e oggetti

8.3.1 Variabili di tipo classe

Le variabili di tipo classe forniscono un nome agli oggetti: vengono loro assegnati degli oggetti, ma il processo è diverso rispetto all'assegnamento di valori di tipo primitivo. Ciascuna variabile, sia essa di tipo primitivo o di tipo classe, è implementata come un'area di memoria. Se la variabile è di tipo primitivo, il suo valore è immagazzinato nell'area

di memoria assegnata alla variabile. Al contrario, una variabile di tipo classe contiene l'indirizzo di memoria dell'oggetto cui fa riferimento la variabile. L'oggetto stesso non è memorizzato nella variabile, ma in un'altra area di memoria. L'indirizzo di questa area di memoria è detto **riferimento all'oggetto** (*reference*). Per questo motivo, i tipi classe sono spesso chiamati **tipi riferimento** (*reference types*). Un tipo riferimento (o, tipo *reference*) è un tipo le cui variabili contengono riferimenti (*reference*), cioè indirizzi di memoria, invece di valori.

C'è un motivo per cui una variabile di tipo primitivo e una variabile di tipo classe fanno riferimento ai valori in maniera diversa. Ciascun valore di tipo primitivo, per esempio il tipo `int`, necessita sempre della stessa quantità di memoria. Java prevede dimensioni massime per il tipo `int` e quindi i valori di tipo `int` non possono superare una certa quantità di memoria. Al contrario, un oggetto, per esempio della classe `String`, può avere una dimensione qualsiasi; pertanto il sistema non può assegnare una quantità di memoria fissa per le variabili che fanno riferimento a oggetti. Ma poiché le dimensioni di un indirizzo di memoria sono fisse, è possibile memorizzarle in una variabile.

Dal momento che le variabili di tipo classe contengono riferimenti e si comportano in maniera molto diversa rispetto alle variabili di tipo primitivo, si possono osservare comportamenti sorprendenti. Si supponga, per esempio, che la classe `SpecieTerzaProva` sia definita come mostrato nel Listato 8.8 e che la prima parte di un programma inserito in un metodo `main` contenga le seguenti istruzioni:

```
SpecieTerzaProva specieKlingon = new SpecieTerzaProva();
SpecieTerzaProva specieTerrestre = new SpecieTerzaProva();
int n = 42;
int m = n;
```

In questo esempio, ci sono due variabili di tipo `int`: `n` e `m`. Entrambe hanno un valore pari a 42, ma se ne viene modificata una, l'altra continua ad avere valore pari a 42. Per esempio, se il programma continua con

```
n = 99;
System.out.println(n + " e " + m);
```

l'output prodotto sarebbe:

```
99 e 42
```

Fino a questo punto non ci sono sorprese; ma si supponga che il programma continui come segue:

```
specieKlingon.setSpecie("Bufalo Klingon", 10, 15);
specieTerrestre.setSpecie("Rinoceronte nero", 11, 2);
specieTerrestre = specieKlingon;
specieTerrestre.setSpecie("Elefante", 100, 12);
System.out.println("specie Terrestre:");
specieTerrestre.scriviOutput();
System.out.println("specie Klingon:");
specieKlingon.scriviOutput();
```

Si potrebbe pensare che nell'output stampato dal programma `specieKlingon` sia Bufalo Klingon e `specieTerrestre` sia Elefante. Al contrario, l'output è il seguente:

```
specie Terrestre:
Nome = Elefante
```

```
Popolazione = 100
Tasso crescita = 12.0%
specie Klingon:
Nome = Elefante
Popolazione = 100
Tasso crescita = 12.0%
```

Quello che è successo è molto semplice. Si hanno due variabili: `specieKlingon` e `specieTerrestre`. All'inizio del programma le due variabili fanno riferimento a due oggetti differenti. Ciascun oggetto è memorizzato in qualche area di memoria del computer e ha un indirizzo. Dato che le variabili di tipo classe memorizzano l'indirizzo di memoria degli oggetti e non gli oggetti stessi, l'istruzione di assegnamento:

```
specieTerrestre = specieKlingon;
```

copia l'indirizzo di memoria della variabile `specieKlingon` nella variabile `specieTerrestre` e fa sì che entrambe le variabili contengano lo stesso indirizzo di memoria e quindi facciano riferimento allo stesso oggetto.

Indipendentemente dalla variabile utilizzata (`specieKlingon` o `specieTerrestre`) per invocare `setSpecie`, l'invocazione al metodo è ricevuta dallo stesso oggetto e quindi viene modificato lo stesso oggetto. L'altro oggetto non è più accessibile dal programma. La Figura 8.5 illustra il comportamento appena descritto.

Si faccia attenzione al fatto che un indirizzo di memoria è certamente un numero, ma non è un valore `int`. Perciò non si provi a trattarlo come un intero qualsiasi.

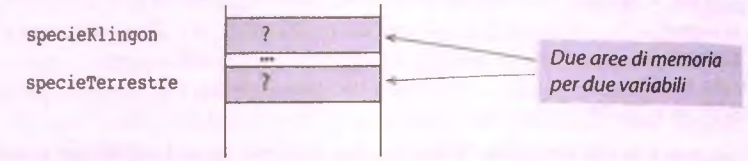
Le variabili di tipo classe contengono un indirizzo di memoria

Una variabile di tipo primitivo contiene un valore del tipo specificato. Una variabile di tipo classe non contiene un oggetto della classe, ma l'indirizzo dell'area di memoria in cui tale oggetto è memorizzato. Questo schema permette di usare una variabile di tipo classe come un nome per un oggetto di quella classe. Tuttavia, alcune operazioni, come = e ==, si comportano in maniera differente con le variabili di tipo classe rispetto a quelle di tipo primitivo.

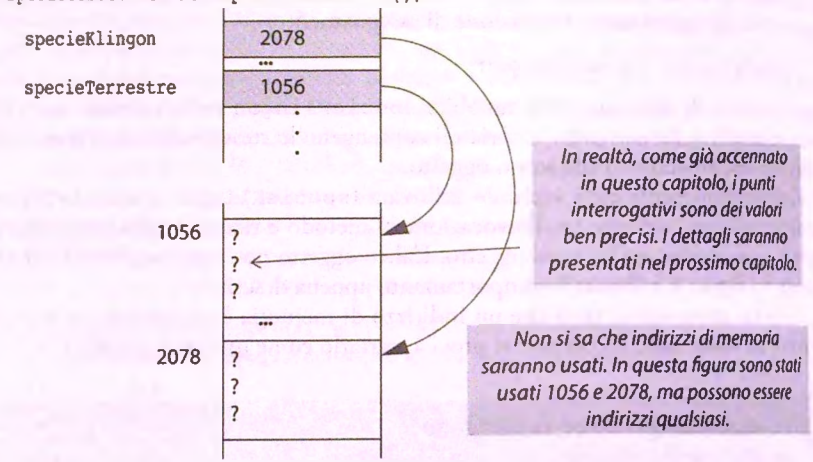
Gli indirizzi di memoria non sono esattamente numeri

Una variabile di tipo classe contiene un indirizzo di memoria. Sebbene un indirizzo di memoria sia un numero, una variabile di tipo classe non può essere usata come una variabile che memorizza un numero. Una proprietà importante di un indirizzo di memoria è che identifica un'area di memoria. Il fatto che gli indirizzi siano numeri, invece che lettere o colori o qualcos'altro è accidentale. Java, infatti, limita l'utilizzo di questa caratteristica, per evitare al programmatore di effettuare operazioni illecite, come ottenere accesso a un'area di memoria altrui o danneggiare il computer. Questo, inoltre, migliora la leggibilità del codice.

```
SpecieTerzaProva specieKlingon, specieTerrestre;
```




```
specieKlingon = new SpecieTerzaProva();
specieTerrestre= new SpecieTerzaProva();
```




```
specieKlingon.setSpecie("Bufalo Klingon", 10, 15);
specieTerrestre setSpecie("Rinoceronte nero", 11, 2);
```

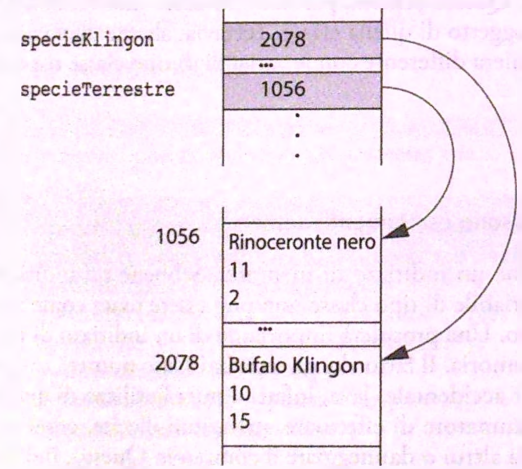
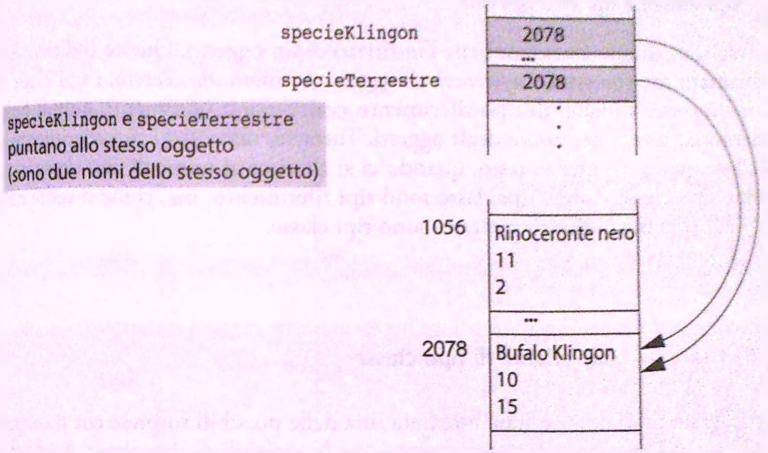


Figura 8.5 Comportamento di una variabile di tipo classe. (*continua*)

```
specieTerrestre = specieKlingon;
```




```
specieTerrestre.setSpecie("Elefante", 100, 12);
```

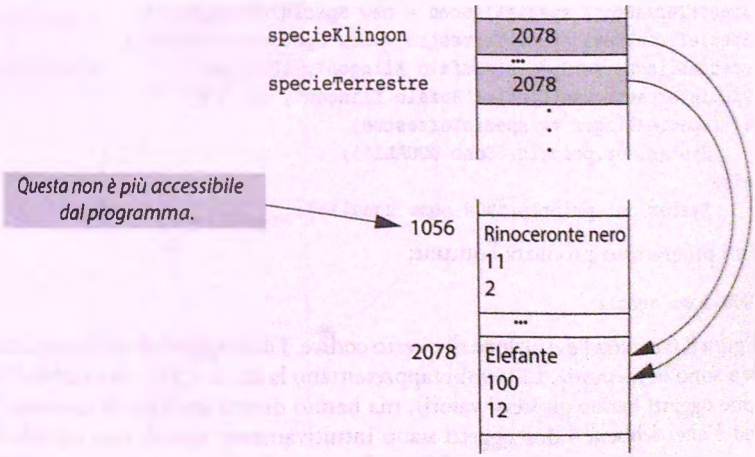


Figura 8.5 Comportamento di una variabile di tipo classe. (*segue*)

Tipi classe e tipi riferimento

Una variabile di tipo classe contiene l'indirizzo di un oggetto. Questo indirizzo è spesso chiamato riferimento (o *reference*) all'oggetto in memoria. Perciò, i tipi classe sono *tipi riferimento*. Variabili di tipo riferimento contengono riferimenti, quindi indirizzi di memoria, invece del valore degli oggetti. Tuttavia, non tutti i tipi riferimento sono tipi classe; perciò in questo testo, quando ci si riferisce al nome di una classe, si usa il termine tipo classe. Tutti i tipi classe sono tipi riferimento, ma, come si vedrà nel Capitolo 11, non tutti i tipi riferimento sono tipi classe.

Usare == con variabili di tipo classe

Nel paragrafo precedente è stata mostrata una delle possibili sorprese che si ottengono quando si usa l'operatore di assegnamento con le variabili di tipo classe. Anche il controllo dell'uguaglianza si comporta in un modo apparentemente strano. Si supponga che la classe `SpecieTerzaProva` sia definita come mostrato nel Listato 8.8 e che un programma contenga le seguenti istruzioni:

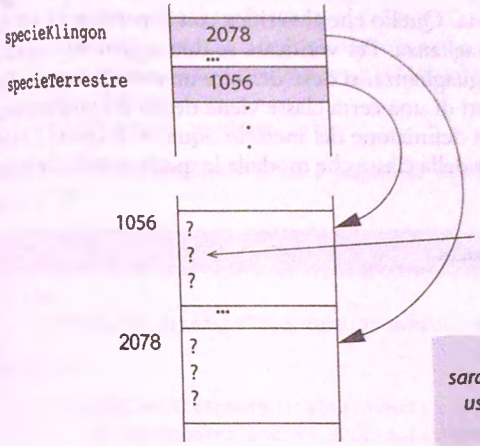
```
SpecieTerzaProva specieKlingon = new SpecieTerzaProva();
SpecieTerzaProva specieTerrestre = new SpecieTerzaProva();
specieKlingon.setSpecie("Bufalo Klingon", 10, 15);
specieTerrestre.setSpecie("Bufalo Klingon", 10, 15);
if (specieKlingon == specieTerrestre)
    System.out.println("Sono UGUALI");
else
    System.out.println("NON sono uguali");
```

Questo programma produrrà l'output:

```
NON sono uguali
```

La Figura 8.6 mostra l'esecuzione di questo codice. I due oggetti di tipo `SpecieTerzaProva` sono in memoria. Entrambi rappresentano la stessa specie (le variabili di istanza dei due oggetti hanno gli stessi valori), ma hanno diversi indirizzi di memoria. Il problema è che, sebbene i due oggetti siano intuitivamente uguali, una variabile di tipo classe in realtà contiene solo un indirizzo di memoria. L'operatore == controlla solo se gli indirizzi di memoria sono gli stessi. Questo operatore verifica un'uguaglianza di un certo tipo, quella degli indirizzi, ma non sempre questa uguaglianza è quella più utile. Ogni volta che si definisce una classe, si dovrebbe definire un metodo della classe, chiamato `equals` (letteralmente "uguale"), che verifica se gli oggetti sono uguali. Per uguaglianza, si intende che i due oggetti si trovano nello stesso stato: hanno, cioè, gli stessi valori nelle stesse variabili di istanza. Il prossimo paragrafo mostra come si realizza tale metodo.

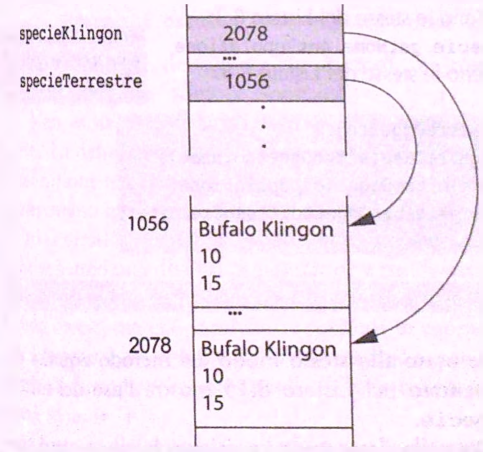
```
specieKlingon = new SpecieTerzaProva();
specieTerrestre = new SpecieTerzaProva();
```



In realtà, come già accennato in questo capitolo, i punti interrogativi sono dei valori ben precisi. I dettagli saranno presentati nel prossimo capitolo.

Non si sa che indirizzi di memoria saranno usati. In questa figura sono stati usati 1056 e 2078, ma possono essere indirizzi qualsiasi.

```
specieKlingon.setSpecie("Bufalo Klingon", 10, 15);
specieTerrestre.setSpecie("Bufalo Klingon", 10, 15);
```




```
if (specieKlingon == specieTerrestre)
   System.out.println("Sono UGUALI");
else
   System.out.println("NON sono uguali");
```

L'output è `NON sono uguali`*, poiché* `2078` *non è uguale a* `1056`*.*

Figura 8.6 Il pericolo di usare == con gli oggetti.

8.3.2 Definire un metodo `equals` per una classe

Quando si confrontano due oggetti usando l'operatore == si controlla se questi oggetti hanno lo stesso indirizzo di memoria. Quello che si verifica con l'operatore == non è ciò che si definisce intuitivamente uguaglianza. Per verificare se due oggetti sono uguali secondo una propria concezione di uguaglianza, si deve definire un metodo `equals`. Cosa si intende per uguaglianza tra oggetti di una certa classe viene deciso dal programmatore che definisce la classe, includendo la definizione del metodo `equals`. Il Listato 8.14 fornisce una nuova e ultima definizione della classe che modella le specie animali che include anche il metodo `equals`.

Lab

LISTATO 8.14 Definire un metodo `equals`.

```
import java.util.Scanner;

public class Specie {

    private String nome;
    private int popolazione;
    private double tassoCrescita;

    <Le definizioni dei metodi leggiInput, scriviOutput e
    prediciPopolazione vanno qui. Sono le stesse del Listato 8.3>
    <Le definizioni dei metodi setSpecie, getNome, getPopolazione
    e getTassoCrescita vanno qui. Sono le stesse del Listato 8.8>

    public boolean equals(Specie altroOggetto) {
        return (this.nome.equalsIgnoreCase(altroOggetto.nome))
            && (this.popolazione == altroOggetto.popolazione)
            && (this.tassoCrescita == altroOggetto.tassoCrescita);
    }
}
```

`equalsIgnoreCase` è un metodo della classe `String`.

Il metodo `equals` di una classe viene usato allo stesso modo del metodo `equals` della classe `String`. Il programma presentato nel Listato 8.15 mostra l'uso del metodo `equals` definito nella nuova classe `Specie`.

La definizione del metodo `equals` nella classe `Specie` ritiene che due oggetti `Specie` sono uguali se hanno lo stesso nome, ignorando maiuscole e minuscole, la stessa popolazione e lo stesso tasso di crescita. Per confrontare i nomi si utilizza il metodo `equalsIgnoreCase` della classe `String`. Questo metodo confronta due stringhe senza distinguere fra lettere maiuscole e minuscole. Come è stato indicato nel Capitolo 2, questo metodo viene fornito automaticamente da Java. L'operatore == viene invece usato per confrontare la popolazione e il tasso di crescita, dato che si tratta di valori di tipo primitivo.

Si noti che il metodo `equals` del Listato 8.14 restituisce sempre il valore `true` o il valore `false`, il valore restituito è quindi di tipo `boolean`. L'istruzione `return` sembra strana, ma non è altro che un'espressione booleana come quelle che potrebbero essere usate in un'istruzione `if-else`. Per comprendere meglio il comportamento del metodo

:quals del Listato 8.14, si noti che la sua definizione potrebbe essere espressa dal seguen-
e pseudocodice:

```
if ((this.nome.equalsIgnoreCase(altroOggetto.nome))
        && (this.popolazione == altroOggetto.popolazione)
        && (this.tassoCrescita == altroOggetto.tassoCrescita))
    allora restituisci true
    in caso contrario restituisci false
```

)uesta modifica renderebbe il codice seguente (estratto dal programma presentato nel
istato 8.15):

```
if (s1.equals(s2))
    System.out.println("Corrispondono secondo il metodo equals.");
else
    System.out.println("Non corrispondono secondo il metodo equals.");
```

quivalente a:

```
if ((s1.nome.equalsIgnoreCase(s2.nome))
        && (s1.popolazione == s2.popolazione)
        && (s1.tassoCrescita == s2.tassoCrescita))
    System.out.println("Corrispondono secondo il metodo equals.");
else
    System.out.println("Non corrispondono secondo il metodo equals.");
```

na descrizione più dettagliata dei metodi che restituiscono valori di tipo `boolean` è
rnita nel paragrafo "Metodi booleani".

Non esiste una definizione unica di `equals` che possa essere usata per tutti gli
ggetti. La definizione del metodo `equals` dipende da come si intende usare la classe.
definizione nel Listato 8.14 indica che due oggetti della classe `Specie` sono uguali se
ppresentano specie con lo stesso nome, la stessa popolazione e lo stesso tasso di crescita.
altri contesti si potrebbe ritenere che due specie siano uguali se hanno lo stesso nome,
che se hanno una diversa popolazione e un diverso tasso di crescita. Questa seconda de-
izione porterebbe a considerare uguali due oggetti che rappresentano valori riguardanti
stessa specie, ma che sono stati registrati in momenti diversi.

Bisognerebbe sempre utilizzare `equals` per identificare il nome del metodo che
rifica se due oggetti sono uguali. È bene non usare altri nomi (per esempio `uguali` o
ual senza la "`s`").

Se non si definisce un metodo `equals` per una classe, Java ne crea automaticamente
o con una definizione di default; tuttavia questo metodo potrebbe non comportarsi nel
odo voluto. Perciò è meglio definire metodi `equals` personalizzati.

ISTATO 8.15 Dimostrazione del metodo `equals`.

MyLab

```
ublic class SpecieEqualsDemo {

  public static void main(String[] args) {

    Specie s1 = new Specie(), s2 = new Specie();
    s1.setSpecie("Bufalo Klingon", 10, 15);
    s2.setSpecie("Bufalo Klingon", 10, 15);
```

```
        if(s1 == s2)
            System.out.println("Corrispondono secondo ==.");
        else
            System.out.println("Non corrispondono secondo ==.");
        if(s1.equals(s2))
            System.out.println("Corrispondono secondo il metodo equals.");
        else
            System.out.println("Non corrispondono secondo il metodo equals.");
        System.out.println("Ora cambiamo un Klingon in lettere minuscole.");
        s2.setSpecie("bufalo klingon", 10, 15); //Usa lettere minuscole

        if (s1.equals(s2))
            System.out.println("Corrispondono secondo il metodo equals.");
        else
            System.out.println("Non corrispondono secondo il metodo equals.");
    }
}
```

Esempio di output

```
Non corrispondono secondo ==.
Corrispondono secondo il metodo equals.
Ora cambiamo un Klingon in lettere minuscole.
Corrispondono secondo il metodo equals.
```

ESEMPIO DI PROGRAMMAZIONE
LA CLASSE `Specie`

La versione finale della classe `Specie`, come definita nel Listato 8.14, è riportata anche nel Listato 8.16 in cui, però, sono stati inclusi tutti i dettagli, in modo da presentare un esempio completo. È stato anche riscritto il metodo `equals`, omettendo la parola chiave `this`. La definizione del metodo `equals` è perfettamente equivalente a quella definita nel Listato 8.14. Infine, la Figura 8.7 illustra il diagramma della classe `Specie`.

LISTATO 8.16 La classe `Specie` completa.

Questa è la stessa definizione di classe del Listato 8.14, ma mostra tutti i dettagli.

```
import java.util.Scanner;

public class Specie {

    private String nome;
    private int popolazione;
    private double tassoCrescita;

    public void leggiInput() {
        Scanner tastiera = new Scanner(System.in);
        System.out.println("Qual e' il nome della specie?");
        nome = tastiera.nextLine();
        System.out.println("A quanto ammonta la popolazione?");
        popolazione = tastiera.nextInt();
```

```
        System.out.println("Inserisci il tasso di crescita " +
                           "(% crescita per anno):");
        tassoCrescita = tastiera.nextDouble();
    }

    public void scriviOutput() {
        System.out.println("Nome = " + nome);
        System.out.println("Popolazione = " + popolazione);
        System.out.println("Tasso crescita = " + tassoCrescita + "%");
    }

    /**
    Restituisce una proiezione della popolazione dopo un numero
    specificato di anni
    */
    public int prediciPopolazione(int anni) {
        int risultato = 0;
        double totalePopolazione = popolazione;
        int contatore = anni;

        while ((contatore > 0) && (totalePopolazione > 0)) {
            totalePopolazione = (totalePopolazione +
                (tassoCrescita / 100) * totalePopolazione);
            contatore--;
        }
        if (totalePopolazione > 0)
            risultato = (int)totalePopolazione;
        return risultato;
    }

    public void setSpecie(String nuovoNome, int nuovaPopolazione,
                          double nuovoTassoCrescita) {
        nome = nuovoNome;
        if (nuovaPopolazione >= 0)
            popolazione = nuovaPopolazione;
        else {
            System.out.println("ERRORE: si sta usando un numero negativo " +
                               "per la popolazione.");
            System.exit(0);
        }
        tassoCrescita = nuovoTassoCrescita;
    }

    public String getNome() {
        return nome;
    }

    public int getPopolazione() {
        return popolazione;
    }
```

```
    public double getTassoCrescita() {
        return tassoCrescita;
    }

    public boolean equals(Specie altroOggetto) {
        return (nome.equalsIgnoreCase(altroOggetto.nome))
            && (popolazione == altroOggetto.popolazione)
            && (tassoCrescita == altroOggetto.tassoCrescita);
    }
}
```

Questa versione di `equals` è equivalente alla versione del Listato 8.14. Qui, la parola riservata `this` è omessa, perché implicita.

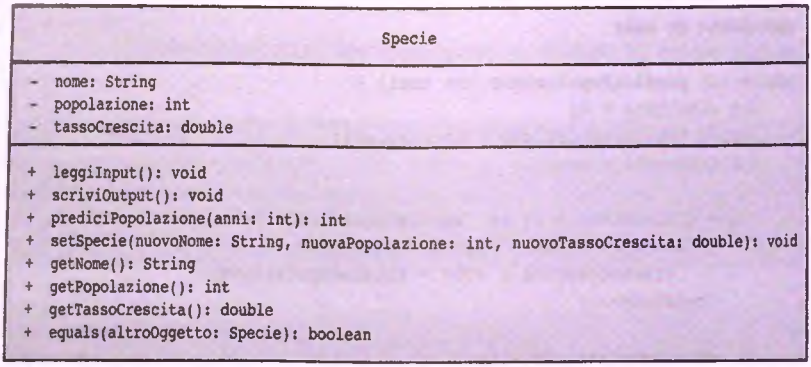


Figura 8.7 Diagramma della classe `Specie`.

8.3.3 Metodi booleani

Come specificato nel Capitolo 5, i metodi possono restituire valori di tipo `boolean`: basta specificare un valore restituito di tipo `boolean` e usare un'espressione booleana nell'istruzione `return`. Uno di questi metodi è già stato presentato nel Listato 8.16 per la classe `Specie`. Questo metodo valuta semplicemente l'espressione booleana nell'istruzione `return`, calcolando un valore `true` o `false`. Il metodo poi restituisce tale valore. Come si è visto in precedenza, un'invocazione del metodo `equals` può essere usata all'interno di un'istruzione `if`, `while` o di un'altra istruzione che richiede un'espressione booleana. Si può memorizzare il valore restituito dal metodo `equals` o da un qualsiasi altro metodo che restituisce un valore booleano, in una variabile di tipo `boolean`. Per esempio:

```
Specie s1 = new Specie(), s2 = new Specie();
<Codice che assegna un valore a s1 e s2.>
boolean sonoUguali = s1.equals(s2);
< Altro codice.>
```

```
if(sonoUguali)
    System.out.println("Sono uguali.");
else
    System.out.println("Non sono uguali.");
```

Il seguente potrebbe essere un altro esempio di un metodo che si potrebbe aggiungere alla definizione della classe `Specie` nel Listato 8.16 e che restituisce un valore booleano.

```
/**
Precondizione: Questo oggetto e l'argomento altraSpecie devono
avere un valore per la popolazione.
Restituisce true se la popolazione di questo oggetto è maggiore
della popolazione di altraSpecie; altrimenti restituisce false.
*/
public boolean isPopolazionePiuGrandeDi(Specie altraSpecie) {
    return popolazione > altraSpecie.popolazione;
}
```

Il metodo `isPopolazioenPiuGrandeDi` può essere usato in maniera simile al metodo `equals`. Per esempio, un programma potrebbe contenere le righe seguenti:

```
Specie s1 = new Specie(), s2 = new Specie();
<Codice che assegna un valore a s1 e s2.>
String nomeMaggiore = s1.getNome();
if (s2.isPopolazionePiuGrandeDi(s1)) {
    nomeMaggiore = s2.getNome();
}
System.out.println(nomeMaggiore + " ha la popolazione piu' grande.");
```

Anche il metodo che segue potrebbe essere aggiunto alla classe `Specie` nel Listato 8.16:

```
/**
Precondizione: La variabile popolazione di questo oggetto
ha un valore assegnato.
Restituisce true se la popolazione di questo oggetto e' zero,
altrimenti restituisce false.
*/
public boolean isEstinta() {
    return popolazione == 0;
}
```

Il codice seguente, invece, potrebbe essere incluso in un programma:

```
Specie s1 = new Specie();
<Codice per assegnare un valore a s1.>
if (s1.isEstinta())
    System.out.println(s1.getNome() + " e' estinto.");
else
    System.out.println(s1.getNome() + " e' ancora con noi.");
```

Dare un nome ai metodi booleani

Quando il metodo `isEstinta` viene invocato all'interno di un'istruzione `if`, se ne può comprendere il significato semplicemente leggendolo. Il termine *is* in inglese vuol dire "è", quindi *is estinta* vuol dire *è estinta*. Per convenzione si usa il termine `is` come prefisso nei metodi booleani per chiarirne lo scopo. Quindi, leggendo l'istruzione:

```
if(s1.isEstinta())
```

si capisce che un'azione viene intrapresa solo nel caso in cui la specie `s1` sia estinta.

8.3.4 Test di unità

Fino ad ora i programmi di esempio sono stati provati eseguendoli, inserendo dei dati di input e controllando visivamente i risultati per verificare che l'output coincidesse con quello atteso. Questo approccio è adatto per programmi semplici, ma diventa in genere insufficiente per programmi complessi. In un programma complesso, esistono, generalmente, così tante combinazioni di valori di input che occorrerebbe troppo tempo per verificare manualmente la correttezza del risultato per ognuna di esse. Inoltre, è possibile che le modifiche al codice producano effetti collaterali inattesi. Per esempio, una modifica per la correzione di un errore potrebbe a sua volta introdurne altri.

Un approccio alla soluzione di questo problema consiste nella scrittura di **test di unità** (*unit test*). Secondo questa metodologia, il programmatore verifica la correttezza di singole unità di codice. Un'unità è generalmente costituita da un metodo, ma potrebbe anche essere una classe o qualunque altra porzione di codice.

Una collezione di test di unità è detta **suite di test** (*test suite*). Generalmente, ogni test è automatizzato, così che non è richiesto alcun intervento da parte del programmatore. L'automazione è un aspetto importante, perché è preferibile avere test che possano essere eseguiti frequentemente e rapidamente. Ciò consente di eseguire i test ripetutamente, per esempio una volta al giorno o ogni volta che il codice viene modificato, in modo da assicurarsi che tutto continui a funzionare. Il procedimento consistente nell'esecuzione ripetuta dei test è detto **test di regressione** (*regression testing*).

Si consideri un semplice caso di test per la classe `Specie` del Listato 8.16. La prima prova potrebbe consistere nella verifica che il nome, la popolazione iniziale e il tasso di crescita vengano impostati correttamente dal metodo `setSpecie`. Ciò può essere realizzato creando un oggetto di tipo `Specie`, invocando su di esso il metodo `setSpecie` e verificando che tutti i valori siano corretti:

```
Specie provaSpecie = new Specie();
// Prova il metodo setSpecie
provaSpecie.setSpecie ("Leone", 100, 50);
if (provaSpecie.getNome().equals("Leone") &&
    (provaSpecie.getPopolazione() == 100) &&
    (provaSpecie.getTassoCrescita() >= 49.99) &&
    (provaSpecie.getTassoCrescita() <= 50.01)) {
    System.out.println("Superato: test setSpecie.");
} else {
    System.out.println("FALLITO: test setSpecie.");
}
```

Poiché `getTassoCrescita` restituisce un valore di tipo `double`, non sarebbe corretto utilizzare `provaSpecie.getTassoCrescita() == 50`

In questo esempio, sono stati forniti dati di input per il caso di test, è stato eseguito il codice e si è verificato che i risultati fossero in accordo con quelli attesi. Se ci fosse un errore nel codice del metodo (per esempio, se ci si fosse dimenticati di impostare il valore della variabile di istanza relativa al tasso di crescita nel metodo `setSpecie`), il test individuerebbe il problema.

Spesso la scrittura dei casi di test è più complessa. Si consideri per esempio il problema di verificare la correttezza del metodo `prediciPopolazione`. Di nuovo, è necessario determinare dei dati di input appropriati e verificare se l'unità da analizzare produce l'output atteso. Un tipo di test che si potrebbe considerare è il cosiddetto **test negativo**. Si tratta di un tipo di test volto a verificare che il programma non termini in modo inatteso se gli vengono forniti dati di input con caratteristiche diverse da quelle attese. Per esempio, se il numero di anni passato al metodo `prediciPopolazione` è negativo, ci si aspetterebbe che il metodo restituisca la popolazione iniziale. Inoltre, si potrebbe provare il metodo con input pari a 1 e 5 anni nel futuro. Un caso di test di questo tipo è riportato nel Listato 8.17.

Una pratica comune è quella di scrivere i test utilizzando l'istruzione `assert`. Per esempio, al posto del blocco `if-else` si potrebbe scrivere:

```
assert (provaSpecie.getPopolazione() == 100);
```

Non si commetta però l'errore di pensare che la correttezza di un programma sia assicurata se tutti i test vengono superati. È sempre possibile che esista una combinazione di dati di input che non è stata verificata nei test e che farà fallire l'esecuzione del programma. Potrebbero anche essere presenti errori nell'integrazione fra loro delle singole unità analizzate. Tuttavia, una campagna di test superata con successo suggerisce solitamente che almeno le funzionalità di base funzionino correttamente.

Idealmente, i casi di test dovrebbero essere mantenuti separati dall'implementazione della classe da analizzare. Nel semplice esempio del Listato 8.17 è stata creata una classe separata con un metodo `main` per provare la classe `Specie`. Per progetti più complessi si potrebbe considerare l'utilizzo di un framework di test come JUnit, progettato per facilitare l'organizzazione e l'esecuzione delle suite di test.

LISTATO 8.17 Semplici test per la classe `Specie`.

MyLab

```
public class SpecieTest {
    public static void main(String[] args) {
        Specie provaSpecie = new Specie();

        // Prova il metodo setSpecie
        provaSpecie.setSpecie("Leone", 100, 50);
        if (provaSpecie.getNome().equals("Leone") &&
                (provaSpecie.getPopolazione() == 100) &&
                (provaSpecie.getTassoCrescita() >= 49.99) &&
                (provaSpecie.getTassoCrescita() <= 50.01)) {
            System.out.println("Superato: test setSpecie.");
        } else {
            System.out.println("FALLITO: test setSpecie.");
        }
```

```
        // Prova il metodo prediciPopolazione
        if ((provaSpecie.prediciPopolazione(-1) == 100) &&
                (provaSpecie.prediciPopolazione(1) == 150) &&
                (provaSpecie.prediciPopolazione(5) == 759)) {
            System.out.println("Superato: test prediciPopolazione.");
        } else {
            System.out.println("FALLITO: test prediciPopolazione.");
        }
    }
}
```

Output

```
Superato: test setSpecie.
Superato: test prediciPopolazione.
```

8.3.5 Parametri di tipo classe

I parametri di tipo classe di un metodo si comportano in maniera diversa rispetto ai valori di tipo primitivo. La differenza è già stata presentata nella parte dedicata all'operatore di assegnamento per gli oggetti. I seguenti paragrafi permettono di capire come funzionano i parametri di tipo classe.

In primo luogo si ricordi come l'operatore di assegnamento si comporta con gli oggetti. Si consideri, a tal proposito, la classe `Specie` definita nel Listato 8.16 e si osservi il codice seguente:

```
Specie specie1 = new Specie();
specie1.leggiInput();
Specie specie2 = specie1;
```

Quando si usa un operatore di assegnamento con oggetti di tipo classe, si copia un indirizzo di memoria. Perciò, come già descritto precedentemente, `specie1` e `specie2` sono ora due nomi che fanno riferimento allo stesso oggetto.

In secondo luogo, si ricordi come funzionano i parametri di tipo primitivo. La definizione del metodo `prediciPopolazione` nel Listato 8.16 inizia come segue:

```
public int prediciPopolazione(int anni) {
    int risultato = 0;
    double totalePopolazione = popolazione;
    int contatore = anni;
    ...
```

Nel Capitolo 5 si è detto che il parametro formale `anni` è una variabile locale. Quando si invoca il metodo `prediciPopolazione`, la variabile locale `anni` viene inizializzata con il valore dell'argomento passato al metodo. Quindi, per esempio, quando si usa la seguente invocazione nel Listato 8.9:

```
int popolazioneFutura = specieDelMese.prediciPopolazione(numeroAnni);
```

il parametro `anni` viene inizializzato con il valore di `numeroAnni`. L'effetto che si ottiene è come quello di un assegnamento temporaneo come:

```
anni = numeroAnni;
```

inserito nella definizione del metodo. In altre parole, è come se la definizione del metodo

prediciPopolazione fosse stata cambiata come segue:

```
public int prediciPopolazione(int anni) {
    anni = numeroAnni;     //Non si può fare, ma Java
                           //si comporta come se venisse fatto
    int risultato = 0;
    double totalePopolazione = popolazione;
    int contatore = anni;
    ...
```

Questo preambolo è molto utile per capire come si comporta un metodo quando si passano degli argomenti di tipo classe. I parametri di tipo classe si comportano come parametri di tipo primitivo, ma dato che l'operatore di assegnamento ha un significato diverso l'effetto è molto diverso.

Si consideri la seguente invocazione del metodo `equals` usata nel Listato 8.15:

```
if(s1.equals(s2))
    System.out.println("Corrispondono secondo il metodo equals.");
else
    System.out.println("Non corrispondono secondo il metodo equals.");
```

In questa invocazione di metodo, `s2` è un argomento di tipo `Specie` definito nel Listato 8.16. Di seguito viene riportata la definizione di `equals` fornita nel Listato 8.16:

```
public boolean equals(Specie altroOggetto) {
    return (nome.equalsIgnoreCase(altroOggetto.nome))
        && (popolazione == altroOggetto.popolazione)
        && (tassoCrescita == altroOggetto.tassoCrescita);
}
```

Quando il metodo `equals` viene invocato in `s1.equals(s2)` si comporta come se all'inizio della definizione del metodo fosse stata temporaneamente inserita la seguente istruzione di assegnamento:

```
altroOggetto = s2;
```

In altre parole, la definizione del metodo corrisponderebbe alla seguente:

```
public boolean equals(Specie altroOggetto) {
    altroOggetto = s2;     //Non si può fare, ma Java
                           //si comporta come se venisse fatto
    return (nome.equalsIgnoreCase(altroOggetto.nome))
        && (popolazione == altroOggetto.popolazione)
        && (tassoCrescita == altroOggetto.tassoCrescita);
}
```

Si ricordi, tuttavia, che questa istruzione di assegnamento si limita a copiare l'indirizzo di memoria di `s2` nella variabile `altroOggetto`; in questo modo `altroOggetto` diventa solo un altro nome per l'oggetto cui fa riferimento `s2`. Perciò, qualsiasi operazione sull'oggetto `altroOggetto` è come se venisse fatta sull'oggetto `s2`. In altre parole, è come se il metodo effettuasse le seguenti azioni:

```
return (nome.equalsIgnoreCase(s2.nome))
    && (popolazione == s2.popolazione)
    && (tassoCrescita == s2.tassoCrescita);
```

Si noti che qualsiasi azione intrapresa con un parametro formale di tipo classe, in questo esempio `altroOggetto`, viene anche intrapresa con l'argomento passato nell'invocazione di metodo, in questo caso `s2`. Perciò l'argomento usato nell'invocazione di metodo è l'oggetto su cui si opera e potrebbe essere modificato nell'invocazione del metodo.
Nel caso del metodo `equals`, l'effetto di questo meccanismo di passaggio di parametri non è diverso da quello che succede quando si passano tipi primitivi. Con altri metodi, invece, la differenza è più forte. Il prossimo paragrafo presenta un esempio che chiarisce meglio la differenza esistente fra parametri di tipo classe e parametri di tipo primitivo.

Chiamata per riferimento

Il meccanismo di passaggio di parametri usato nelle invocazioni che riguardano argomenti di tipo classe fa sì che alcuni programmatori chiamino queste invocazioni **chiamate per riferimento** (*call-by-reference*). Altri dicono che questa terminologia non è corretta. Il problema è che esiste più di una definizione di *chiamata per riferimento*. Inoltre, in Java i parametri di tipo classe hanno un comportamento leggermente diverso dei parametri usati nelle chiamate per riferimento negli altri linguaggi. Proprio per questo motivo, in questo testo il termine "chiamata per riferimento" non viene adottato. È importante capire come funzionano i parametri di tipo classe, indipendentemente dal modo in cui vengono chiamati.

Parametri di tipo classe

I parametri formali sono definiti tra parentesi dopo il nome del metodo all'inizio della definizione del metodo. Un parametro formale di tipo classe è una variabile locale che contiene l'indirizzo di memoria di un oggetto del tipo classe specificato dal parametro. Quando viene invocato il metodo, il parametro viene inizializzato con l'indirizzo di memoria dell'argomento passato nell'invocazione del metodo. In termini più semplici, questo vuol dire che il parametro formale è un nome alternativo per l'oggetto fornito come argomento in un'invocazione di metodo.

Un metodo può modificare un oggetto passato come argomento

Un oggetto che viene passato come argomento in un metodo può essere alterato. Tuttavia, l'oggetto non può essere sostituito con un altro oggetto. L'esempio di programmazione che segue dimostra questi punti.

ESEMPIO DI PROGRAMMAZIONE
PARAMETRI DI TIPO CLASSE VS. PARAMETRI DI TIPO PRIMITIVO

Si supponga di aggiungere tre metodi alla classe `Specie` per definire una nuova classe, `SpecieDemo`, mostrata nel Listato 8.18, cui andrebbero aggiunti tutti i metodi definiti nel Listato 8.16, ma che per gli scopi di questo esempio non sono necessari.

LISTATO 8.18 Una classe dimostrativa.

MyLab

```
import java.util.Scanner;

/**
Questo esempio della classe Specie serve solo per mostrare
la differenza tra parametri di tipo classe e parametri di tipo primitivo.
*/
public class SpecieDemo {

    private String nome;
    private int popolazione;
    private double tassoCrescita;

    /**
    Prova ad assegnare all'argomento variabileInt il valore
    di popolazione, ma il parametro primitivo non può essere
    modificato
    */
    public void provaACambiare(int variabileInt) {
        variabileInt = this.popolazione;
    }

    /**
     Prova a far sì che altroOggetto referenzi l'oggetto this.
     Ma non si possono sostituire gli argomenti di tipo classe.
    */
    public void provaASostituire(SpecieDemo altroOggetto) {
        altroOggetto = this;
    }

    /**
     Cambia i dati in altroOggetto con quelli dell'oggetto this,
     che non viene modificato.
    */
    public void cambia(SpecieDemo altroOggetto) {
        altroOggetto.nome = this.nome;
        altroOggetto.popolazione = this.popolazione;
        altroOggetto.tassoCrescita = this.tassoCrescita;
```

```
    <Il resto della definizione della classe è analogo a quello della classe Specie nel Listato 8.16>
}
```

Si osservi il metodo `provaACambiare` della classe `SpecieDemo`. Questo metodo ha un parametro formale di tipo primitivo `int`. All'interno del corpo del metodo viene effettuato un assegnamento a questo parametro. Il programma presentato nel Listato 8.19 invoca il metodo `provaACambiare` passandogli l'argomento di tipo `int` `unaPopolazione`. Tuttavia, l'assegnamento effettuato nel corpo del metodo non ha alcun effetto sull'argomento `unaPopolazione`. Dato che le variabili di tipo primitivo contengono valori concreti, non indirizzi di memoria, il meccanismo di invocazione per valore di Java copia il valore dell'argomento nel parametro, che è una variabile locale. Perciò, tutti i cambiamenti che il metodo effettua su quel parametro si limitano a questa variabile e non vengono applicati all'argomento.

Il metodo `provaASostituire` ha un parametro di tipo `SpecieDemo`, perciò è di tipo classe. Il meccanismo di chiamata per valore di Java copia nel parametro il valore dell'argomento. Tuttavia, dato che sia l'argomento, sia il parametro sono di tipo classe, nel parametro viene copiato l'indirizzo di memoria dell'argomento. L'istruzione di assegnamento presente nel corpo del metodo assegna, quindi, un nuovo valore al parametro. Questo nuovo valore è l'indirizzo dell'oggetto che riceve la chiamata, come indicato dalla parola chiave `this`. Come nel caso del metodo `provaACambiare`, questo assegnamento non si riflette sull'argomento. Perciò `s2`, l'oggetto di tipo `SpecieDemo` (che il programma presentato nel Listato 8.19 passa al metodo `provaASostituire`) non viene modificato.

Infine, il tipo del parametro del metodo `cambia` è `SpecieDemo`. Il programma presentato nel Listato 8.19 invoca il metodo `cambia` passandogli l'argomento `s2`. Le istruzioni di assegnamento all'interno del corpo del metodo cambiano il valore delle variabili di istanza dell'argomento `s2`. Di conseguenza, un metodo può cambiare lo stato di un argomento di tipo classe.

Come si può notare, i parametri di tipo classe sono più versatili dei tipi primitivi. I parametri di tipo primitivo passano valori a un metodo, ma un metodo non può cambiare il valore di variabili di tipo primitivo che gli vengono passate come argomento. D'altro canto, i parametri di tipo classe possono essere usati non solo per fornire informazioni a un metodo, ma un metodo può anche cambiare lo stato di un oggetto passato come argomento. Il metodo tuttavia, non può sostituire l'oggetto che viene passato come argomento con un altro oggetto.

Differenze tra tipi primitivi e tipi classe

Un metodo non può modificare il valore di un argomento di tipo primitivo che gli viene passato. Inoltre, un metodo non può sostituire un oggetto che riceve come argomento con un altro oggetto. D'altro canto, un metodo può modificare i valori delle variabili di istanza di un argomento di tipo classe.

LISTATO 8.19 Parametri di tipo classe vs. parametri di tipo primitivo.

MyLab

```
public class ParametriDemo {

    public static void main(String[] args) {
        SpecieDemo s1 = new SpecieDemo(), s2 = new SpecieDemo();
        s1.setSpecie("Bufalo Klingon", 10, 15);
        int unaPopolazione = 42;
        System.out.println("unaPopolazione PRIMA di invocare provaACambiare: " +
                           unaPopolazione);
        s1.provaACambiare(unaPopolazione);
        System.out.println("unaPopolazione DOPO aver invocato provaACambiare: " +
                           unaPopolazione);
        s2.setSpecie("Furetto", 90, 56);
        System.out.println("s2 PRIMA di invocare provaASostituire:");
        s2.scriviOutput();
        s1.provaASostituire(s2);
        System.out.println("s2 DOPO aver invocato provaASostituire:");
        s2.scriviOutput();
        s1.cambia(s2);
        System.out.println("s2 DOPO ave invocato cambia:");
        s2.scriviOutput();
    }
}
```

Esempio di output

```
unaPopolazione PRIMA di invocare provaACambiare: 42
unaPopolazione DOPO aver invocato provaACambiare: 42
s2 PRIMA di invocare provaASostituire:
Nome = Furetto
Popolazione = 90
Tasso crescita = 56.0%
s2 DOPO aver invocato provaASostituire:
Nome = Furetto
Popolazione = 90
Tasso crescita = 56.0%
s2 DOPO ave invocato cambia:
Nome = Bufalo Klingon
Popolazione = 10
Tasso crescita = 15.0%
```

Il valore di un argomento di tipo primitivo non può essere modificato.

Un argomento di tipo classe non può essere sostituito.

Lo stato di un argomento di tipo classe può essere modificato

8.4 Riepilogo

- Le classi hanno variabili di istanza per memorizzare dati e definizioni di metodi che eseguono azioni.
- Classi, variabili di istanza e definizioni di metodo possono essere pubblici o privati. Quando sono pubblici possono essere usati da qualunque posizione. Una variabile

di istanza privata non può essere usata al di fuori della definizione di classe. Tuttavia, può essere usata all'interno delle definizioni dei metodi della classe stessa. Un metodo privato non può essere invocato al di fuori della classe in cui è definito. Tuttavia, può essere invocato all'interno della definizione di altri metodi della stessa classe.

- Le variabili di istanza dovrebbero essere private, anche se questo implica che possono essere usate solo all'interno della classe in cui sono definite.
- I metodi d'accesso o metodi *get*, restituiscono il valore di una variabile di istanza. I metodi di modifica o metodi *set*, assegnano un valore a una variabile di istanza.
- Ciascun metodo appartiene a una classe ed è utilizzabile dagli oggetti di quella classe.
- La parola chiave `this`, quando usata all'interno della definizione di un metodo, rappresenta l'oggetto che riceve l'invocazione di metodo.
- I metodi possono avere parametri di tipo primitivo e/o parametri di tipo classe, ma i due tipi di parametri si comportano in maniera differente. Un parametro di tipo primitivo è inizializzato al valore primitivo dell'argomento corrispondente. Un parametro di tipo classe è inizializzato con l'indirizzo di memoria, detto anche riferimento (*reference*), dell'argomento.
- Ogni cambiamento operato su un parametro di tipo primitivo non viene effettuato sull'argomento corrispondente.
- A seguito di un'invocazione di metodo, un argomento di tipo classe viene riferito (o referenziato) anche da un'altra variabile, il parametro formale. Perciò, qualsiasi cambiamento effettuato allo stato del parametro formale si riflette sullo stato dell'argomento corrispondente. Tuttavia, se il parametro è sostituito da un altro oggetto istanziato all'interno del metodo stesso, questa modifica non si applica all'argomento iniziale.
- Il termine incapsulamento indica che i dati e le azioni sono combinati in un unico oggetto istanza di una classe e che i dettagli dell'implementazione sono nascosti. Rendere tutte le variabili di istanza private è parte del processo di incapsulamento.
- Una precondizione di un metodo commenta le condizioni riguardo il suo stato che devono valere prima che il metodo sia invocato. Le postcondizioni di un metodo indicano le condizioni che valgono dopo la sua invocazione. Le postcondizioni descrivono cioè gli effetti prodotti da un'invocazione del metodo qualora valgano le precondizioni. Precondizioni e postcondizioni sono asserzioni.
- Il programma `javadoc` crea documentazione a partire dai commenti inseriti in una classe.
- I progettisti delle classi usano la notazione UML per rappresentarle.
- Il test di unità (*unit testing*) è una metodologia tale per cui un programmatore scrive una suite di test per verificare se le unità di codice funzionano correttamente.
- Gli operatori == e =, quando usati con oggetti, non si comportano allo stesso modo di quando sono usati con i tipi primitivi.
- È buona norma definire un metodo `equals` per ogni classe implementata.

8.5 Esercizi

1. Si definisca una classe per rappresentare una carta di credito. Si pensi agli attributi della carta di credito, cioè ai dati tipici della carta. Si pensi inoltre al suo funzionamento. Si definisca quindi un diagramma delle classi UML della carta di credito e si diano tre esempi d'istanza della classe.
2. Si ripeta l'Esercizio 1 per l'estratto conto di una carta di credito invece che per la carta stessa. Un estratto conto rappresenta i pagamenti e i versamenti effettuati usando la carta di credito.
3. Si ripeta l'Esercizio 1 considerando una moneta invece della carta di credito.
4. Si ripeta l'Esercizio 1 considerando un insieme di monete invece della carta di credito.
5. Si consideri una classe Java da usare per ricevere dall'utente un intero valido. Un oggetto di questa classe deve avere i seguenti attributi:
   - valore minimo accettato;
   - valore massimo accettato;
   - stringa di sollecito.

   Inoltre deve avere il seguente metodo:
   - `getValore` – mostra la stringa di sollecito e legge un valore usando la classe `Scanner`. Se il valore letto non è compreso tra il minimo e il massimo, ripete queste azioni finché non viene inserito un valore accettabile. Il metodo restituisce il valore letto.

   a. Si scrivano le precondizioni e postcondizioni del metodo `getValore`.

   b. Si implementi la classe in Java.

   c. Si scrivano le istruzioni Java necessarie per collaudare la classe.
6. Si consideri una classe che registra le vendite di un articolo. Un oggetto di questa classe avrà i seguenti attributi:
   - numero venduti;
   - totale vendite;
   - totale scontati;
   - costo per articolo;
   - quantità all'ingrosso;
   - sconto percentuale all'ingrosso.

   Inoltre avrà i seguenti metodi:
   - `registraVendita(n)` – registra la vendita di n articoli. Se n supera la quantità all'ingrosso, il costo per ogni articolo deve essere ridotto della percentuale di sconto all'ingrosso;
   - `mostraVendite` – mostra il numero di articoli venduti, il totale delle vendite e lo sconto totale.

a. Si implementi la classe in Java.

b. Si scrivano le istruzioni Java necessarie per collaudare la classe.

7. Si consideri la classe `BarcaAMotore` che rappresenta una barca a motore. Una barca a motore presenta i seguenti attributi:
   - la capacità del serbatoio;
   - la quantità di carburante nel serbatoio;
   - la velocità massima della barca;
   - la velocità corrente della barca;
   - l'efficienza del motore della barca;
   - la distanza percorsa.

   La classe ha i metodi per le seguenti attività:
   - cambiare la velocità della barca;
   - far navigare la barca per un certo tempo alla velocità corrente;
   - riempire il serbatoio con una certa quantità di carburante;
   - restituire l'ammontare di carburante nel serbatoio;
   - restituire la distanza percorsa.

   Se la barca ha un'efficienza e, l'ammontare di carburante usato quando si naviga a una velocità s per un tempo t è $e \times s^2 \times t$. La distanza percorsa è $s \times t$.

   a. Si scriva l'intestazione di ciascun metodo.

   b. Si scrivano le precondizioni e postcondizioni di ciascun metodo.

   c. Si scrivano le istruzioni Java per collaudare la classe.

   d. Si implementi la classe.

8. Si consideri una classe `IndirizzoPersona` che rappresenta un elemento di una rubrica. I suoi attributi sono:
   - nome della persona;
   - cognome della persona;
   - indirizzo e-mail della persona;
   - numero di telefono della persona.

   La classe avrà i metodi per:
   - accedere a ogni attributo;
   - cambiare l'indirizzo e-mail;
   - cambiare il numero di telefono;
   - verificare se due istanze sono uguali sulla base del nome.

   a. Si scriva l'intestazione di ogni metodo.

   b. Si scrivano le precondizioni e le postcondizioni di ogni metodo.

c. Si scrivano le istruzioni Java per collaudare la classe.

d. Si implementi la classe.

9. Si consideri la classe `Punteggio` che rappresenta un punteggio numerico per la valutazione di qualcosa, come, per esempio, un film. I suoi attributi sono:

   - descrizione dell'oggetto valutato;
   - punteggio massimo possibile;
   - punteggio.

   La classe avrà i metodi per:

   - ottenere un punteggio da un utente;
   - restituire il massimo punteggio possibile;
   - restituire il punteggio;
   - restituire una stringa che mostra il punteggio in un formato utile per essere stampato.

   a. Si scriva l'intestazione di ogni metodo.

   b. Si scrivano le precondizioni e postcondizioni di ogni metodo.

   c. Si scrivano le istruzioni Java per collaudare la classe.

   d. Si implementi la classe.

10. Si consideri la classe `ProgettoScienzaPunteggio` per giudicare un progetto scientifico. Questa classe userà la classe `Punteggio` descritta in precedenza. Gli attributi della nuova classe sono:

    - nome del progetto;
    - stringa identificativa univoca del progetto;
    - nome della persona;
    - un punteggio per l'abilità creatività (max 30);
    - un punteggio per il valore scientifico del progetto (max 30);
    - un punteggio per la completezza (max 15);
    - un punteggio per l'abilità tecnica (max 15);
    - un punteggio per la chiarezza (max 10).

    La classe avrà i metodi per:

    - ricevere il numero di giudici;
    - ricevere tutti punteggi per un particolare progetto;
    - restituire il totale dei punteggi per un particolare progetto;
    - restituire il massimo punteggio possibile;
    - restituire una stringa che mostra il punteggio di un progetto in un formato utile per la visualizzazione.

    a. Si scriva l'intestazione di ogni metodo.

    b. Si scrivano le precondizioni e postcondizioni di tutti i metodi.

c. Si scrivano le istruzioni Java per collaudare la classe.

d. Si implementi la classe.

8.6 Progetti

1. Si scriva un programma che risponda a domande come la seguente. Si supponga che la specie Bufalo Klingon abbia una popolazione di 100 individui e un tasso di crescita del 15% e che la specie Elefante abbia una popolazione di 10 individui e un tasso di crescita del 35%. In quanti anni la popolazione di elefanti supererà quella dei Bufali Klingon?

   Si usi la classe `Specie` definita nel Listato 8.16. Il programma richiederà i dati di entrambe le specie e risponderà dicendo quanti anni ci vorranno per far sì che la specie con il minor numero di individui superi quella con il numero maggiore di individui. Le due specie possono essere inserite in qualsiasi ordine. È possibile che la specie con la popolazione minore non superi mai quella con la popolazione maggiore. In questo caso il programma dovrà mostrare un messaggio adeguato a sottolineare questo fatto.

2. Si definisca una classe `Contatore`. Un oggetto di questa classe viene usato per contare delle cose, quindi registra il conteggio, che è un numero intero positivo. Si includano i metodi per assegnare il valore 0 al contatore, per incrementare il contatore di 1 e per decrementarlo di 1. Ci si assicuri che nessun metodo permetta al contatore di diventare negativo. Si includa, inoltre, un metodo *get* che restituisca il valore corrente del conteggio e un metodo che mostri il conteggio a schermo. Non si definisca un metodo di input. L'unico metodo che può assegnare valori al contatore è quello che pone il suo valore a 0. Si scriva un programma per collaudare questa classe. *Suggerimento*: è necessaria una sola variabile di istanza.

3. Si scriva un programma di valutazione per un insegnante il cui corso ha le seguenti caratteristiche.

   - Vengono dati due esercizi ciascuno con un punteggio di 10.
   - Ci sono due esami intermedi e uno finale ciascuno con un punteggio massimo di 100.
   - L'esame finale vale il 50% del punteggio totale, mentre gli esami intermedi il 25%. I due esercizi, assieme, valgono il 25%. È bene non dimenticarsi di normalizzare i punteggi: questi dovrebbero essere convertiti in percentuali prima di essere usati per calcolare la media finale.

   Ciascun punteggio che supera il 90% del punteggio totale si trasforma in un voto A, un punteggio tra 80 e 89% in B, tra 70 e 79 in C, tra 60 e 69 in C, ogni punteggio sotto 60 è una F.

   Il programma dovrebbe leggere i punteggi di ogni studente e mostrarli facendo vedere i punteggi dei due esercizi, dei due esami intermedi, dell'esame finale, la media dell'intero corso e il voto espresso in lettera. Il voto finale è compreso tra 0 e 100 e rappresenta la media pesata del lavoro dello studente.

Si definisca e si usi una classe per registrare queste informazioni. La classe dovrebbe avere variabili di istanza per i voti degli esercizi, degli esami intermedi, dell'esame finale, per il punteggio totale del corso e per il voto espresso in lettera. La classe dovrebbe avere metodi di input e output. Il metodo di input non dovrebbe chiedere il voto finale, né in percentuale né in lettera. La classe dovrebbe avere i metodi per calcolare la media totale e il voto finale in lettera. Questi due metodi saranno metodi `void` che assegnano i valori calcolati alle variabili di istanza appropriate. Si ricordi che un metodo può invocare un altro metodo. Se si preferisce, si può definire un singolo metodo che assegna sia il punteggio totale, sia il punteggio in lettere, ma se questa è la scelta, è bene usare un metodo ausiliario. Il programma dovrebbe usare tutti i metodi descritti. La classe dovrebbe avere un insieme ragionevole di metodi *set* e *get*, anche se il programma non li usa tutti. Si possono aggiungere altri metodi se si desidera.

4. Si crei una classe `Persona` che presenti gli attributi nome ed età e i metodi per eseguire le seguenti attività.
   - Assegnare un nome a un oggetto `Persona`.
   - Assegnare un valore all'attributo età di un oggetto `Persona`.
   - Verificare se due oggetti `Persona` sono uguali (hanno stesso nome e età).
   - Verificare se due oggetti `Persona` hanno lo stesso nome.
   - Verificare se due oggetti `Persona` hanno la stessa età.
   - Verificare se una persona è più vecchia di un'altra.
   - Verificare se una persona è più giovane di un'altra.

   Si scriva un programma di test che mostri l'esecuzione di ciascun metodo.

5. Si crei una classe che rappresenta la distribuzione di voti per un dato corso. Si scrivano i metodi per effettuare le seguenti attività.
   - Assegnare il numero di studenti che hanno preso un certo voto (per ognuno dei voti A, B, C, D, F).
   - Leggere il numero di studenti che hanno preso un certo voto (per ognuno dei voti A, B, C, D, F).
   - Restituire il numero totale di voti.
   - Restituire la percentuale di voti per ciascuna lettera come un intero compreso tra 0 e 100.
   - Disegnare un grafico a barre che visualizzi la distribuzione dei voti.

   Il grafico avrà cinque barre. Ciascuna barra può essere una riga orizzontale di asterischi: il numero di asterischi di una riga sarà proporzionale alla percentuale per ciascuna categoria. Se si fa in modo che un asterisco rappresenti il 2%, allora 50 asterischi rappresenteranno il 100%. Si gradui l'asse orizzontale al 10 per cento da 0 a 100 e si etichetti ciascuna riga con il nome della lettera corrispondente.

   Per esempio, se i punteggi sono 1 A, 4 B, 6 C, 2 D, 1 F, il numero totale di punteggi è 14, la percentuale di A è 7, di B è 29 di C è 43, di D è 14 e di F è 7. La riga A conterrà 4 asterischi (7% di 50 arrotondato per eccesso), la riga B 14, la C 21, la D 7 e la riga F ne conterrà 4.

Il grafico sarà simile al seguente:

```
0    10   20   30   40   50   60   70   80   90   100%
|    |    |    |    |    |    |    |    |    |    |
**************************************************
**** A
************** B
********************** C
******* D
**** F
```

6. Si scriva un programma che usa la classe `Acquisto` del Listato 8.10 per assegnare i seguenti prezzi.

   Arance: 10 per 2.29 Euro.

   Uova: 12 per 1.69 Euro.

   Mele: 3 per 1.00 Euro.

   Meloni: 4.39 Euro l'uno.

   Panini: 6 per 3.50 Euro.

   Si calcoli quindi il costo di ciascuno dei seguenti articoli e il totale:

   2 dozzine di arance;

   3 dozzine di uova;

   20 mele;

   2 meloni

   1 dozzina di panini.

7. Si scriva un programma che risponda a domande come la seguente. Si supponga che la specie Bufalo Klingon abbia una popolazione di 100 individui, un tasso di crescita del 15% e che viva in un'area di 1500 km^2. Quanto ci metterà la popolazione a superare una densità di 1 individuo per km^2? Si usi la classe `Specie` del Listato 8.16 con l'aggiunta del metodo `getDensita` che restituisce la densità della popolazione calcolata sulla base del numero di individui e dell'estensione del territorio.

8. Si consideri una classe che può essere usata per un gioco in cui si deve indovinare una parola, indicando le diverse lettere in essa contenute. La classe deve avere i seguenti attributi:

   - la parola da indovinare;
   - le lettere scoperte, in cui ciascuna lettera non ancora scoperta è sostituita da un punto interrogativo. Per esempio, se la parola segreta è *abracadabra* e le lettere *a*, *b* ed *e* sono state indovinate dai giocatori, la parola scoperta sarà *ab?a?a?ab?a*;
   - il numero di tentativi fatti;
   - il numero di tentativi non corretti.

   La classe avrà i seguenti metodi:

   - `indovina(c)` – prova a indovinare se la lettera `c` è parte della parola;

- `getParolaScoperta` – restituisce una stringa che contiene le lettere indovinate nelle loro corrette posizioni e le lettere non ancora scoperte sostituite con un punto interrogativo;
- `getParolaDaIndovinare` – restituisce la parola da indovinare;
- `getNumeroTentativi` – restituisce il numero di tentativi;
- `isIndovinata` – restituisce vero se la parola è stata indovinata.

a. Si scriva l'intestazione per ciascun metodo.
b. Si scrivano le precondizioni e postcondizioni di ciascun metodo.
c. Si scrivano le istruzioni Java per collaudare la classe.
d. Si implementi la classe.
e. Si elenchino gli attributi aggiuntivi non indicati nel testo, ma necessari per l'implementazione.
f. Si scriva un programma che usi la classe definita per implementare il gioco completo.

9. Si consideri una classe `PartitaBasket` che rappresenta lo stato di una partita di basket. I suoi attributi sono:
   - nome della prima squadra;
   - nome della seconda squadra;
   - punteggio della prima squadra;
   - punteggio della seconda squadra;
   - stato del gioco (finito o ancora in corso).

   Deve avere metodi per le seguenti attività:
   - registrare la realizzazione di un canestro da 1 punto fatto da una squadra;
   - registrare la realizzazione di un canestro da 2 punti fatto da una squadra;
   - registrare la realizzazione di un canestro da 3 punti fatto da una squadra;
   - cambiare lo stato del gioco da *ancora in corso* a *finito*;
   - restituire il punteggio di una squadra;
   - restituire il nome della squadra vincitrice.

   a. Si scriva l'intestazione di ciascun metodo.
   b. Si scrivano le precondizioni e postcondizioni di ciascun metodo.
   c. Si scrivano le istruzioni Java per collaudare la classe.
   d. Si implementi la classe.
   e. Si elenchino gli attributi aggiuntivi non indicati nel testo, ma necessari per l'implementazione.
   f. Si scriva un programma che usi la classe definita per tracciare il punteggio di una partita di basket. Si usi un ciclo che legge in input un valore ogni volta che viene segnato un canestro. Per far questo è necessario indicare il nome della squadra e

il numero di punti realizzati: 1,2 o 3. Dopo che viene letto l'input, si mostri il punteggio corrente. Per esempio, una porzione dell'interazione con il programma potrebbe essere la seguente:

```
Inserisci un punteggio:
a 1
Gatti 1, Cani 0. I gatti stanno vincendo.
Inserisci un punteggio:
a 2
Gatti 3, Cani 0. I gatti stanno vincendo.
Inserisci un punteggio:
b 2
Gatti 3, Cani 2. I gatti stanno vincendo.
Inserisci un punteggio:
b 3
Gatti 3, Cani 5. I cani stanno vincendo.
```

10. Si consideri una classe `PromotoreConcerto` che registra i biglietti venduti per un concerto. Prima del giorno del concerto i biglietti vengono venduti solo al telefono. Le vendite il giorno del concerto sono fatte solo di persona, sul posto.

    La classe ha i seguenti attributi:

    - nome del gruppo;
    - capacità del luogo in cui si svolge il concerto;
    - numero biglietti venduti;
    - prezzo di un biglietto venduto al telefono;
    - prezzo di un biglietto venduto sul posto;
    - ricavato totale dalla vendita.

    Ha metodi per le seguenti attività:

    - registrare la vendita di uno o più biglietti;
    - cambiare lo stato da vendita al telefono a vendita sul posto;
    - restituire il numero di biglietti venduti;
    - restituire il numero di biglietti rimanenti;
    - restituire il totale delle vendite per il concerto.

    a. Si scrivano le intestazioni di tutti i metodi.
    b. Si scrivano le precondizioni e postcondizioni dei metodi.
    c. Si scrivano le istruzioni Java per collaudare la classe.
    d. Si implementi la classe.
    e. Si elenchino gli attributi aggiuntivi non indicati nel testo, ma necessari per l'implementazione.

f. Si scriva un programma che usa la classe scritta per registrare le vendite per un concerto. Il programma deve registrare le vendite effettuate al telefono e quelle sul posto. A mano a mano che vengono venduti i biglietti, devono essere mostrati i posti disponibili. Al termine il programma deve mostrare il numero di biglietti venduti e il ricavato totale dalle vendite.

11. Si riscriva la classe `Cane` del Listato 8.1 utilizzando le informazioni e i principi dell'incapsulamento descritti nella Sezione 8.2. La nuova versione dovrebbe includere metodi *set* e *get*. Si definisca anche un metodo `equals` che restituisca `true` se il nome, l'età e la razza del cane coincidono con quelli dell'oggetto con il quale si esegue il confronto. Si includa un metodo `main` per verificare le funzionalità della nuova classe `Cane`.

12. Si consideri una classe `Film` che contenga informazioni relative a un film. La classe ha i seguenti attributi:

- il titolo del film;
- la classificazione MPAA (*Motion Picture Association of America*) (per esempio G, PG, PG-13, R);
- il numero di persone che hanno assegnato al film la valutazione 1 (Terribile);
- il numero di persone che hanno assegnato al film la valutazione 2 (Brutto);
- il numero di persone che hanno assegnato al film la valutazione 3 (Normale);
- il numero di persone che hanno assegnato al film la valutazione 4 (Bello);
- il numero di persone che hanno assegnato al film la valutazione 5 (Grandioso).

Si implementi la classe con i metodi *set* e *get* per il titolo del film e la sua classificazione MPAA. Si scriva un metodo `aggiungiValutazione` che richiede in input un parametro di tipo intero. Il metodo deve verificare che il parametro sia un numero tra 1 e 5 e, in tal caso, incrementare di un'unità il numero di persone che hanno espresso la valutazione corrispondente. Per esempio, se il valore del parametro è 3, il numero di persone che hanno assegnato al film la valutazione 3 deve essere incrementato di uno. Si scriva poi un altro metodo, `getMedia`, che restituisca la media delle valutazioni.

Si verifichi il funzionamento della classe scrivendo un metodo `main` che crei almeno due oggetti di tipo `Film`, aggiunga a ognuno dei due almeno cinque valutazioni e stampi il titolo, la classificazione MPAA e la media delle valutazioni per ognuno dei due film.

Capitolo 9

Approfondimenti su classi, oggetti e metodi

OBIETTIVI

- Definire e utilizzare i costruttori di una classe.
- Scrivere e utilizzare variabili e metodi statici.
- Utilizzare le classi *wrapper* predefinite.
- Utilizzare l'*overloading*.
- Utilizzare gli array con classi e oggetti.
- Definire e utilizzare metodi che fanno uso di enumerazioni.
- Definire e utilizzare package e l'istruzione `import`.

Questo capitolo prosegue la presentazione di classi e metodi, introducendo alcuni concetti più avanzati. In particolare, saranno introdotti i costruttori ovvero i metodi usati per creare un nuovo oggetto. In realtà, i costruttori sono già stati utilizzati nel capitolo precedente, quando si è adoperato l'operatore new. I costruttori usati in precedenza sono stati definiti da Java. In questo capitolo verrà mostrato come scrivere costruttori personalizzati.

Il capitolo presenta, inoltre, alcuni approfondimenti sui metodi statici, già introdotti nel Capitolo 5.

Il capitolo presenta anche l'*overloading*, una caratteristica dei linguaggi di programmazione a oggetti (e quindi anche di Java), che consente di assegnare a due o più metodi lo stesso nome all'interno di una medesima classe. Verrà poi discusso l'utilizzo di classi e oggetti con gli array, sia utilizzando array come variabili di istanza di una classe sia utilizzando classi come tipi base degli array.

Il capitolo presenterà, inoltre, i package, che sono librerie di classi che possono essere utilizzate nella definizione di altre classi. Si è già fatto uso dei package quando sono state utilizzate le classi della Java Class Library.

Prerequisiti

Prima di leggere questo capitolo, occorre aver assimilato gli argomenti affrontati nel Capitolo 8, mentre alcuni paragrafi possono essere letti in un secondo momento. Il Paragrafo 9.4, per esempio, tratta alcuni punti delicati riguardanti l'utilizzo delle variabili di istanza di una classe. Questa parte del capitolo non è necessaria per comprendere gli argomenti trattati in questo testo e quindi la sua lettura può essere rimandata a un secondo tempo. In ogni caso, il Paragrafo 9.4 si occupa di questioni fondamentali che dovrebbero essere lette per approfondire le caratteristiche del linguaggio.

Il materiale riguardante la scrittura dei package nel Paragrafo 9.8 richiede la conoscenza delle cartelle (directory) e delle variabili di percorso (*path variable*). Entrambi non sono argomenti centrali di Java, ma sono argomenti specifici dei sistemi operativi, in quanto i loro dettagli dipendono dallo specifico sistema operativo che si utilizza. Il Paragrafo 9.8 non è necessario alla comprensione di questo testo, perciò è possibile riprenderlo in futuro, dopo aver appreso queste nozioni.

9.1 Costruttori

Quando si crea un oggetto di una classe utilizzando l'operatore new, si invoca un particolare tipo di metodo chiamato *costruttore*. In quel momento, spesso è opportuno compiere operazioni di inizializzazione, come assegnare specifici valori alle variabili di istanza. Un costruttore eseguirà queste inizializzazioni. Questo paragrafo spiega come definire e utilizzare i costruttori.

9.1.1 Definire i costruttori

Un **costruttore** è un particolare metodo che viene invocato quando si utilizza l'operatore new per creare un nuovo oggetto. I costruttori utilizzati finora sono stati definiti da Java. Per esempio, considerando la classe `Specie` nel Listato 8.16 del capitolo precedente, è possibile creare un nuovo oggetto `Specie` scrivendo:

```
Specie specieTerrestre = new Specie();
```

La prima parte di questa istruzione, `Specie specieTerrestre`, dichiara che la variabile `specieTerrestre` è un riferimento (*reference*) a un oggetto della classe `Specie`. La seconda parte, `new Specie()`, crea e inizializza un nuovo oggetto il cui indirizzo viene quindi assegnato a `specieTerrestre`. In questa istruzione, `Specie()` è una chiamata al costruttore che Java ha fornito automaticamente alla classe. Le parentesi sono vuote perché questo particolare costruttore non richiede alcun argomento.

Nelle classi viste finora, i costruttori creano gli oggetti e forniscono alle loro variabili di istanza un valore iniziale di default. Questi valori potrebbero, però, non essere i valori desiderati. Per esempio, si potrebbe volere che alcune (o tutte) variabili di istanza vengano inizializzate con specifici valori nel momento in cui l'oggetto viene creato. La definizione di un costruttore permette proprio di fare questa operazione.

Un costruttore può eseguire qualsiasi azione inserita nella sua definizione, anche se un costruttore è preposto essenzialmente a eseguire azioni di inizializzazione, per esempio del valore delle variabili di istanza. I costruttori hanno essenzialmente lo stesso compito

dei metodi *set*. Ma, a differenza dei metodi *set*, i costruttori creano un oggetto oltre a inizializzarlo. Come i metodi *set*, i costruttori possono avere parametri.

Si consideri, per esempio, una classe che rappresenta un generico animale domestico. Si supponga di descrivere ogni animale mediante il suo nome, la sua età e il suo peso. È possibile aggiungere dei semplici comportamenti all'oggetto animale per impostare od ottenere i valori dei suoi tre attributi. Si immagini ora di disegnare un diagramma delle classi come quello rappresentato nella Figura 9.1 per descrivere la classe `Animale`. Si notino i quattro metodi che impostano il valore delle variabili di istanza. Uno di loro imposta il valore di tutte e tre le variabili di istanza `nome`, `eta` e `peso`. Gli altri tre metodi impostano, ciascuno, il valore di una sola variabile di istanza. Questo diagramma delle classi non include i costruttori, come tipicamente accade.

Una proprietà dei costruttori, che in un primo momento potrebbe sembrare strana, consiste nel fatto che ogni costruttore ha lo stesso nome della sua classe. Pertanto, se una classe si chiama `Specie`, anche i suoi costruttori si chiameranno `Specie`. Se una classe si chiama `Animale`, i suoi costruttori si chiameranno `Animale`.

I costruttori hanno spesso più definizioni, ognuna delle quali presenta un differente numero di parametri o differenti tipi di parametri e, a volte, assomigliano ai metodi *set* della classe. Per esempio, il Listato 9.1 contiene una definizione della classe `Animale` che include diversi costruttori. Si noti che le intestazioni dei suoi costruttori *non* contengono la parola `void`. Quando si definisce un costruttore, non si specifica alcun tipo di ritorno e tanto meno `void`. I costruttori della classe `Animale` somigliano molto ai metodi *set*, che *sono* metodi `void`. Al solo scopo di enfatizzare le analogie e le differenze, ogni costruttore è stato raggruppato con il suo analogo metodo *set*. In ogni caso, diversamente da alcuni metodi *set*, i costruttori forniscono un valore a tutte le variabili di istanza, anche se non hanno un parametro per ognuna di esse. Se il costruttore non inizializza una particolare variabile di istanza, lo farà Java, assegnandole un valore di default. In ogni modo, quando si definisce un costruttore, è una normale pratica in programmazione assegnare esplicitamente un valore a tutte le variabili di istanza.

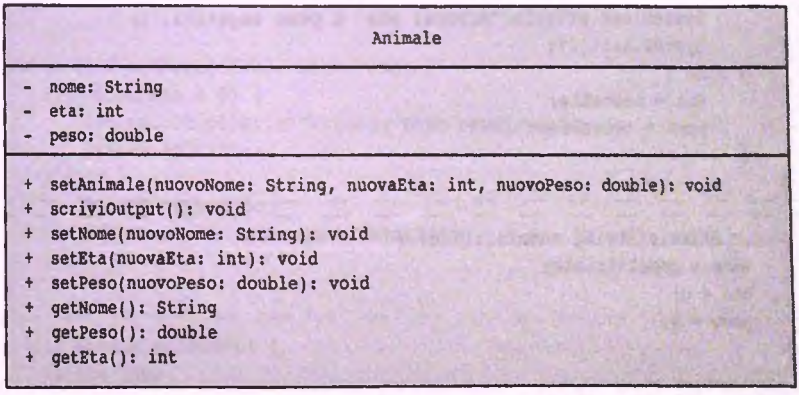


Figura 9.1 Diagramma delle classi per la classe `Animale`.

MyLab

LISTATO 9.1 La classe `Animale`: un esempio sui costruttori e sui metodi *set*.

```
/**
  Classe che descrive un animale
*/
public class Animale {

    private String nome;
    private int eta;      //in anni
    private double peso; //in Kg

    public Animale() {
        nome = "Nessun nome";
        eta = 0;
        peso = 0;
    }

    public Animale(String nomeIniziale, int etaIniziale, double pesoIniziale) {
        nome = nomeIniziale;
        if ((etaIniziale < 0) || (pesoIniziale < 0)) {
            System.out.println("Errore: eta' o peso negativi.");
            System.exit(0);
        } else {
            eta = etaIniziale;
            peso = pesoIniziale;
        }
    }

    public void setAnimale(String nuovoNome, int nuovaEta, double nuovoPeso) {
        nome = nuovoNome;
        if ((nuovaEta < 0) || (nuovoPeso < 0)) {
            System.out.println("Errore: eta' o peso negativi.");
            System.exit(0);
        } else {
            eta = nuovaEta;
            peso = nuovoPeso;
        }
    }

    public Animale(String nomeIniziale) {
        nome = nomeIniziale;
        eta = 0;
        peso = 0;
    }

    public void setNome(String nuovoNome) {
        nome = nuovoNome; //eta' e peso rimangono invariate
    }
```

Costruttore di default.

```
public Animale(int etaIniziale) {
    nome = "Nessun nome";
    peso = 0;
    if (etaIniziale < 0) {
        System.out.println("Errore: eta' negativa.");
        System.exit(0);
    } else {
        eta = etaIniziale;
    }
}

public void setEta(int nuovaEta) {
    if (nuovaEta < 0) {
        System.out.println("Errore: eta' negativa.");
        System.exit(0);
    } else {
        eta = nuovaEta;
        //nome e peso rimangono invariate
    }
}
```

```
public Animale(double pesoIniziale) {
    nome = "Nessun nome";
    eta = 0;
    if (pesoIniziale < 0) {
        System.out.println("Errore: peso negativo.");
        System.exit(0);
    } else {
        peso = pesoIniziale;
    }
}

public void setPeso(double nuovoPeso) {
    if (nuovoPeso < 0) {
        System.out.println("Errore: peso negativo.");
        System.exit(0);
    } else {
        peso = nuovoPeso;
        //nome e eta' rimangono invariate
    }
}
```

```
public String getNome() {
    return nome;
}

public int getEta() {
    return eta;
}
```

```
    public double getPeso() {
        return peso;
    }

    public void scriviOutput(){
        System.out.println("Nome: " + nome);
        System.out.println("Eta: " + eta + " anni");
        System.out.println("Peso: " + peso + " Kg");
    }
}
```

Costruttori e metodi `set` sono correlati, ma usati in modi differenti

I costruttori vengono invocati esclusivamente quando viene creato un oggetto. I metodi *set* vengono utilizzati per cambiare lo stato di un oggetto esistente.

Il Listato 9.1 include un costruttore chiamato `Animale` senza parametri. Questo costruttore è chiamato **costruttore di default**. Tipicamente, quando si vuole definire un solo costruttore, è opportuno definire anche il costruttore senza parametri. La definizione della classe `Specie` introdotta nel capitolo precedente non contiene alcuna definizione di costruttore. Qualora una definizione di classe non contenga alcun costruttore, Java definisce automaticamente il costruttore di default, che è senza parametri. Questo costruttore, definito in maniera automatica, crea un oggetto e inizializza le variabili di istanza con i valori di default dei loro tipi. Tuttavia, se in una classe viene definito almeno un costruttore, non viene più aggiunto automaticamente nessun altro costruttore. Questo anche nel caso in cui si sia definito un costruttore con parametri. Quindi, per la classe `Animale` presentata nel Listato 9.1, dal momento che sono stati definiti dei costruttori, si è stati attenti a includere anche un costruttore senza parametri, che è il costruttore di default.

Quando si crea un oggetto utilizzando l'operatore `new` occorre sempre includere l'invocazione a uno dei costruttori definiti nella classe. Come per ogni invocazione di metodo, bisogna indicare la lista degli argomenti tra parentesi dopo il nome del costruttore; il nome, si ricorda, è lo stesso di quello della classe. Per esempio, si supponga di voler usare `new` per creare un nuovo oggetto della classe `Animale`. Si potrebbe procedere come segue:

```
Animale pesce = new Animale("Bavosetta", 2, 0.11);
```

La parte `Animale("Bavosetta", 2, 0.11)` è un'invocazione al costruttore di `Animale` che accetta tre argomenti: uno di tipo `String`, uno di tipo `int` e l'ultimo di tipo `double`. Questa istruzione crea un nuovo oggetto che rappresenta un animale chiamato Bavosetta che ha 2 anni e pesa 0,11 kg.

Un'invocazione a un costruttore restituisce un riferimento (*reference*), cioè l'indirizzo di memoria di un oggetto. L'esempio precedente assegna questo riferimento alla variabile `pesce`. La Figura 9.2 illustra il funzionamento.

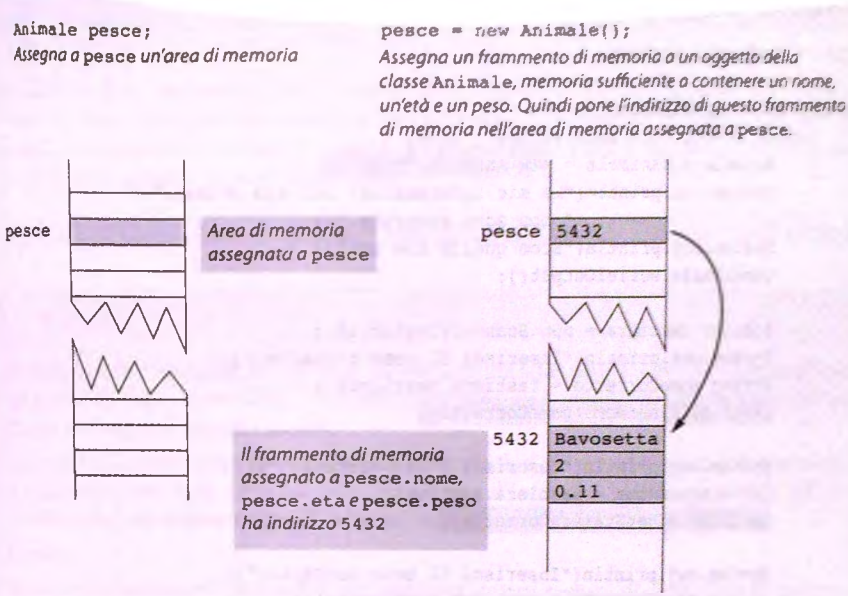


Figura 9.2 Un costruttore che restituisce un riferimento.

Si consideri un altro esempio. L'istruzione:

```
Animale mioAnimale = new Animale();
```

crea un nuovo oggetto della classe `Animale` invocando il costruttore di default, cioè il costruttore senza parametri. Se si osserva con attenzione la definizione della classe `Animale` nel Listato 9.1, si nota che il costruttore di default assegna all'oggetto il nome `Nessun nome` e imposta a 0 il valore delle variabili di istanza `eta` e `peso` (naturalmente, un animale appena nato non pesa 0; il valore zero è solo fittizio finché non potrà essere determinato il vero peso).

Non è possibile usare un oggetto già esistente per invocare un costruttore. Perciò, l'invocazione seguente sull'oggetto `mioAnimale` della classe `Animale` non è valida:

```
mioAnimale.Animale("Pippo", 1, 2.6);  // Non valido!
```

Una volta creato un oggetto, l'unico modo per modificare i valori contenuti nelle sue variabili di istanza consiste nell'invocare i suoi metodi *set*. Quindi, data la classe definita nel Listato 9.1, anziché utilizzare la precedente invocazione errata del costruttore `Animale`, si può invocare `setAnimale` come mostrato di seguito:

```
mioAnimale.setAnimale("Pippo", 1, 2.6);
```

Il Listato 9.2 presenta un semplice programma che mostra l'uso di un costruttore e di alcuni metodi *set*.

MyLab

LISTATO 9.2 Usare un costruttore e i metodi *set*.

```
import java.util.Scanner;

public class AnimaleDemo {
    public static void main(String[] args) {
        Animale tuoAnimale = new Animale("Fido");
        System.out.println("Le mie informazioni sul tuo animale" +
                           " non sono accurate.");
        System.out.println("Ecco quello che so:");
        tuoAnimale.scriviOutput();

        Scanner tastiera = new Scanner(System.in);
        System.out.println("Inserisci il nome corretto:");
        String nomeCorretto = tastiera.nextLine();
        tuoAnimale.setNome(nomeCorretto);

        System.out.println("Inserisci l'eta corretta:");
        int etaCorretta = tastiera.nextInt();
        tuoAnimale.setEta(etaCorretta);

        System.out.println("Inserisci il peso corretto:");
        double pesoCorretto = tastiera.nextDouble();
        tuoAnimale.setPeso(pesoCorretto);

        System.out.println("Dati aggiornati:");
        tuoAnimale.scriviOutput();
    }
}
```

Esempio di output

```
Le mie informazioni sul tuo animale non sono accurate.
Ecco quello che so:
Nome: Fido
Eta: 0 anni
Peso: 0.0 Kg
Inserisci il nome corretto:
Speedy
Inserisci l'eta corretta:
1
Inserisci il peso corretto:
4.5
Dati aggiornati:
Nome: Speedy
Eta: 1 anni
Peso: 4.5 Kg
```

Raggruppare le definizioni dei costruttori

Nel Listato 9.1, ogni costruttore è stato raggruppato con il suo analogo metodo *set*. Questo è stato fatto per enfatizzare le somiglianze e le differenze tra questi due tipi di metodi. Generalmente, infatti, i costruttori sono definiti uno di seguito all'altro e posti prima di tutti gli altri metodi della classe.

Costruttori

Un costruttore è un metodo che viene invocato quando viene creato un oggetto di una classe usando l'operatore new. Un costruttore fornisce il valore iniziale delle variabili di istanza dell'oggetto appena creato.

I costruttori devono avere lo stesso nome della classe a cui appartengono. Un costruttore è definito come ogni altro metodo, tranne per il fatto che non ha specificato nell'intestazione del metodo un tipo di ritorno, nemmeno `void`.

Esempi

Si faccia riferimento al Listato 9.1 che contiene diversi esempi di definizioni di costruttori. Di seguito sono presentati alcuni esempi di invocazione:

```
Animale mioCane = new Animale("Fido", 2, 4.5);
Animale tuoCane = new Animale("Fuffy");
Animale nostroCane = new Animale();
```

Costruttore di default

Un costruttore senza parametri è chiamato costruttore di default. Quando si definisce una classe, si dovrebbe definire anche un costruttore di default.

Java può definire un costruttore di default

Se in una classe non viene definito alcun costruttore, Java ne definisce comunque uno di default, che assegna alle variabili di istanza il valore di default per i loro tipi. Nel momento in cui viene definito un costruttore in una classe, Java non ne definirà altri in maniera automatica. In questo caso, se non viene definito esplicitamente il costruttore di default, la classe non lo avrà.

Omettere il costruttore di default

Si immagini di aver omesso la definizione del costruttore di default nella classe `Animale` nel Listato 9.1. L'istruzione:

```
Animale gatto = new Animale();
```

non sarebbe valida e produrrebbe un messaggio d'errore. Poiché, se non viene definito alcun costruttore, Java fornisce automaticamente un costruttore di default, si potrebbe pensare che questa istruzione sia valida. Ma dato che la definizione di questa classe contiene la definizione di almeno un costruttore, Java non fornisce alcun costruttore.

Le classi vengono spesso riusate più e più volte e alla fine si potrebbe avere l'esigenza di creare un nuovo oggetto utilizzando il costruttore senza argomenti. Bisogna quindi considerare attentamente la decisione di omettere un costruttore di default dalla definizione della classe che si sta scrivendo.

FAQ È necessario includere i costruttori nel diagramma delle classi?

Un diagramma delle classi non deve includere tutti i metodi di una classe. Poiché un diagramma delle classi è uno strumento di progettazione, le informazioni che esso deve contenere dipendono dalla specifica situazione che si sta trattando. Normalmente i costruttori non vengono inclusi nel diagramma delle classi, perché sono sempre necessari e svolgono sempre la medesima funzione.

9.1.2 Invocare metodi da costruttori

Nel capitolo precedente si è visto che un metodo può invocare altri metodi all'interno della medesima classe. Allo stesso modo, un costruttore può invocare metodi definiti all'interno della sua classe. Per esempio, tutti i costruttori nella definizione della classe `Animale` nel Listato 9.1 possono essere rielaborati in modo che invochino uno dei metodi *set*. In particolare, la definizione del secondo costruttore di `Animale` è molto simile alla definizione del metodo `setAnimale`. Si può evitare di ripetere questo codice definendo il costruttore come di seguito:

```
public Animale(String nomeIniziale, int etaIniziale,
               double pesoIniziale) {
    setAnimale(nomeIniziale, etaIniziale, pesoIniziale);
}
```

Sebbene avere un metodo che ne invoca un altro per evitare di ripetere codice sia generalmente una buona pratica, bisogna prestare molta attenzione quando i costruttori invocano metodi pubblici delle proprie classi. Il problema ha a che fare con l'ereditarietà che sarà trattata nel Capitolo 10. Come si vedrà, un'altra classe può alterare il comportamento dei metodi pubblici e, di conseguenza, può alterare accidentalmente il comportamento di un costruttore. Il Capitolo 11 mostrerà un modo per prevenire questo problema, ma per il momento è possibile avvalersi di un'altra soluzione: definire come privato ogni metodo invocato da un costruttore. Nell'esempio precedente, porre il metodo `setAnimale` come

privato non costituisce una soluzione sensata, in quanto è necessario che esso sia pubblico. È comunque possibile definire un metodo privato che viene invocato sia dal metodo `setAnimale` sia dal costruttore, come dimostra il codice seguente:

```
public Animale(String nomeIniziale, int etaIniziale,
               double pesoIniziale) {
    set(nomeIniziale, etaIniziale, pesoIniziale);
}

public void setAnimale(String nuovoNome, int nuovaEta,
                       double nuovoPeso) {
    set(nuovoNome, nuovaEta, nuovoPeso);
}

private void set(String nuovoNome, int nuovaEta, double nuovoPeso) {
    nome = nuovoNome;
    if ((nuovaEta < 0) || (nuovoPeso < 0)) {
        System.out.println("Errore: peso o eta' negativi");
        System.exit(0);
    } else {
        eta = nuovaEta;
        peso = nuovoPeso;
    }
}
```

Gli altri costruttori e metodi *set* della classe `Animale` possono essere rielaborati in modo da invocare il metodo privato `set`, come mostra il Listato 9.3.

LISTATO 9.3 Costruttori e metodi *set* che invocano un metodo `private`.

MyLab

```
/**
  Classe riveduta che descrive un animale
*/
public class Animale2 {

    private String nome;
    private int eta;      //in anni
    private double peso; //in Kg

    public Animale2(String nomeIniziale, int etaIniziale, double pesoIniziale) {
        set(nomeIniziale, etaIniziale, pesoIniziale);
    }

    public Animale2(String nomeIniziale) {
        set(nomeIniziale, 0, 0);
    }

    public Animale2(int etaIniziale) {
        set("Nessun nome", etaIniziale, 0);
    }
```

```
    public Animale2(double pesoIniziale) {
        set("Nessun nome", 0, pesoIniziale);
    }

    public Animale2() {
        set("Nessun nome", 0, 0);
    }

    public void setAnimale(String nuovoNome, int nuovaEta,
                           double nuovoPeso) {
        set(nuovoNome, nuovaEta, nuovoPeso);
    }

    public void setNome(String nuovoNome) {
        set(nuovoNome, eta, peso);
        //eta' e peso rimangono invariate
    }

    public void setEta(int nuovaEta) {
        set(nome, nuovaEta, peso);
        //nome e peso rimangono invariate
    }

    public void setPeso(double nuovoPeso) {
        set(nome, eta, nuovoPeso);
        //nome ed eta' rimangono invariate
    }

    private void set(String nuovoNome, int nuovaEta, double nuovoPeso) {
        nome = nuovoNome;
        if ((nuovaEta < 0) || (nuovoPeso < 0)) {
            System.out.println("Errore: eta' o peso negativi.");
            System.exit(0);
        } else {
            eta = nuovaEta;
            peso = nuovoPeso;
        }
    }

    <I metodi getNome, getEta, getPeso e scriviOutput sono gli stessi del Listato 9.1>
}
```

9.1.3 Invocare un costruttore da un altro costruttore

Si consideri la classe `Animale2` del Listato 9.3. Una volta definito il primo costruttore, quello che ha tre parametri, e il metodo privato `set`, è possibile aggiungere un altro co-

struttore che invochi il primo costruttore. Per fare ciò, si utilizza la parola chiave `this` come se fosse il nome di un metodo in un'invocazione. Per esempio, l'istruzione:

```
this(nomeIniziale, 0, 0);
```

invoca il costruttore con tre parametri dal corpo di un altro costruttore della classe. L'invocazione deve essere la prima azione eseguita all'interno del corpo del costruttore. Il Listato 9.4 mostra come si potrebbero modificare i costruttori nel Listato 9.3. Si noti che i metodi *set* non usano l'istruzione `this`: essi continuano a invocare il metodo privato set come in `Animale2`. L'utilizzo della parola chiave `this` per invocare un costruttore è consentito, infatti, solo all'interno di un altro costruttore della stessa classe.

LISTATO 9.4 Costruttori che invocano un altro costruttore.

MyLab

```
/**
 Classe riveduta che descrive un animale
*/
public class Animale3 {

   private String nome;
   private int eta;        //in anni
   private double peso;    //in Kg

   public Animale3(String nomeIniziale, int etaIniziale, double pesoIniziale) {
      set(nomeIniziale, etaIniziale, pesoIniziale);
   }

   public Animale3(String nomeIniziale) {
      this(nomeIniziale, 0, 0);
   }

   public Animale3(int etaIniziale) {
      this("Nessun nome", etaIniziale, 0);
   }

   public Animale3(double pesoIniziale) {
      this("Nessun nome", 0, pesoIniziale);
   }

   public Animale3() {
      this("Nessun nome", 0, 0);
   }

   <Il resto della classe è uguale ad Animale2 nel Listato 9.3>
}
```

Costruttori che invocano altri costruttori nella stessa classe

Quando si definisce un costruttore in una classe e si desidera invocare un altro costruttore definito nella stessa classe, si utilizza la parola chiave `this` al posto del nome del costruttore. Qualsiasi chiamata a `this` deve essere la prima azione eseguita dal costruttore.

Esempio

```
public Animale3(String nomeIniziale) {
    this(nomeIniziale, 0, 0);
}
```

Scrivere costruttori interdipendenti

Quando in una classe si scrivono più costruttori, è bene identificare quello che gli altri possono invocare con la parola chiave `this`. Scrivendo i costruttori in questo modo, si localizza l'inizializzazione in un solo posto, rendendo le classi meno complesse e meno soggette a errori. Nel Capitolo 10 si entrerà più in dettaglio su questo modo di scrivere i costruttori.

9.1.4 La costante `null`

La costante `null` è una speciale costante che può essere assegnata a una variabile di qualunque tipo classe. È utilizzata per indicare che la variabile non ha alcun "valore reale". Se il compilatore richiede che una variabile di tipo classe venga inizializzata e non sono disponibili oggetti opportuni per inizializzarla, si può utilizzare il valore `null`, come nell'esempio seguente:

```
MiaClasse mioOggetto = null;
```

È anche usuale utilizzare il valore `null` nei costruttori per inizializzare variabili di tipo classe quando non è ovvio quale oggetto utilizzare.

Si noti che `null` non è un oggetto. È come un riferimento (indirizzo di un'area di memoria) che non si riferisce ad alcun oggetto (non referenzia alcuna area di memoria). Quindi, se si vuole verificare se una variabile di tipo classe ha il valore `null`, si utilizza `==` o `!=` e non un metodo `equals`. Per esempio, il codice che segue verifica correttamente se una variabile ha il valore `null`:

```
if (mioOggetto == null)
    System.out.println("Nessun oggetto reale.");
```

`null`

`null` è una costante speciale che può essere assegnata a qualunque variabile di tipo classe. La costante `null` non è un oggetto, ma una specie di marcatore utilizzato al posto del riferimento a un oggetto. Dato che viene trattata come un riferimento (indirizzo di memoria), per verificare se una variabile ha il valore `null` si utilizzano gli operatori `==` e `!=` e non un metodo `equals`.

Null Pointer Exception

Se il compilatore richiede che una variabile di tipo classe venga inizializzata, si può sempre inizializzarla a `null`. Tuttavia, `null` non è un oggetto, quindi non si può invocare un metodo utilizzando una variabile inizializzata a `null`. Se si prova a farlo, si ottiene un messaggio "*Null Pointer Exception*" (letteralmente, "eccezione di tipo puntatore a null"). Per esempio, il codice che segue produrrebbe un errore di questo tipo:

```
ProvaClasse variabile = null;
String rappresentazione = variabile.toString();
```

Il problema è che si sta cercando di invocare il metodo `toString()` utilizzando `null` come oggetto chiamante. Ma `null` non è un oggetto, è solo un marcatore che indica che la variabile non fa riferimento ad alcun oggetto reale. Di conseguenza, `null` non ha metodi. L'errore "*Null Pointer Exception*" è causato dall'utilizzo scorretto di `null`. Lo si ottiene ogniqualvolta si chiamano metodi su una variabile di tipo classe alla quale non è ancora stato assegnato un (riferimento a un) oggetto, anche se non è stato assegnato esplicitamente il valore `null` alla variabile. Quando si ottiene un errore "*Null Pointer Exception*", occorre cercare una variabile di tipo classe non correttamente inizializzata.

Inizializzazione automatica delle variabili

Si ricorda che in Java, le variabili locali a un metodo non vengono inizializzate automaticamente. Quindi, è necessario inizializzare esplicitamente una variabile locale prima di poterla utilizzare. Le variabili di istanza, al contrario, sono inizializzate automaticamente. Le variabili di istanza di tipo `boolean` sono inizializzate automaticamente al valore `false`. Le variabili di istanza degli altri tipi primitivi sono inizializzate allo "zero" del tipo corrispondente. Le variabili di istanza di tipo classe sono inizializzate automaticamente al valore `null`. Nonostante le variabili di istanza siano inizializzate automaticamente, è preferibile inizializzarle esplicitamente in un costruttore, anche quando il valore di inizializzazione coincide con quello di default, in modo da rendere più chiaro il codice.

Un modo alternativo per inizializzare le variabili di istanza

Le variabili di istanza sono solitamente inizializzate nei costruttori, che sono considerati il punto in cui è preferibile effettuare le inizializzazioni. Tuttavia, esiste un'alternativa. È possibile inizializzare le variabili di istanza al momento della loro dichiarazione nella definizione della classe, come mostrato di seguito:

```
public class Data {
    private String mese = "Gennaio";
    private int giorno = 1;
    private int anno = "2013";
    ...
}
```

Se le variabili di istanza vengono inizializzate tutte in questo modo, potrebbe non essere necessario definire dei costruttori. Se si definiscono comunque dei costruttori, in genere è meglio definire anche un costruttore di default, anche nel caso in cui questo abbia corpo vuoto.

9.2 Variabili statiche e metodi statici

Una classe può avere anche variabili statiche e metodi statici. Le variabili e i metodi statici appartengono all'intera classe e non a un singolo oggetto. I metodi statici (o di classe) sono stati introdotti nel Capitolo 5 quando ancora i concetti relativi a classi e oggetti non erano stati presentati. Ne è stata quindi fornita una visione limitata. In questo paragrafo, saranno contestualizzati e posti in relazione ai metodi di istanza definiti nel Capitolo 8.

9.2.1 Variabili statiche

Le **variabili statiche** (o **variabili di classe**) sono già state utilizzate in un caso speciale e cioè nella definizione di costanti, come nella seguente istruzione:

```
public static final int GIORNI_PER_SETTIMANA = 7;
```

Esiste solo una copia di `GIORNI_PER_SETTIMANA`. Essa appartiene alla classe e dunque i singoli oggetti della classe che contiene questa definizione non hanno una propria versione di questa costante. Pertanto, essi usano e condividono un'unica costante.

Questa particolare variabile statica non può cambiare valore (è una costante) perché la sua definizione contiene la parola chiave `final`. È possibile, comunque, definire variabili statiche che possono cambiare valore. La loro dichiarazione è simile a quella delle variabili di istanza, tranne per la parola chiave `static`. Per esempio, la seguente variabile statica può cambiare valore:

```
private static int numeroInvocazioni;
```

Una variabile statica può essere pubblica o privata. Analogamente alle variabili di istanza, anche le variabili statiche che non sono costanti, normalmente dovrebbero essere private e dovrebbero essere lette o modificate solo, rispettivamente, attraverso metodi *get* e *set*.

Esiste solo una variabile chiamata `numeroInvocazioni` e può essere letta da ogni oggetto della sua classe. Questo significa che la variabile statica potrebbe consentire agli oggetti di comunicare tra di loro o di eseguire qualche azione in modo coordinato. Per esempio, un metodo definito nella stessa classe in cui è definita la variabile statica `numeroInvocazioni` potrebbe incrementare il valore di `numeroInvocazioni` in modo da tenere traccia del numero di invocazioni di un metodo che sono state eseguite da tutti gli oggetti della classe. Il prossimo paragrafo fornisce un esempio d'uso di una variabile statica.

È possibile inizializzare una variabile statica nello stesso modo in cui si inizializza una costante, tranne che per il fatto che si deve omettere la parola chiave `final`:

```
private static int numeroInvocazioni = 0;
```

Come segnalato in precedenza, le variabili statiche sono anche chiamate variabili di classe. Per evitare confusione tra il termine *variabile di classe* e *variabile di tipo classe*, si è preferito, in questo testo, adottare il termine *variabili statiche*. Non bisogna confondere, infatti, il termine *variabile di classe*, che indica una variabile all'interno di una classe che è condivisa da tutti gli oggetti della classe che la definisce, con la nozione *variabile di tipo classe*, che indica una variabile il cui tipo è una classe, cioè, una variabile che è usata per riferire (referenziare) oggetti di una classe. A volte, sia le variabili statiche sia le variabili di istanza sono chiamate **campi** o **membri**.

Variabili statiche

La dichiarazione di una variabile statica contiene la parola chiave `static`. Una variabile statica è condivisa da tutti gli oggetti della sua classe.

Tre tipi di variabili

Java ha tre tipi di variabili: variabili locali, variabili di istanza e variabili statiche.

9.2.2 Metodi statici

I metodi presentati nel Capitolo 5 sono chiamati metodi statici. In questo paragrafo si comprenderà il loro ruolo all'interno di una classe. Per fare ciò, si riprenderanno alcuni concetti introdotti nel Capitolo 5.

A volte si ha la necessità che un metodo non abbia alcuna relazione con un oggetto qualunque sia il suo tipo. Per esempio, si potrebbe avere bisogno di un metodo per calcolare il massimo tra due interi o per calcolare la radice quadrata di un numero o per convertire una lettera minuscola in maiuscola. Nessuno di questi metodi ha un oggetto evidente a cui dovrebbe appartenere. In questi casi, si può definire il metodo come statico. Quando si definisce un **metodo statico** (o **metodo di classe**), il metodo è ancora membro di una classe, poiché è definito all'interno di una classe, ma può essere invocato senza usare alcun oggetto. Normalmente si invoca un metodo statico utilizzando il nome della classe anziché quello di un oggetto.

Come riportato sopra, un metodo statico è anche chiamato metodo di classe. Per uniformità con la terminologica usata per le variabili di classe, anche i metodi di classe saranno sempre chiamati metodi statici.

Il Listato 9.5 contiene una classe chiamata `ConvertitoreDimensioni` che definisce due metodi statici per convertire misure effettuate in piedi e in pollici. Un metodo statico si definisce nello stesso modo in cui si definisce un metodo di istanza, tranne per l'aggiunta della parola chiave `static` nell'intestazione del metodo.

LISTATO 9.5 Metodi statici.

```
/**
  Classe con metodi statici per effettuare conversioni di misure
*/
public class ConvertitoreDimensioni {
    public static final int POLLICI_PER_PIEDE = 12;

    public static double convertiPiediInPollici(double piedi) {
        return piedi * POLLICI_PER_PIEDE;
    }

    public static double convertiPolliciInPiedi(double pollici) {
        return pollici / POLLICI_PER_PIEDE;
    }
}
```

Una costante statica; potrebbe essere dichiarata `private`.

Gli esempi seguenti mostrano come vengono invocati questi metodi:

```
double piedi = ConvertitoreDimensioni.convertiPolliciInPiedi(53.7);
double pollici = ConvertitoreDimensioni.convertiPiediInPollici(2.6);
```

Come già detto nel Capitolo 5, quando si invoca un metodo statico, occorre indicare il nome della classe anziché il nome dell'oggetto. Il Listato 9.6 mostra l'uso di questi metodi in un programma completo. Sebbene sia possibile creare un oggetto della classe `ConvertitoreDimensioni` e utilizzarlo per invocare i metodi statici della classe, l'istruzione derivante potrebbe risultare poco chiara ai programmatori che leggono il codice.

Una classe che contiene solo metodi statici

Quando più metodi statici compiono operazioni tra loro correlate, è possibile raggrupparli definendoli in una sola classe.

Nella classe `ConvertitoreDimensioni` del Listato 9.5, sia la costante sia i due metodi sono statici e pubblici. Si sarebbe potuto dichiarare la costante `POLLICI_PER_PIEDE` come privata, qualora non si fosse voluto renderla accessibile al di fuori della classe. Anche se questa classe non dichiara alcuna variabile di istanza, una classe può avere variabili di istanza, variabili statiche, costanti statiche, metodi statici e metodi di istanza. Sebbene siano possibili tutte queste combinazioni, è bene prestare attenzione a come realizzarle.

LISTATO 9.6 Usare i metodi statici.

MyLab

```
import java.util.Scanner;

/**
 Esempio d'uso della classe ConvertitoreDimensioni
*/
public class ConvertitoreDimensioniDemo {
   public static void main(String[] args) {
      Scanner tastiera = new Scanner(System.in);
      System.out.print("Inserisci una misura in pollici: ");
      double pollici = tastiera.nextDouble();

      double piedi = ConvertitoreDimensioni.convertiPolliciInPiedi(pollici);
      System.out.println(pollici + " pollici = " + piedi + " piedi.");

      System.out.print("Inserisci una misura in piedi: ");
      piedi = tastiera.nextDouble();
      pollici = ConvertitoreDimensioni.convertiPiediInPollici(piedi);
      System.out.println(piedi + " piedi = " + pollici + " pollici.");
   }
}
```

Esempio di output

```
Inserisci una misura in pollici: 18
18.0 pollici = 1.5 piedi.
Inserisci una misura in piedi: 1.5
1.5 piedi = 18.0 pollici.
```

Si esaminerà ora come avviene l'interazione tra i diversi membri statici e di istanza di una classe analizzando la classe `ContoBanca` nel Listato 9.7. Questa classe contiene il saldo di un conto corrente e dichiara, quindi, una variabile di istanza chiamata `saldo`. Tutti i conti hanno il medesimo tasso di interesse, perciò la classe ha una variabile statica chiamata `tassoInteresse`. Inoltre, la classe tiene traccia del numero di conti aperti, utilizzando la variabile statica `numeroDiConti`.

LISTATO 9.7 Uso in una classe di variabili di istanza e di variabili statiche.

MyLab

```
/**
 Una classe con variabili di istanza e variabili statiche.
*/
public class ContoBanca {
   private double saldo; <-- Variabile di istanza.
   public static double tassoInteresse = 0; <-- Variabile statica.
   public static int numeroDiConti = 0; <-- Variabile statica.
```

```
    public ContoBanca() {
        saldo = 0;
        numeroDiConti++;
    }

    public static void setTassoInteresse(double nuovoTasso) {
        tassoInteresse = nuovoTasso;
    }

    public static double getTassoInteresse() {
        return tassoInteresse;
    }

    public static int getNumeroDiConti() {
        return numeroDiConti;
    }

    public void deposita(double somma) {
        saldo = saldo + somma;
    }

    public double preleva(double ammontare) {
        if (saldo >= ammontare)
            saldo = saldo - ammontare;
        else
            ammontare = 0;
        return ammontare;
    }

    public void aggiungiInteresse() {
        double interesse = saldo * tassoInteresse;
        //si puo sostituire tassoInteresse con getTassoInteresse()
        saldo = saldo + interesse;
    }

    public double getSaldo() {
        return saldo;
    }

    public static void mostraSaldo(ContoBanca conto) {
        System.out.print(conto.getSaldo());
    }
}
```

Un metodo di istanza può accedere a una variabile statica.

Un metodo statico può accedere a una variabile statica, ma non a una variabile di istanza.

Un metodo di istanza può accedere a una variabile statica o invocare un metodo statico.

Un metodo statico non può invocare un metodo di istanza, a meno che non lo faccia tramite un oggetto.

La classe ha metodi statici *get* e *set* per il tasso di interesse, un metodo statico *get* per il numero di conti e metodi di istanza per depositare, per prelevare, per aggiungere gli interessi e per ottenere il saldo del conto. Infine, la classe ha un metodo statico per mostrare il saldo di ogni conto.

Nella definizione di un metodo statico, non è possibile far riferimento a una variabile di istanza. Infatti, dato che un metodo statico può essere invocato senza utilizzare alcun oggetto, non esiste alcuna variabile di istanza alla quale ci si può riferire. Per esempio, il metodo statico `setTassoInteresse` può fare riferimento alla variabile statica `tassoInteresse`, ma non alla variabile di istanza `saldo`, che non è statica.

Un metodo statico non può invocare un metodo di istanza senza avere un oggetto da usare nell'invocazione. Per esempio, il metodo statico `mostraSaldo` accetta come parametro un oggetto di tipo `ContoBanca`. Il metodo usa l'oggetto per invocare il metodo di istanza `getSaldo`. Infatti, `mostraSaldo` può ottenere informazioni sul saldo di un conto solo attraverso un oggetto che rappresenti il conto.

Il metodo di istanza `aggiungiInteresse` può far riferimento alla variabile statica `tassoInteresse` oppure invocare il metodo statico `getTassoInteresse`. Nell'invocazione di `getTassoInteresse` si può far precedere al nome del metodo il nome della classe seguito da un punto. In questo caso, tuttavia, ciò è opzionale, perché i metodi sono tutti contenuti in `ContoBanca`. Si noti, infine, che il costruttore incrementa la variabile statica `numeroDiConti` così da contare il numero di volte che è invocato. Questo permette di contare il numero di nuovi conti.

Il Listato 9.8 contiene un semplice programma che mostra l'utilizzo della classe `ContoBanca`.

LISTATO 9.8 Usare metodi di istanza e statici.

MyLab

```
public class ContoBancaDemo {

    public static void main(String[] args) {
        ContoBanca.setTassoInteresse(0.01);
        ContoBanca mioConto = new ContoBanca();
        ContoBanca tuoConto = new ContoBanca();

        System.out.println("Ho depositato 10.75 Euro.");
        mioConto.deposita(10.75);
        System.out.println("Hai depositato 75 Euro.");
        tuoConto.deposita(75.00);
        System.out.println("Hai depositato 55 Euro.");
        tuoConto.deposita(55.00);

        double contante = tuoConto.preleva(15.75);
        System.out.println("Hai prelevato " + contante + " Euro.");
        if (tuoConto.getSaldo() > 100.00) {
            System.out.println("Hai ricevuto un interesse.");
            tuoConto.aggiungiInteresse();
        }
        System.out.println("Il tuo conto e' di " +
                           tuoConto.getSaldo() + " Euro.");
```

```
        System.out.print("Il mio conto e' di ");
        ContoBanca.mostraSaldo(mioConto);
        System.out.println(" Euro.");
        int conti = ContoBanca.getNumeroDiConti();
        System.out.println("Abbiamo aperto " + conti +
                           " conti in banca oggi.");
    }
}
```

Esempio di output

```
Ho depositato 10.75 Euro.
Hai depositato 75 Euro.
Hai depositato 55 Euro.
Hai prelevato 15.75 Euro.
Hai ricevuto un interesse.
Il tuo conto e' di 115.3925 Euro.
Il mio conto e' di 10.75 Euro.
Abbiamo aperto 2 conti in banca oggi.
```

Invocazione di un metodo di istanza all'interno di uno statico

Spesso si sente dire: "Non è possibile invocare un metodo di istanza all'interno della definizione di un metodo statico".

Questo non è completamente vero. Una frase più precisa e corretta è: "Non è possibile invocare un metodo di istanza all'interno di un metodo statico *a meno che non si abbia un oggetto da utilizzare nella chiamata del metodo di istanza*".

In altre parole, nella definizione di un metodo statico, come il metodo `main`, non si può utilizzare un metodo che ha come oggetto chiamante un oggetto `this` implicito o esplicito.

Metodi statici

Se si usa la parola chiave `static` nell'intestazione di un metodo, il metodo può essere invocato usando il nome della classe che lo definisce al posto del nome di un oggetto. Dal momento che un metodo statico non ha bisogno di un oggetto per essere invocato, non può far riferimento a variabili di istanza della classe. Non può nemmeno invocare un metodo di istanza della classe, a meno che non abbia un oggetto della classe e utilizzi questo oggetto nell'invocazione. In altre parole, la definizione di un metodo statico non può utilizzare una variabile di istanza o un metodo di istanza che abbia come oggetto chiamante un `this` esplicito o implicito.

9.2.3 Suddividere le attività del metodo `main` in sotto-attività

Quando un programma definito nel metodo `main` ha una logica complicata o il suo codice è ripetitivo, è possibile definire più metodi statici nei quali eseguire le varie sotto-attività e richiamare questi metodi dal metodo `main`. Per illustrare questa tecnica viene ripreso il programma nel Listato 8.15 del capitolo precedente che era stato usato per collaudare i metodi `equals` della classe `Specie`. Il Listato 9.9 riproduce tale programma evidenziando due sezioni di codice che sono identiche. Anziché ripetere il codice, è possibile inserirlo in un metodo statico e invocarlo due volte all'interno del metodo `main`.

Questa modifica è stata riportata nel Listato 9.10, dove è stato inoltre definito un metodo aggiuntivo per il codice racchiuso all'interno del rettangolo nel Listato 9.9. I due nuovi metodi ausiliari sono statici e privati.

LISTATO 9.9 Un metodo `main` con codice ripetuto.

MyLab

```
public class SpecieEqualsDemo {

   public static void main(String[] args) {
      Specie s1 = new Specie(), s2 = new Specie();

      s1.setSpecie("Bufalo Klingon", 10, 15);
      s2.setSpecie("Bufalo Klingon", 10, 15);

      if (s1 == s2)
         System.out.println("Corrispondono secondo ==.");
      else
         System.out.println("Non corrispondono secondo ==.");

      if (s1.equals(s2))
         System.out.println("Corrispondono secondo il metodo equals.");
      else
         System.out.println("Non corrispondono secondo il metodo equals.");

      System.out.println("Ora cambiamo un Klingon in lettere minuscole.");
      s2.setSpecie("klingon", 10, 15);    //Usa minuscole

      if (s1.equals(s2))
         System.out.println("Corrispondono secondo il metodo equals.");
      else
         System.out.println("Non corrispondono secondo il metodo equals.");
   }
}
```

LISTATO 9.10 Un metodo `main` che usa metodi ausiliari (o di appoggio).

MyLab

```
public class SpecieEqualsDemo2 {

   public static void main(String[] args) {
      Specie s1 = new Specie(), s2 = new Specie();
```

Un metodo di istanza può accedere a una variabile statica.

```
        s1.setSpecie("Bufalo Klingon", 10, 15);
        s2.setSpecie("Bufalo Klingon", 10, 15);

        testConOperatoreUguale(s1, s2);
        testConMetodoEquals(s1, s2);

        System.out.println("Ora cambiamo un Klingon in lettere minuscole.");
        s2.setSpecie("klingon", 10, 15);     //Usa minuscole

        testConOperatoreUguale(s1, s2);
    }

    private static void testConOperatoreUguale(Specie s1, Specie s2) {
        if (s1 == s2)
            System.out.println("Corrispondono secondo ==.");
        else
            System.out.println("Non corrispondono secondo ==.");
    }

    private static void testConMetodoEquals(Specie s1, Specie s2) {
        if (s1.equals(s2))
            System.out.println("Corrispondono secondo il metodo equals.");
        else
            System.out.println("Non corrispondono secondo il metodo equals.");
    }
}
```

Metodi ausiliari per il metodo `main`

Quando si sviluppano programmi, anche semplici, è bene semplificare la logica del metodo `main` di un'applicazione attraverso invocazioni di metodi ausiliari. Questi metodi dovrebbero essere statici, in quanto `main` è statico. Poiché questi sono metodi ausiliari, dovrebbero essere anche privati.

Il metodo `main` è statico

Dato che il metodo `main` è statico, è necessario rispettare le indicazioni fornite nel paragrafo precedente dedicato ai metodi statici. In generale, un metodo statico può invocare solo metodi statici e far riferimento solo a variabili statiche. Non si può invocare un metodo di istanza della stessa classe a meno che non si abbia un oggetto della classe da utilizzare per l'invocazione.

9.2.4 Aggiungere un metodo `main` a una classe

Finora, ogni volta che si è scritto un metodo `main` è stato definito in una classe a parte, all'interno di un file distinto, ad eccezione di alcuni esempi mostrati nel Capitolo 5. Tuttavia, a volte ha senso avere un metodo `main` all'interno di una definizione di classe. La classe può quindi avere due scopi: può essere utilizzata per creare oggetti in altre classi oppure può essere eseguita come un programma. Per esempio, è possibile scrivere un metodo `main` all'interno della definizione della classe `Specie` fornita nel Listato 8.16 del capitolo precedente. Il risultato è mostrato nel Listato 9.11.

LISTATO 9.11 Inserire un metodo `main` in una definizione di classe.

```
import java.util.Scanner;

public class Specie {
    private String nome;
    private int popolazione;
    private double tassoCrescita;

    <I metodi leggiInput, scriviOutput, prediciPopolazione, setSpecie, getNome,
    getPopolazione, getTassoDiCrescita e equals vanno in questa posizone.
    Sono gli stessi definiti nel Listato 8.16>

    public static void main(String args[]){
        Specie specieAdOggi = new Specie();

        System.out.println("Inserisci i dati sulla specie ad oggi:");
        specieAdOggi.leggiInput();
        specieAdOggi.scriviOutput();
        System.out.println("Inserisci il numero di anni" +
                           " su cui calcolare la proiezione:");

        Scanner tastiera = new Scanner(System.in);
        int numeroAnni = tastiera.nextInt();
        int popolazioneFutura = specieAdOggi.prediciPopolazione(numeroAnni);
        System.out.println("Tra " + numeroAnni +
                           " la popolazione sara' di " + popolazioneFutura);
        specieAdOggi.setSpecie("Felini", 10, 15);
        System.out.println("La nuova specie e':");
        specieAdOggi.scriviOutput();
    }
}
```

Dopo aver ridefinito la classe `Specie` in questo modo, se la si esegue come programma viene invocato il suo metodo `main`. Quando invece `Specie` è usata come una classe ordinaria per creare oggetti, il metodo `main` viene ignorato.

Poiché un metodo `main` è statico, non può contenere una chiamata di un metodo di istanza della stessa classe a meno che al suo interno non sia definito un oggetto della classe stessa e che si usi questo oggetto per l'invocazione del metodo di istanza. Il metodo `main` del Listato 9.11 invoca i metodi della classe `Specie` creando prima un oggetto della classe `Specie` e poi utilizzandolo per richiamare altri metodi. È necessario operare in questo modo anche se il metodo `main` si trova all'interno della definizione della classe `Specie`.

Aggiungere un metodo `main` alla classe per collaudarla

Non si dovrebbe inserire un metodo `main` nella definizione di una classe che deve essere utilizzata per creare oggetti. Tuttavia, l'inserimento di un piccolo metodo `main` all'interno della definizione della classe fornisce un modo pratico per collaudarla.

9.2.5 Classi wrapper

Java distingue fra i tipi primitivi, come `int`, `double` e `char`, e i tipi classe, come la classe `String` e le classi definite dal programmatore. Per esempio, si è visto nel Capitolo 8 che un argomento di un metodo è trattato in modo diverso a seconda che l'argomento sia di un tipo primitivo o di un tipo classe. Se un metodo ha bisogno di un argomento di un tipo classe, ma si ha a disposizione un valore di tipo primitivo, è necessario convertire il valore primitivo (come il valore di tipo `int` 42) in un equivalente "valore" di un certo tipo classe che corrisponde al tipo `int` primitivo. Per effettuare questa conversione Java fornisce una **classe wrapper** (letteralmente "involucro") per ciascuno dei tipi primitivi. Tali classi definiscono i metodi che possono agire sui valori di un tipo primitivo.

Per esempio, la classe *wrapper* per il tipo primitivo `int` è la classe predefinita `Integer`. Se si vuole convertire un valore `int`, per esempio `42`, in un oggetto di tipo `Integer`, occorre utilizzare la seguente forma:

```
Integer n = new Integer(42);
```

Dopo l'esecuzione dell'istruzione precedente, `n` referenzia un oggetto della classe `Integer` che corrisponde al valore `42` di tipo `int`. Infatti, l'oggetto `n` ha una variabile di istanza contenente il valore `int` `42`. Al contrario, la seguente dichiarazione converte un oggetto di tipo `Integer` in un valore `int`:

```
int i = n.intValue();
```

Il metodo `intValue` estrae il valore equivalente `int` da un oggetto di tipo `Integer`.

Le classi *wrapper* per i tipi primitivi `long`, `float`, `double` e `char` sono rispettivamente `Long`, `Float`, `Double` e `Character`. Naturalmente al posto del metodo `intValue`, le classi `Long`, `Float`, `Double` e `Character` utilizzano, rispettivamente, i metodi `longValue`, `floatValue`, `doubleValue` e `charValue`.

La conversione da un tipo primitivo, come `int`, alla sua classe *wrapper* corrispondente, come `Integer`, viene svolta automaticamente da Java. Questo tipo di conversione si chiama **boxing**. Si può pensare l'oggetto come a una "scatola" in cui si inserisce il valore del tipo primitivo e lo si assegna come valore a una variabile di istanza privata. Così le dichiarazioni:

```
Integer n = new Integer(42);
```

```
Double d = new Double(9.99);
Character c = new Character('Z');
```

possono essere scritte più semplicemente come:

```
Integer n = 42;
Double d = 9.99;
Character c = 'Z';
```

Questi semplici assegnamenti sono in realtà solo abbreviazioni per le versioni più lunghe che includono l'operatore new. La situazione è analoga a ciò che accade quando un valore di tipo int viene assegnato a una variabile di tipo double: viene operata automaticamente una conversione di tipo.

La conversione inversa da un oggetto di una classe *wrapper* al valore del suo tipo primitivo associato è chiamata **unboxing**. Anche l'operazione di *unboxing* viene svolta automaticamente da Java. Quindi, se n referenzia un oggetto della classe Integer, l'istruzione:

```
int i = n.intValue();
```

può essere scritta in modo abbreviato come:

```
int i = n;
```

Di seguito vengono riportati altri esempi di *unboxing* automatico:

```
int i = new Integer(42);
double f = new Double(9.99);
char s = new Character('Z');
```

Quello che accade realmente è che Java applica automaticamente il metodo appropriato (in questi esempi intValue, doubleValue o charValue) per ottenere il valore del tipo primitivo che viene assegnato alla variabile.

Le operazioni di *boxing* e *unboxing* automatico si applicano ai parametri così come alle dichiarazioni di assegnamento che sono state appena discusse. È possibile assegnare un valore di un tipo primitivo, per esempio un valore di tipo int, a un parametro della classe *wrapper* associata, come Integer. Allo stesso modo, è possibile assegnare un argomento *wrapper* di tipo classe, come un argomento di tipo Integer, a un parametro di tipo primitivo associato, per esempio int.

L'importanza principale delle classi *wrapper* è che esse contengono una serie di costanti e metodi statici molto utili. Per esempio, è possibile utilizzare la classe *wrapper* associata per sapere il più grande e più piccolo valore associabile ai corrispondenti tipi primitivi. Il valore più grande e più piccolo per il tipo int sono:

```
Integer.MAX_VALUE e Integer.MIN_VALUE
```

I valori più grande e più piccolo per il tipo double sono rispettivamente:

```
Double.MAX_VALUE e Double.MIN_VALUE
```

MAX_VALUE e MIN_VALUE sono costanti statiche definite in ciascuna delle classi *wrapper* Integer, Long, Float, Double e Character. Per esempio, la classe Integer definisce MAX_VALUE come:

```
public static final int MAX_VALUE = 2147483647;
```

I metodi statici definiti nelle classi *wrapper* possono essere usati per convertire una stringa al corrispondente numero di tipo `int`, `double`, `long` o `float`. Per esempio, il metodo statico `parseDouble` della classe *wrapper* `Double` converte una stringa in un valore di tipo `double`. Così

```
Double.parseDouble("199.98")
```

restituisce il valore 199.98 di tipo `double`. Naturalmente questo si poteva sapere semplicemente osservando l'istruzione. Ma la stessa tecnica può essere utilizzata per modificare il valore di una variabile stringa. Per esempio, si supponga che `laStringa` sia una variabile di tipo `String` il cui valore è la rappresentazione in forma di stringa di un numero di tipo `double`. Il codice seguente restituisce il valore corrispondente al valore `double` della stringa `laStringa`:

```
Double.parseDouble(laStringa)
```

Se sussiste la possibilità che `laStringa` contenga spazi vuoti iniziali o finali, si dovrebbe usare

```
Double.parseDouble(laStringa.trim())
```

Il metodo `trim` è definito nella classe `String` ed elimina gli spazi bianchi (*whitespace*) iniziali e finali, come gli spazi vuoti. È sempre più sicuro usare `trim` quando si richiama un metodo come `parseDouble`. Infatti, non si può prevedere con certezza l'assenza di spazi bianchi iniziali o finali.

Se la stringa non è un numero, l'invocazione del metodo `Double.parseDouble` causerà la fine del programma.

La conversione di una stringa in un numero può essere svolta con una qualsiasi delle classi *wrapper* `Integer`, `Long`, `Float`, così come è stato fatto con la classe *wrapper* `Double`. Basta utilizzare uno dei metodi statici `Integer.parseInt`, `Long.parseLong` o `Float.parseFloat` invece del metodo `Double.parseDouble`.

Ogni *wrapper* delle classi numeriche ha anche un metodo statico chiamato `toString` che converte nella direzione opposta: converte, cioè, un valore numerico nella sua rappresentazione stringa. Per esempio

```
Integer.toString(42)
```

restituisce il valore `"42"`, di tipo `String`, e

```
Double.toString(199.98)
```

restituisce il valore il valore `"199.98"` sempre di tipo `String`.

`Character` è il *wrapper* per il tipo primitivo `char`. Il codice seguente mostra alcuni metodi base di questa classe:

```
Character c1 = new Character('a');
Character c2 = new Character('A');
if (c1.equals(c2))
    System.out.println(c1.charValue() + " e' uguale a " +
                       c2.charValue());
else
    System.out.println(c1.charValue() + " non e' uguale a " +
                       c2.charValue());
```

Nome	Descrizione	Tipo di argomento	Tipo restituito	Esempio	Valore restituito
toUpperCase	Converte in maiuscolo	char	char	`Character.toUpperCase('a')` `Character.toUpperCase('A')`	`'A'` `'A'`
toLowerCase	Converte in minuscolo	char	char	`Character.toLowerCase('a')` `Character.toLowerCase('A')`	`'a'` `'a'`
isUpperCase	Verifica se è maiuscola	char	boolean	`Character.isUpperCase('A')` `Character.isUpperCase('a')`	`true` `false`
isLowerCase	Verifica se è minuscola	char	boolean	`Character.isLowerCase('A')` `Character.isLowerCase('a')`	`false` `true`
isLetter	Verifica se è una lettera	char	boolean	`Character.isLetter('A')` `Character.isLetter('%')`	`true` `false`
isDigit	Verifica se è una cifra	char	boolean	`Character.isDigit('5')` `Character.isDigit('A')`	`true` `false`
isWhitespace	Verifica la presenza di spazi bianchi	char	boolean	`Character.isWhitespace(' ')` `Character.isWhitespace('A')`	`true` `false`
Gli "spazi bianchi" sono caratteri che vengono stampati come spazi vuoti, come lo spazio, la tabulazione ('\t'), e il carattere di nuova riga ('\n')					

Figura 9.3 Metodi statici della classe `Character`.

Questo codice produce il seguente output:

```
a non e' uguale a A
```

Il metodo `equals` controlla l'uguaglianza fra i caratteri; le lettere maiuscole e minuscole sono considerate differenti.

La classe `Character` presenta anche altri metodi statici molto utili. Alcuni di questi sono elencati nella Figura 9.3.

Java definisce anche una classe *wrapper* per il tipo primitivo `boolean`. Questa classe definisce due costanti di tipo `boolean`: `Boolean.TRUE` e `Boolean.FALSE`. Tuttavia, le parole chiave di Java `true` e `false` sono molto più comode da usare.

Le classi wrapper

Ogni tipo primitivo ha una corrispondente classe *wrapper*. Le classi *wrapper* permettono di avere un oggetto di tipo classe che corrisponde a un valore di un tipo primitivo. Le classi *wrapper* contengono anche una serie di utili costanti e metodi predefiniti. Ogni classe *wrapper* ha due usi connessi, ma distinti.

Per esempio, è possibile creare oggetti della classe *wrapper* `Integer` che corrispondono a valori di tipo `int`, come in:

```
Integer n = new Integer(42);
```

La classe *wrapper* `Integer` serve anche come libreria di metodi statici, come il metodo `parseInt`:

```
String stringaNumerica;
...
int numero = Integer.parseInt(stringaNumerica);
```

Ogni programma può utilizzare una classe *wrapper* in entrambi i modi.

Le classi wrapper non hanno un costruttore di default

Le classi *wrapper* `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, `Short` non hanno costruttori di default. Così, una dichiarazione come:

```
Integer n = new Integer();
```

non è corretta. Quando si crea un oggetto di una di queste classi bisogna fornire un valore di inizializzazione come argomento del costruttore, come nell'esempio seguente:

```
Character mioCarattere = new Character('Z');
```

Le classi *wrapper* non hanno metodi *set*, quindi non è possibile modificare il valore degli oggetti di queste classi.

9.3 Overloading

I capitoli precedenti hanno mostrato come due o più classi possono definire metodi che hanno lo stesso nome. Per esempio, più classi possono aver definito il metodo `leggiInput`. Questo non è un problema: il tipo dell'oggetto che richiama il metodo permette a Java di capire quale definizione del metodo `leggiInput` utilizzare. Una caratteristica più sorprendente, invece, riguarda il fatto che Java permette di avere più metodi con lo stesso nome all'interno di una stessa classe.

9.3.1 Concetti di base dell'overloading

Quando si assegna lo stesso nome a due o più metodi *all'interno di una stessa classe* si dice che si sta effettuando l'**overloading** del nome del metodo, letteralmente si sta "sovraccaricando" di significati il metodo. Per compiere questa operazione è necessario che le definizioni dei diversi metodi presentino differenze nell'elenco di parametri che ricevono.

Il Listato 9.12 contiene un esempio molto semplice di *overloading*. Qui la classe `Overload` definisce tre metodi statici diversi tra loro, tutti chiamati `calcolaMedia`. In questo esempio, i metodi sono statici perché, essendo puramente funzionali, non lavorano con le variabili di istanza di un oggetto, ma solo con gli argomenti passati in ingresso

ai metodi. Quando viene invocato il metodo `Overload.calcolaMedia`, Java deve capire quale metodo eseguire. Se tutti i argomenti in ingresso sono di tipo `double`, Java riesce a capire quale metodo utilizzare sulla base del numero di argomenti: se il metodo `calcolaMedia` viene invocato con due argomenti di tipo `double`, Java usa la prima definizione; se viene invocato con tre parametri usa la seconda.

Ora si supponga che in un'invocazione al metodo `calcolaMedia` vengano passati due argomenti di tipo `char`. In questo caso, Java utilizza la terza definizione di `calcolaMedia`, specifica per il tipo di argomenti passati.

LISTATO 9.12 Overloading.

MyLab

```
/**
 Questa classe illustra la tecnica di overloading.
*/
public class Overload {

   private static double calcolaMedia(double primo, double secondo) {
      return (primo + secondo) / 2.0;
   }

   private static double calcolaMedia(double primo, double secondo,
                                      double terzo) {
      return (primo + secondo + terzo) / 3.0;
   }

   private static char calcolaMedia(char primo, char secondo) {
      return (char)(((int)primo + (int)secondo) / 2);
   }

   public static void main(String args[] ) {
      double media1 = Overload.calcolaMedia(40.0, 50.0);
      double media2 = Overload.calcolaMedia(1.0, 2.0, 3.0);
      char media3 = Overload.calcolaMedia('a', 'c');

      System.out.println("media1 = " + media1);
      System.out.println("media2 = " + media2);
      System.out.println("media3 = " + media3);
   }
}
```

Esempio di output

```
media1 = 45.0
media2 = 2.0
media3 = b
```

Calcolare la media di due caratteri

Nell'esempio presentato nei paragrafi precedenti non era necessario capire come fosse implementata la media tra due caratteri, tuttavia la tecnica usata nel Listato 9.12 rappresenta un modo interessante per calcolare la media tra caratteri. Se due caratteri sono entrambi minuscoli, la media corrisponderà alla lettera minuscola che si trova a metà strada tra di essi nell'ordine alfabetico. Analogamente, se le due lettere sono maiuscole, la media corrisponderà alla lettera maiuscola che si trova a metà strada tra di esse nell'ordine alfabetico. Questo approccio funziona in quanto i caratteri Unicode sono numerati in ordine crescente rispetto all'ordine alfabetico. Il numero assegnato al carattere 'b' è maggiore di un'unità rispetto al numero assegnato al carattere 'a', così come il numero assegnato a 'c' è maggiore di un'unità rispetto al numero assegnato al carattere 'b' e così via. Per questo motivo, se si convertono due caratteri in numeri, si calcola la media tra i numeri ottenuti e si converte la media ottenuta nuovamente in caratteri, si ottiene la lettera che si trova a metà strada tra i due caratteri dati.

Se si effettua l'*overloading* di un nome di metodo, cioè se si assegna lo stesso nome a più definizioni di metodi di una stessa classe, Java distingue i metodi sulla base del numero di parametri che ricevono e dei tipi di parametri. Se un'invocazione di metodo corrisponde alla definizione di un metodo in termini di nome, tipo del primo argomento, tipo del secondo argomento e così via, allora Java esegue quel metodo. Se non esiste alcun metodo che corrisponde (in questi termini di confronto) al metodo invocato, allora Java prova a eseguire automaticamente alcune delle conversioni di tipo presentate nei capitoli precedenti (per esempio la conversione di tipi `int` in tipi `double`) per verificare se trova una corrispondenza. Se non c'è alcuna corrispondenza nemmeno in questo caso, Java restituisce un errore durante la compilazione del programma.

Il nome di un metodo, il numero e il tipo di parametri che utilizza sono detti **firma** (*signature*) del metodo. Un altro modo per descrivere la regola dell'*overloading* consiste, quindi, nell'affermare che tutti i metodi di una classe devono avere firme diverse. Una classe, quindi, non può definire due metodi con la stessa firma.

Si noti che l'*overloading* è già stato utilizzato all'interno di questo testo anche se il termine non era stato ancora introdotto. Infatti, i metodi `print` e `println` della classe `PrintStream` della Java Class Library sono stati definiti sfruttando l'*overloading*. Ciascuno di questi metodi, infatti, riceve un solo argomento, ma, in una versione questo argomento è di tipo `String`, in un'altra di tipo `int`, in un'altra di tipo `double` e così via. Anche molti metodi della classe `Math`, discussa nel Capitolo 5, sfruttano l'*overloading*. Per esempio, la classe `Math` include diverse versioni del metodo `max`, ciascuna delle quali riceve due argomenti. Il tipo di questi argomenti permette di individuare il metodo corretto da invocare. Per esempio, se gli argomenti passati al metodo sono di tipo `int`, il metodo `max` restituisce un valore di tipo `int`. Se i due argomenti sono di tipo `double`, restituisce un valore di tipo `double`.

L'*overloading* può essere applicato a qualsiasi tipo di metodo: a metodi `void`, a metodi che restituiscono un valore, a metodi statici, a metodi di istanza o a una qualsiasi combinazione di questi. L'*overloading* può essere applicato anche ai costruttori. Per esempio, tutti i costruttori della classe `Animale` nel Listato 9.1 hanno lo stesso nome e per

questo motivo sono frutto di un *overloading*. Tali costruttori costituiscono un semplice esempio e saranno quindi utilizzati per analizzare nel dettaglio l'*overloading*.

Di seguito sono riportate le firme dei costruttori definiti nel Listato 9.1:

```
public Animale()
public Animale(String nomeIniziale, int etaIniziale, double pesoIniziale)
public Animale(String nomeIniziale)
public Animale(int etaIniziale)
public Animale(double pesoIniziale)
```

Si noti che ciascun costruttore presenta o un diverso numero di parametri oppure un parametro il cui tipo è diverso da quello degli altri costruttori. Ciascuna invocazione a un costruttore coinvolge, quindi, una diversa definizione.

Per esempio, ognuno dei seguenti oggetti di tipo `Animale` viene creato da un'invocazione a un costruttore differente:

```
Animale unAnimale = new Animale();
Animale mioGatto = new Animale("Speedy", 2, 4.5);
Animale mioCane = new Animale("Fido");
Animale miaTartaruga = new Animale(20);
Animale mioCavallo = new Animale(300.6);
```

La prima istruzione invoca il costruttore di default, in quanto non viene fornito alcun argomento. Nella seconda istruzione l'oggetto assegnato alla variabile `mioGatto` è il risultato dell'invocazione del costruttore che riceve tre argomenti. Ciascuna delle istruzioni rimanenti invoca un costruttore che richiede un solo argomento. Java sceglie il costruttore la cui firma contiene un parametro dello stesso tipo dell'argomento passato nell'invocazione del metodo. Perciò, l'istruzione `Animale("Fido")` causa l'invocazione del costruttore che riceve un valore di tipo `String`, mentre `Animale(20)` causa l'invocazione del costruttore il cui parametro è di tipo `int`, mentre `Animale(300.5)` invoca il costruttore il cui parametro è di tipo `double`.

Overloading

All'interno di una stessa classe si possono avere due o più definizioni di un metodo che presentano lo stesso nome, ma che hanno un diverso numero di parametri oppure parametri di diverso tipo. In pratica, si possono avere metodi che hanno lo stesso nome, ma una firma differente. Questa caratteristica è detta *overloading* del nome del metodo.

9.3.2 Overloading e conversione automatica di tipo

In certi casi, due ingredienti, buoni se presi singolarmente, interagiscono male e portano a un risultato scadente se inseriti insieme in una ricetta. La stessa cosa accade anche in Java. L'*overloading* è una funzionalità utile del linguaggio Java. La conversione automatica di tipo degli argomenti (cioè il fatto che un `int` come 2 viene convertito nel valore `double` 2.0 quando un metodo richiede come argomento un tipo `double`) è un'altra funzionalità utile del linguaggio Java. Tuttavia, queste due funzionalità possono, a volte, intralciarsi e portare a risultati non corretti.

Per esempio l'istruzione:

```
Animale mioCavallo = new Animale(300.0);
```

crea un oggetto di tipo `Animale` il cui peso è di 300.0 Kg. Si supponga ora di dimenticarsi del punto decimale e dello zero che segue la virgola, scrivendo quindi:

```
Animale mioCavallo = new Animale(300);
```

Invece di creare un oggetto di tipo `Animale` il cui peso è di 300 Kg, in questo caso si crea un oggetto di tipo `Animale` la cui età è 300. Dato che l'argomento 300 è di tipo `int`, questo corrisponde al costruttore che ha un parametro di tipo `int`. Questo costruttore assegna il valore dell'argomento alla variabile di istanza `eta` e non alla variabile di istanza `peso`. Se Java riesce a trovare una definizione di metodo i cui parametri corrispondono per numero e per tipo agli argomenti passati, non effettuerà alcuna conversione di tipo.

Nel caso appena mostrato sarebbe necessaria una conversione di tipo che tuttavia non viene effettuata. Ci sono inoltre casi in cui non si vuole una conversione di tipo e invece ne viene effettuata una. Si supponga, per esempio, che il nome di `mioCane` sia `Fuffy`, il peso di 2 kg e l'età di 3 anni. Si potrebbe eseguire la seguente istruzione:

```
Animale mioCane = new Animale("Fuffy", 2, 3);
```

Quest'istruzione assegnerebbe a `mioCane` un'età di 2 anni (invece di 3) e gli assegnerebbe un peso di 3.0 Kg (invece di 2). Chiaramente il problema è che sono stati invertiti il secondo e il terzo argomento, ma è bene considerare il problema dal punto di vista di Java. Data l'invocazione precedente, Java cerca un costruttore la cui intestazione abbia la seguente forma:

```
public Animale(String par_1, int par_2, int par_3)
```

`Animale` non ha un costruttore di questo tipo, perciò non esiste alcuna corrispondenza esatta per questo costruttore. Allora Java prova a convertire un `int` in un `double` per trovare una corrispondenza e nota che se converte `3` in `3.0` ottiene una corrispondenza con il metodo:

```
public Animale(String nuovoNome, int nuovaEta, double nuovoPeso )
```

e quindi effettua la conversione di tipo.

L'errore del programmatore sta nel fatto che, non solo ha invertito i due argomenti, ma ha anche scritto il peso come 2, invece che 2.0. Se avesse utilizzato 2.0 avrebbe ottenuto un messaggio d'errore. La conversione automatica di tipo svolta da Java in questo caso non è stata utile.

L'overloading precede la conversione automatica di tipo

Java cerca di usare l'*overloading* prima di usare la conversione automatica di tipo. Se Java individua una definizione di metodo che corrisponde al tipo di argomenti fornito, utilizza tale definizione. Java non effettua una conversione automatica di tipo degli argomenti passati a un metodo finché non si accerta che non esista una definizione di metodo che corrisponde al tipo degli argomenti passati al metodo.

Overloading e conversione automatica di tipo

A volte, un'invocazione di metodo può essere risolta da Java in due diversi modi che dipendono da come interagiscono *overloading* e conversione di tipo. In Java invocazioni ambigue non sono permesse e generano un messaggio d'errore a run-time o durante la compilazione. Per esempio, il nome del metodo `metodoProblematico` appartenente alla classe `ClasseDiEsempio` potrebbe essere definito sfruttando il seguente *overloading*:

```
public class ClasseDiEsempio {
    public static void metodoProblematico(double n1, int n2) ...
    public static void metodoProblematico(int n1, double n2) ...
}
```

Questa classe verrebbe compilata senza problemi. Tuttavia un'invocazione come la seguente:

```
ClasseDiEsempio.metodoProblematico(5, 10);
```

causerebbe un messaggio d'errore, in quanto Java non potrebbe decidere quale definizione considerare: convertire il primo valore intero in `double` e usare la prima definizione del metodo oppure convertire il secondo valore intero in `double` e usare la seconda definizione del metodo? In questo caso Java genera un messaggio d'errore che indica che l'invocazione di metodo è ambigua.

Le seguenti invocazioni di metodo sono invece permesse:

```
ClasseDiEsempio.metodoProblematico(5.0, 10);
ClasseDiEsempio.metodoProblematico(5, 10.0);
```

Tuttavia queste situazioni, sebbene valide, dovrebbero essere evitate.

Scegliere nomi descrittivi per evitare l'overloading

Si considerino i quattro metodi `set` della classe `Animale` nel Listato 9.1. Se si fosse utilizzato lo stesso nome per tutti e quattro i metodi, si sarebbero incontrate le stesse difficoltà descritte nei paragrafi precedenti. Proprio per evitare queste difficoltà, invece di usare lo stesso nome, sono stati utilizzati nomi differenti: al posto di chiamare ogni metodo `set`, sono stati usati nomi più descrittivi come `setNome`, `setEta`, `setPeso` e `setAnimale`. L'*overloading* deve essere utilizzato quando si ha una buona ragione per farlo, ma non quando nomi di metodo differenti sarebbero più descrittivi. Questa indicazione chiaramente non può essere seguita quando si definiscono i costruttori dal momento che devono tutti avere lo stesso nome della classe. Per questo motivo, infatti, l'*overloading* dei costruttori è molto comune.

9.3.3 Overloading e tipo di ritorno

Non si può effettuare l'*overloading* di un nome di metodo fornendo due definizioni la cui intestazione differisce solo per il tipo del valore restituito. Per esempio, si potrebbe voler aggiungere alla classe `Animale` del Listato 9.1 un altro metodo `getPeso` che restituisca un carattere che indichi se l'animale è sottopeso (restituendo il carattere `'-'`), sovrappeso (restituendo il carattere `'+'`) o abbia un peso giusto (restituendo il carattere `'*'`). Il valore restituito del metodo sarebbe `char`. Si supponga, quindi, di avere le seguenti intestazioni:

```
/**
 Restituisce il peso dell'animale.
*/
public double getPeso()

/**
 Restituisce '+' se sovrappeso, '-' se sottopeso
 e '*' se il peso e' OK.
*/
public char getPeso()
```

NON è possibile avere entrambi i metodi nella stessa classe.

Sfortunatamente ciò non è valido. Infatti, in ogni definizione di classe i metodi con lo stesso nome devono presentare un numero diverso di parametri o almeno un parametro di tipo diverso. Non si può effettuare l'*overloading* solo sulla base del tipo di ritorno. La firma di un metodo non include il tipo di ritorno, perciò le firme dei due metodi `getPeso` sono uguali, il che è illegale.

È facile capire perché non è proprio possibile implementare un compilatore che effettui l'*overloading* sulla base solo del tipo di ritorno. Si consideri l'esempio che segue:

```
Animale mioAnimale = new Animale();
...
double peso = mioAnimale.getPeso();
```

Si supponga di poter includere entrambe le versioni di `getPeso` nella definizione della classe, al contrario di quanto si può fare in realtà. Si consideri quindi il lavoro che dovrebbe svolgere il compilatore. Sebbene non si sia considerato mai esplicitamente questo caso, bisogna sapere che si può memorizzare un valore di tipo `char` in una variabile di tipo `double`. Perciò, in questo scenario ipotetico, la variabile `peso` potrebbe ricevere sia un valore `double` sia un valore `char`.

Il compilatore quindi non avrebbe modo di capire quale versione di `getPeso` il programmatore intendesse usare.

Non si può eseguire l'*overloading* solo sulla base del tipo di ritorno

Una classe non può contenere metodi che hanno lo stesso nome e gli stessi parametri e solo valori restituiti differenti. Questi metodi, infatti, avrebbero la stessa firma e in Java non si possono avere più metodi con la stessa firma.

ESEMPIO DI PROGRAMMAZIONE
UNA CLASSE CHE RAPPRESENTA SOMME DI DENARO

Il Listato 9.13 contiene una classe, `Soldi`, i cui oggetti rappresentano somme di denaro in Euro, per esempio 9.99, 500.00 o 0.50 Euro. Di solito si pensa alle somme di denaro come a valori di tipo `double`, tuttavia sia per i programmatori che usano la classe `Soldi`, sia per gli utenti finali dei programmi che utilizzano la classe `Soldi`, questi valori non sono considerati dei valori in virgola mobile oppure interi, ma sono proprio considerati come tipi a se stanti, `Soldi` appunto. Per una persona comune, 9.99 Euro non corrispondono a un valore in virgola mobile. E non dovrebbe essere così nemmeno per il programmatore. Alle volte si possono utilizzare variabili di tipo `double` per rappresentare somme di denaro, ma questo potrebbe portare a potenziali problemi. Un valore di tipo `double` è una quantità approssimata, inadatta per un programma di contabilità. I clienti di una banca avrebbero certamente qualcosa da ridire se il loro estratto conto avesse uno scarto di qualche Euro o anche solo di pochi centesimi.

Chiaramente, per implementare la classe `Soldi` è necessario decidere come rappresentare i dati. Dato che si intende rappresentare le somma di denaro come quantità esatte, è necessario usare un tipo intero. Tuttavia, il tipo `int` non può rappresentare valori molto grandi e perciò è bene usare un tipo `long`. Una somma di denaro come 3500.36 Euro, può essere rappresentata con due interi: 3500 e 36. Tali valori dovranno essere memorizzati all'interno di variabili di istanza di tipo `long`. Sebbene abbia senso considerare somme di denaro negative, per semplificare l'esempio si considereranno solamente valori positivi.

Si noti che il metodo `set` è stato definito sfruttando l'*overloading*. I quattro metodi con nome `set` permettono al programmatore di impostare una somma di denaro nel modo che a lui risulta più comodo. Infatti, per indicare una somma in Euro, il programmatore potrebbe usare un singolo valore intero che non include centesimi, un singolo valore di tipo `double`, un altro oggetto di tipo `Soldi`, oppure una stringa come `"E9.98"` o `"9.98"`. Il programmatore non deve preoccuparsi di quali variabili di istanza vengano usate all'interno della classe `Soldi`, ma può semplicemente pensare in termini di somme di denaro.

Si osservino i dettagli della definizione dei metodi `set`. Il metodo `set` che presenta un parametro di tipo `long` è banale: assegna direttamente il valore totale di Euro. Il metodo `set` che presenta un parametro di tipo `double` opera convertendo il valore `double` in un valore che rappresenta la somma di denaro come il numero di centesimi totali presenti nella somma. Questo viene effettuato come segue:

```
long centesimiTotali = Math.round(nuovaSomma * 100);
```

Il metodo `Math.round` elimina un'eventuale parte frazionaria. Quando l'argomento passato è un `double`, come in questo caso, questo metodo restituisce un valore di tipo `long`. Gli operatori di divisione `/` e `%` convertono il numero `centesimiTotali` in Euro e centesimi di Euro.

Anche il metodo `set` che riceve come argomento un oggetto di tipo `Soldi` è molto semplice. Tuttavia, è bene notare un aspetto importante. Le variabili di istanza dell'oggetto `Soldi` sono accessibili per nome all'interno della definizione della classe `Soldi`. Nel caso in oggetto, queste variabili di istanza sono `altriSoldi.euro` e `altriSoldi.centesimi`. All'interno della definizione di una classe, infatti, si può accedere alle variabili di istanza di un qualsiasi oggetto della classe stessa. Il metodo `set` che

riceve un argomento di tipo `String` trasforma una stringa come "`E12.75`" o "`12.75`" in due numeri interi, come 12 e 75. In primo luogo, controlla se il primo carattere della stringa è il simbolo dell'Euro `E`, mediante le seguenti istruzioni:

```
if (sommaString.charAt(0) == 'E')
    sommaString = sommaString.substring(1);
```

L'invocazione al metodo `charAt` della classe `String` permette di ottenere il primo carattere di `sommaString`. Se tale carattere è '`E`', la stringa viene sostituita dalla sua sottostringa che va dalla posizione 1 alla fine della stringa rimuovendo di fatto il primo carattere. Il metodo `substring` della classe `String` effettua proprio questa operazione, come indicato nella Figura 2.5 del Capitolo 2. In considerazione del fatto che potrebbero esserci uno o più spazi bianchi tra il carattere '`E`' e la somma di denaro, viene invocato il metodo `trim` che permette di rimuovere gli eventuali spazi bianchi:

```
sommaString = sommaString.trim();
```

La stringa viene quindi separata in due sotto-stringhe: una per gli Euro e una per i centesimi. Questa operazione viene effettuata individuando il punto decimale e spezzando la stringa proprio sul punto decimale. La posizione del punto decimale è memorizzata nella variabile `posizionePunto` come segue:

```
int posizionePunto = sommaString.indexOf(".");
```

Le sottostringhe che contengono Euro e centesimi vengono quindi costruite come segue:

```
stringaEuro = sommaString.substring(0, posizionePunto);
stringaCentesimi = sommaString.substring(posizionePunto + 1);
```

Infine, le stringhe che contengono Euro e centesimi vengono convertite in valori di tipo `long` attraverso l'invocazione del metodo statico `parseLong` appartenente alla classe *wrapper* `Long` (introdotta in precedenza in questo capitolo):

```
euro = Long.parseLong(stringaEuro);
centesimi = Long.parseLong(stringaCentesimi);
```

Il metodo `moltiplica` è utilizzato per moltiplicare una somma di denaro per un intero. Il metodo `somma` è utilizzato per sommare due oggetti di tipo `Soldi`. Si supponga, per esempio, che `s1` e `s2` siano oggetti di tipo `Soldi` che rappresentano entrambi una somma pari a 2.00 Euro. L'invocazione del metodo `s1.moltiplica(3)` restituisce un oggetto della classe `Soldi` che rappresenta un ammontare di 6.00 Euro. L'invocazione del metodo `s1.somma(s2)`, invece, restituisce un oggetto della classe `Soldi` che rappresenta un ammontare di 4.00 Euro.

Per capire meglio la definizione dei metodi `moltiplica` e `somma` si ricordi che 1 Euro corrisponde a 100 centesimi di Euro. Perciò, se la variabile di istanza `prodotto.centesimi` ha un valore di 100 o superiore, la seguente istruzione assegnerà alla variabile `riporto` un valore pari al numero di Euro interi corrispondenti ai centesimi dati.

```
long riporto = prodotto.centesimi / 100;
```

Il numero di centesimi rimanenti a seguito della rimozione degli Euro interi è dato dall'istruzione:

```
prodotto.centesimi % 100
```

Il Listato 9.14 presenta un semplice programma che utilizza la classe `Soldi`.

LISTATO 9.13 La classe Soldi.

MyLab

```
import java.util.Scanner;

/**
 Classe che rappresenta una somma di denaro non negativa
*/
public class Soldi {

    private long euro;
    private long centesimi;

    public void set(long nuoviEuro) {
        if (nuoviEuro < 0) {
            System.out.println("Errore: somme negative di soldi" +
                               " non sono permesse.");
            System.exit(0);
        } else {
            euro = nuoviEuro;
            centesimi = 0;
        }
    }

    public void set(double nuovaSomma) {
        if (nuovaSomma < 0) {
            System.out.println("Errore: somme negative di soldi" +
                               " non sono permesse.");
            System.exit(0);
        } else {
            long centesimiTotali = Math.round(nuovaSomma * 100);
            euro = centesimiTotali / 100;
            centesimi = centesimiTotali % 100;
        }
    }

    public void set(Soldi altriSoldi){
        this.euro = altriSoldi.euro;
        this.centesimi = altriSoldi.centesimi;
    }

    /**
    Precondizione: L'argomento è una rappresentazione corr
    di una somma di denaro, con o senza segno.
    Frazioni di centesimi non sono ammesse.
    */
    public void set(String sommaString)    {
        String stringaEuro;
        String stringaCentesimi;
```

```
            sommaString = sommaString.substring(1);
        sommaString = sommaString.trim();

        //Trova il punto decimale:
        int posizionePunto = sommaString.indexOf(".");
        if (posizionePunto < 0) { //Se non presente
            centesimi = 0;
            euro = Long.parseLong(sommaString);
        } else { //La stringa ha un punto decimale.
            stringaEuro = sommaString.substring(0, posizionePunto);
            stringaCentesimi = sommaString.substring(posizionePunto + 1);
            //una cifra nei centesimi significa decine di Euro
            if (stringaCentesimi.length() <= 1)
                stringaCentesimi = stringaCentesimi + "0";
            // converte in formato numerico
            euro = Long.parseLong(stringaEuro);
            centesimi = Long.parseLong(stringaCentesimi);
            if ((euro < 0) || (centesimi < 0) || (centesimi > 99)) {
                System.out.println("Errore: rappresentazione illegale" +
                                   " di soldi.");
                System.exit(0);
            }
        }
    }

    public void leggiInput() {
        System.out.println("Scrivi l'ammontare su una riga:");
        Scanner tastiera = new Scanner(System.in);
        String somma = tastiera.nextLine();
        set(somma.trim());
    }

    /**
    Non va a capo dopo aver scritto la somma di denaro.
    */
    public void scriviOutput() {
        System.out.print("E" + euro);
        if (centesimi < 10)
            System.out.print(".0" + centesimi);
        else
            System.out.print("." + centesimi);
    }

    /**
    Moltiplica i soldi per il valore n fornito in input
    */
    public Soldi moltiplica(int n) {
        Soldi prodotto = new Soldi();
        prodotto.centesimi = n * centesimi;
```

È stato utilizzato `nextLine` al posto di `next` poiché potrebbe esserci uno spazio tra il simbolo dell'Euro e il numero.

```
        long riporto = prodotto.centesimi / 100;
        prodotto.centesimi = prodotto.centesimi % 100;
        prodotto.euro = n * euro + riporto;
        return prodotto;
    }

    /**
    Restituisce la somma tra l'oggetto invocante
    e l'argomento fornito in input
    */
    public Soldi somma(Soldi altriSoldi) {
        Soldi somma = new Soldi();
        somma.centesimi = this.centesimi + altriSoldi.centesimi;
        long riporto = somma.centesimi / 100;
        somma.centesimi = somma.centesimi % 100;
        somma.euro = this.euro + altriSoldi.euro + riporto;
        return somma;
    }
}
```

LISTATO 9.14 Utilizzo della classe `Soldi`.

MyLab

```
public class SoldiDemo {

    public static void main(String[] args) {
        Soldi soldiIniziali = new Soldi();
        Soldi soldiPossibili = new Soldi();

        System.out.println("Scrivi il tuo saldo corrente:");
        soldiIniziali.leggiInput();
        soldiPossibili = soldiIniziali.moltiplica(2);

        System.out.print("Se lo raddoppi avrai ");
        soldiPossibili.scriviOutput();
        System.out.println(", e, ancora meglio,");
        soldiPossibili = soldiIniziali.somma(soldiPossibili);
        System.out.println("se lo triplichi avrai");
        soldiPossibili.scriviOutput();
        System.out.println();

        System.out.println("Ricorda: un centesimo risparmiato");
        System.out.println("e' un centesimo guadagnato.");
    }
}
```

Si termina la riga poiché `scriviOutput` non termina la riga.

Esempio di output

```
Scrivi il tuo saldo corrente:
Scrivi l'ammontare su una riga:
£500.99
```

```
Se lo raddoppi avrai E1001.98 e, ancora meglio,
se lo triplichi avrai
E1502.97
Ricorda: un centesimo risparmiato
e' un centesimo guadagnato.
```

9.4 Information hiding rivisitato

Gli argomenti illustrati in questo paragrafo non sono necessari per comprendere il resto del testo. Per questo motivo, è possibile posticiparne la lettura fino al momento in cui ci si sentirà più in grado di padroneggiare le funzionalità delle classi. Questo paragrafo discute un problema subdolo che può presentarsi nel momento in cui si definiscono certi tipi di classi. Questo problema non riguarda quelle classi le cui variabili di istanza sono di tipo primitivo, come `int`, `double`, `char` e `boolean`, oppure di tipo `String`. Si possono perciò scrivere numerose classi senza incorrere in questo problema.

9.4.1 Privacy leak

Una classe può avere variabili di istanza di qualsiasi tipo, anche di tipo classe. Tuttavia, usare variabili di istanza di tipo classe può introdurre un problema che richiede particolare attenzione. Il problema si presenta poiché le variabili di tipo classe contengono l'indirizzo di memoria in cui si trova fisicamente l'oggetto. Per esempio, si supponga che `buono` e `cattivo` siano entrambe variabili della classe `Animale`, definita nel Listato 9.1. Si supponga, quindi, che alla variabile `buono` siano assegnati diversi oggetti e che il programma si trovi, infine, a dover eseguire la seguente operazione di assegnamento:

```
cattivo = buono;
```

Dopo l'esecuzione di questa istruzione, le variabili `buono` e `cattivo` referenziano lo stesso oggetto. Perciò, se si modifica `cattivo` si modifica anche `buono`.

Le seguenti righe inseriscono l'istruzione precedente in un contesto più ampio per permettere di capire meglio le implicazioni.

```
Animale buono = new Animale();
buono.set("Cane da Guardia", 5, 35);
Animale cattivo = buono;
cattivo.set("Bullo", 8, 100);
buono.scrivi();
```

Dato che `cattivo` e `buono` referenziano lo stesso oggetto, il codice precedente porterà al seguente output:

```
Nome: Bullo
Eta': 8 anni
Peso: 100.0 kg
```

La modifica apportata a `cattivo` modifica anche `buono` in quanto le variabili `buono` e `cattivo` referenziano lo stesso oggetto. La stessa cosa può succedere anche con variabili di istanza e può dar luogo ad alcuni problemi. Si consideri l'esempio che segue.

Il Listato 9.15 contiene la definizione della classe `CoppiaAnimali` che rappresenta una coppia di oggetti `Animale`. La classe possiede due variabili di istanza private di tipo `Animale`. Il programmatore che ha scritto questa classe ha erroneamente pensato che gli oggetti referenziati da queste due variabili non possano venire modificati da alcun programma che usi la classe `CoppiaAnimali`.

Infatti, le variabili di istanza sono private e non possono essere viste dall'esterno della classe. Inoltre, lo sviluppatore non ha introdotto alcun metodo *set* che cambi le istanze private della classe. Nonostante tutti questi accorgimenti, chiunque può accedere alle variabili di istanza attraverso i metodi *get* `getPrimo` e `getSecondo`. Quello appena presentato, è un errore di programmazione molto comune.

LISTATO 9.15 Una classe poco sicura. MyLab

```
public class CoppiaAnimali {

    private Animale primo, secondo;

    public CoppiaAnimali(Animale primoAnimale, Animale secondoAnimale) {
        primo = primoAnimale;
        secondo = secondoAnimale;
    }

    public Animale getPrimo() {
        return primo;
    }

    public Animale getSecondo() {
        return secondo;
    }

    public void scriviOutput() {
        System.out.println("Primo animale nella coppia:");
        primo.scriviOutput();
        System.out.println("\nSecondo animale nella coppia:");
        secondo.scriviOutput();
    }
}
```

Il programma presentato nel Listato 9.16 crea un oggetto di tipo `CoppiaAnimali` e quindi modifica lo stato dell'oggetto referenziato dalla sua variabile privata `primo`! Questo accade poiché una variabile di un tipo classe contiene un indirizzo di memoria e si può usare l'operatore di assegnamento per far sì che due variabili diverse referenzino lo stesso oggetto. Questo è quanto ha fatto il programmatore "hacker" che ha scritto il programma nel Listato 9.16. Invocando il metodo `getPrimo`, il programmatore ottiene l'indirizzo della variabile di istanza privata `primo`.

Questo indirizzo viene memorizzato nella variabile `cattivo`. La variabile `cattivo` è quindi un altro riferimento a `primo`. In questo modo, il programmatore, non potendo accedere a `primo`, aggira il problema usando la variabile `cattivo` per riferirsi a `primo`. Invocando il metodo `setAnimale` sulla variabile `cattivo`, il programmatore può modificare lo stato dell'oggetto referenziato da una variabile di istanza privata. Questo fenomeno è detto **privacy leak** (letteralmente "falla nella sicurezza").

Sebbene sembri impossibile evitare che un programmatore "hacker" acceda alle variabili di istanza private di una classe, in realtà ci sono vari modi per impedire che questo accada; alcuni più semplici e altri più complessi.

Un modo semplice per evitare questo problema consiste nell'usare solo variabili di istanza di tipo primitivo. Chiaramente questa soluzione rappresenta una limitazione molto forte. Un'altra possibilità consiste nel definire classi che non abbiano alcun metodo *set*, per esempio la classe `String`. Infatti, quando viene creato un oggetto `String`, i suoi dati privati non possono essere modificati e, di conseguenza, il programmatore "hacker" non può alterarne lo stato.

Un altro modo semplice consiste nel definire metodi *get* che restituiscono i singoli attributi delle variabili di istanza di tipo classe al posto dell'oggetto. Chiaramente, attributi che sono essi stessi oggetti potrebbero essere a loro volta affetti dallo stesso problema.

Quindi, per esempio, invece di definire in `CoppiaAnimali` un metodo *get* come `getPrimo`, si dovrebbero definire tre metodi: `getNomePrimo`, `getEtaPrimo` e `getPesoPrimo`.

Esiste anche una soluzione più complessa, che tuttavia va oltre gli obiettivi di questo testo. Tale soluzione prevede la definizione di un metodo che crea un duplicato esatto di un oggetto. Questi oggetti sono detti **cloni** (*clone*). Invece di restituire un oggetto referenziato da una variabile di istanza di una classe che potrebbe creare un *privacy leak*, può essere restituito un clone della variabile di istanza. In questo modo il programmatore potrebbe modificare il clone senza tuttavia alterare i dati privati dell'oggetto di partenza.

Questi esempi non devono dare l'impressione che definire variabili di istanza di tipo classe sia una cattiva idea. Al contrario, esse sono molto naturali e utili. Tuttavia, è necessaria attenzione nel gestirle.

yLab

LISTATO 9.16 Modificare un oggetto privato di una classe.

```
/**
  Programma giocattolo che dimostra come impiegare
  e modificare i dati privati della classe CoppiaAnimali
*/
public class Hacker {

    public static void main(String[] args) {
        Animale buono = new Animale("Cane da Guardia", 5, 75.0);
        Animale altro = new Animale("Fido", 4, 60.5);
        CoppiaAnimali coppia = new CoppiaAnimali(buono, altro);
        System.out.println("La coppia:");
        coppia.scriviOutput();

        Animale cattivo = coppia.getPrimo();
        cattivo.setAnimale("Bullo", 1200, 500);
```

```
        System.out.println("\nLa coppia ora:");
        coppia.scriviOutput();

        System.out.println("L'animale non era molto privato!");
        System.out.println("Sembra una falla di sicurezza.");
    }
}
```

Esempio di output

```
La coppia:
Primo animale nella coppia:
Nome: Cane da Guardia
Eta: 5 anni
Peso: 75.0 Kg

Secondo animale nella coppia:
Nome: Fido
Eta: 4 anni
Peso: 60.5 Kg

La coppia ora:
Primo animale nella coppia:
Nome: Bullo
Eta: 1200 anni
Peso: 500.0 Kg

Secondo animale nella coppia:
Nome: Fido
Eta: 4 anni
Peso: 60.5 Kg
L'animale non era molto privato!
Sembra una falla di sicurezza.
```

Questo programma ha modificato un oggetto referenziato da una variabile di istanza dell'oggetto `coppia`

Privacy Leak

Una variabile di istanza privata di tipo classe referenzia un oggetto che, in alcuni casi, potrebbe essere modificato dall'esterno della classe. Per evitare questo problema si può adottare una delle seguenti soluzioni.

- Dichiarare variabili di istanza che sono di tipi non modificabili (cioè, la classe che le definisce non prevede metodi *set*), come la classe `String`.
- Omettere i metodi *get* che restituiscono oggetti referenziati da una variabile di istanza. Definire, invece, metodi che restituiscono individualmente gli attributi interni di questi oggetti.
- Far sì che i metodi *get* restituiscano un clone dell'oggetto referenziato dalla variabile di istanza invece dell'oggetto stesso.

9.5 Rappresentare in UML le relazioni associative fra classi

La classe `CoppiaAnimali` definita nel Listato 9.15 possiede due variabili di istanza di tipo `Animale`. Ogni volta che all'interno delle classi vengono definite variabili di istanza di tipo classe, sussiste una **relazione associativa** (detta semplicemente **associazione**) tra le classi. Una relazione associativa realizza una relazione *has-a* (letteralmente "ha-un"). Esiste un altro tipo di relazione fra le classi che sarà discussa nel Capitolo 10. Di seguito viene descritto come esprimere questo tipo di relazione in un diagramma delle classi. Questo può essere utile in tutti i casi in cui si dispone di un diagramma delle classi e si deve procedere con la relativa codifica. Si sottolinea che, sebbene esistano modi più elaborati per codificare una relazione associativa a partire da un diagramma delle classi, in questa sede verrà presentata quella più semplice, che è poi quella adottata nel Listato 9.15.

Prima di illustrare il diagramma delle classi che rappresenta esattamente l'effettiva codifica delle classi `Animale` e `CoppiaAnimali`, occorre introdurre alcuni concetti.

Si supponga di voler definire due classi: una che rappresenta un'automobile e una che rappresenta una persona. Sia la persona sia l'automobile sono caratterizzate da attributi classici, quali per esempio nome ed età per la persona e targa e colore per l'automobile. Una persona può, però, essere anche caratterizzata dal fatto di possedere un'automobile. Quest'ultima proprietà identifica una relazione associativa tra persona e automobile.

In un diagramma delle classi, una relazione associativa fra classi si esprime tracciando una linea che collega le classi. Disegnando il diagramma delle classi a seguito delle specifiche sopra riportate e sapendo come esprimere un'associazione, si giunge al diagramma rappresentato nella Figura 9.4.

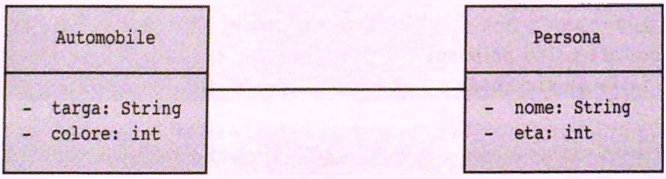


Figura 9.4 Relazione associativa fra classi.

Sfortunatamente, tale diagramma non contiene informazioni sufficienti per stabilire come debba essere codificata la relazione associativa. Si ricorda che un diagramma delle classi dovrebbe contenere informazioni sufficienti affinché la sua codifica sia immediata senza l'ausilio di una descrizione verbale. Tale diagramma lascia aperte varie questioni non specificando tre importanti proprietà: la navigabilità, la cardinalità e i ruoli.

La **navigabilità** specifica la direzione di lettura dell'associazione. In linea generale, la navigabilità può non essere specificata (come nel caso del diagramma rappresentato nella Figura 9.4), unidirezionale (da una classe verso l'altra) o bi-direzione (da entrambe le classi). Una navigabilità non specificata non fornisce alcuna informazione utile per la codifica. Una navigabilità unidirezionale permette di identificare in quale classe saranno definite variabili di istanza dell'altra classe. Infine, una navigabilità bi-direzionale specifica che le variabili di istanza saranno presenti in entrambe le classi.

In base alla specifica iniziale, una persona può possedere (*has-a*) un'automobile. Questo si traduce in una navigabilità unidirezionale dalla classe `Persona` verso la classe `Automobile`, come illustrato nella Figura 9.5. Come si può notare, la navigabilità è rappresentata da una freccia aperta posta a un estremo dell'associazione. Nel caso specifico, la freccia è rivolta verso la classe `Automobile` poiché l'associazione è navigabile nella direzione che va da `Persona` ad `Automobile`. Una persona ha-una (*has-a*) automobile. In un certo senso è come dire che `Persona` "vede" `Automobile`; effettivamente è `Persona` che avrà delle variabili di istanza di tipo `Automobile` e non viceversa.

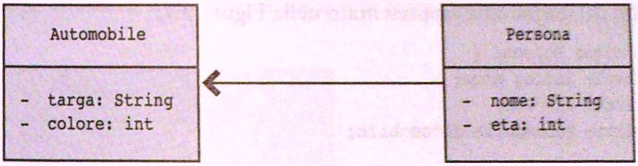


Figura 9.5 Relazione associativa fra classi navigabile.

La **cardinalità** di un'associazione permette di specificare quante istanze entrano in gioco in un'associazione. Sebbene sia possibile specificare la cardinalità a entrambi gli estremi di un'associazione, per semplicità ci si limiterà a specificarla solo all'estremo verso la classe `Automobile`. Si è scelto questo estremo poiché questa informazione è rilevante al fine di identificare quante variabili di istanza di tipo `Automobile` dovranno essere definite nella classe `Persona` dal momento che la navigabilità è da `Persona` ad `Automobile`. Poiché una persona può (ipotetico) possedere un'automobile, la cardinalità deve essere posta pari a 0..1. Questo significa che nella classe `Persona` sarà dichiara una variabile di istanza di tipo `Automobile` il cui valore può essere `null` (nel caso la persona non possieda un'automobile) o un riferimento a un oggetto di tipo `Automobile` (nel caso la persona possieda un'automobile). La Figura 9.6 arricchisce il diagramma rappresentato nella Figura 9.5 con la cardinalità.

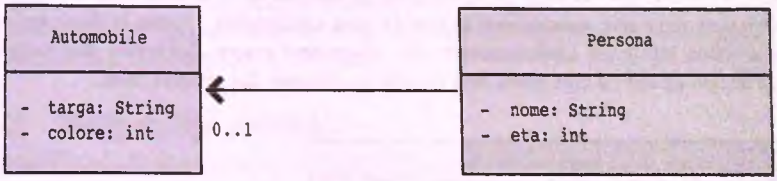


Figura 9.6 Relazione associativa fra classi navigabile con cardinalità.

Infine, il **ruolo** specifica la funzione che una classe svolge nei confronti della classe con cui è in associazione. Un ruolo è, quindi, un nome. È possibile assegnare ruoli a entrambi gli estremi di un'associazione. Per gli scopi di questo esempio, è importante specificare il ruolo che gioca la classe `Automobile` nei confronti della classe `Persona`. A tal fine, si pone all'estremo dell'associazione verso la classe `Automobile` il ruolo `automobile`. Questo vuol dire che un'automobile gioca il ruolo di "automobile" nei confronti di una

persona. Dal punto di vista implementativo, un ruolo specifica il nome che deve essere assegnato alla variabile di istanza che realizza l'associazione. Inoltre, un ruolo specifica anche la visibilità di tale variabile (il suo modificatore d'accesso). La Figura 9.7 illustra il diagramma delle classi completo dell'esempio.

Si può notare che ora all'interno della classe `Persona` compare una variabile di istanza chiamata proprio `automobile` e di tipo `Automobile`. Dal momento che l'associazione era ben dettagliata e che non lasciava alcun dubbio su come codificarla, la variabile di istanza `automobile` nella classe `Persona` si sarebbe potuta omettere.

Di seguito viene riportato un frammento della definizione della classe `Persona` come dedotto dal diagramma rappresentato nella Figura 9.7:

```
public class Persona {
    private String nome;
    private int eta;
    private Automobile automobile;
    -
}
```

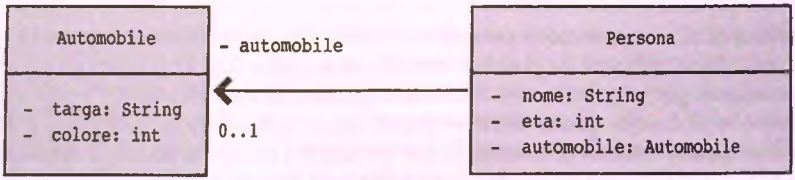


Figura 9.7 Relazione associativa fra classi navigabile, con cardinalità e ruoli.

La Figura 9.8 illustra il diagramma per le classi `Animale` (del Listato 9.1) e `CoppiaAnimali` (del Listato 9.15). Come si può notare, esiste una doppia associazione tra la classe `Animale` e la classe `CoppiaAnimale`. Entrambe le associazioni sono navigabili dalla classe `CoppiaAnimali` verso la classe `Animale`. Infatti, è all'interno della classe `CoppiaAnimali` che sono state definite delle variabili di istanza di tipo `Animale`. Il motivo per cui sono state usate due associazioni entrambe con cardinalità `1` verso la classe `Animale`, è che si voleva esprimere esplicitamente che dovevano essere dichiarate due variabili di istanza di tipo `Animale` con nomi ben precisi specificati dai relativi ruoli.

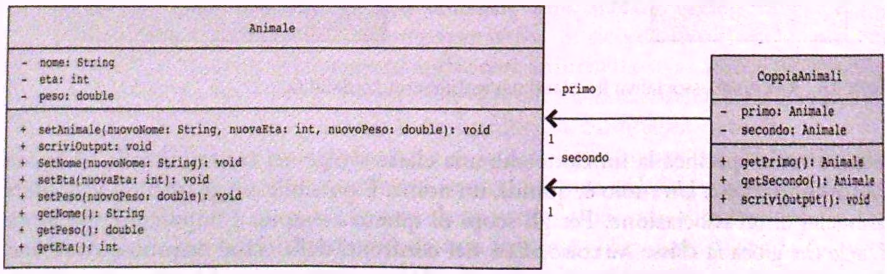


Figura 9.8 Il diagramma delle classi `Animale` e `CoppiaAnimali`.

Come si vedrà nel prossimo paragrafo, è possibile raggruppare insiemi di elementi (oggetti o meglio riferimenti a oggetti) in un array. Supponiamo di dover definire la classe `Zoo` che è in associazione con un certo numero di `Animali`. Uno zoo ospita animali e quindi possiede degli animali. In questo caso, è possibile definire una sola associazione tra `Zoo` e `Animale`, rendere la navigabilità di questa associazione da `Zoo` verso `Animale`, assegnare cardinalità pari a `1..*` verso `Animale` e assegnare il ruolo `fauna` verso la classe `Animale`. La cardinalità pari a `1..*` vuol dire che nell'associazione entra in gioco un numero di animali che va da un minimo di 1 fino a un massimo non specificato (sarà specificato a run-time). Questo diagramma si tradurrà nella definizione all'interno della classe `Zoo` di un array di elementi di tipo `Animale` il cui nome è `fauna`. Il diagramma è raffigurato in Figura 9.9.

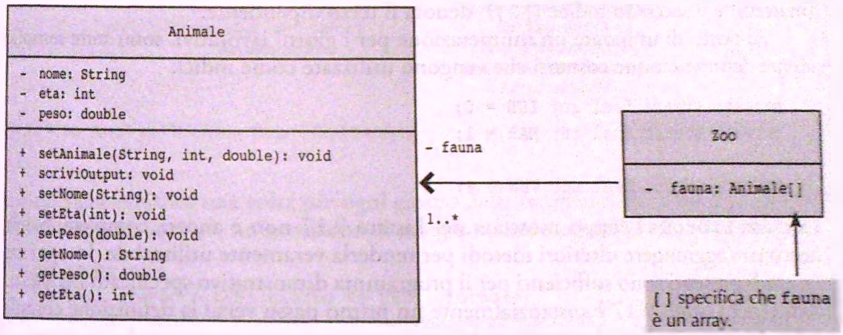


Figura 9.9 Una relazione associativa UML 1 a molti.

9.6 Array nelle definizioni di classe

Gli array possono essere utilizzati all'interno delle definizioni di classe. Il tipo base di questi array può essere un tipo qualunque e, in particolare, può essere anche una classe. Il prossimo paragrafo illustra l'utilizzo di array il cui tipo base è un tipo primitivo, mentre il paragrafo successivo illustra come un array con tipo base di tipo classe possa essere utilizzato per realizzare le associazioni tra classi.

9.6.1 Array di tipi primitivi

Sebbene gli array di tipo primitivo siano stati presentati nel Capitolo 6, in questo capitolo viene presentato un caso di studio in cui gli array sono utilizzati per definire delle variabili di istanza all'interno di una classe.

ESEMPIO DI PROGRAMMAZIONE
MEMORIZZAZIONE DELLE ORE LAVORATIVE DEI DIPENDENTI

In questo esempio viene utilizzato un array bidimensionale (chiamato `ore`) per memorizzare il numero di ore di lavoro di ogni dipendente di un'azienda per ognuno dei cinque giorni lavorativi della settimana (da lunedì a venerdì). Il primo indice dell'array è

utilizzato per rappresentare i giorni della settimana, mentre il secondo è utilizzato per rappresentare i dipendenti. L'array bidimensionale è una variabile di istanza privata nella classe `LibroDelTempo` presentato nel Listato 9.17. La classe include un metodo `main` che dimostra l'utilizzo della classe stessa applicata a una piccola azienda con tre dipendenti. I dipendenti sono numerati 1, 2 e 3, ma sono memorizzati nelle posizioni 0, 1 e 2 dell'array, dato che gli indici dell'array sono numerati partendo da 0. Di conseguenza, è necessario operare un aggiustamento pari a -1 ogni volta che si deve specificare l'indice di un dipendente nell'array. Inoltre si possono numerare i giorni assegnando 0 a lunedì, 1 a martedì e così via, in modo da evitare aggiustamenti tra i numeri e gli indici dei giorni. Per esempio, le ore lavorate dal dipendente numero 3 di martedì sono memorizzate in `ore[1][2]`. Il primo indice (`[1]`) denota il secondo giorno lavorativo della settimana (martedì) e il secondo indice (`[2]`) denota il terzo dipendente.

Al posto di utilizzare un'enumerazione per i giorni lavorativi, sono state semplicemente definite cinque costanti che vengono utilizzate come indici:

```
private static final int LUN = 0;
private static final int MAR = 1;
...
private static final int VEN = 4;
```

La classe `LibroDelTempo` mostrata nel Listato 9.17 non è ancora completa. Sarebbe necessario aggiungere ulteriori metodi per renderla veramente utilizzabile. In ogni caso, i metodi presenti sono sufficienti per il programma dimostrativo specificato nel `main`. Il codice nel Listato 9.17 è sostanzialmente un primo passo verso la definizione completa della classe in oggetto. Il metodo `setOre` è, inoltre, un prototipo (*stub*). Si ricorda che un prototipo è un metodo la cui definizione può essere utilizzata a scopo di test, ma che non è ancora nella sua versione finale. Tuttavia, `setOre` è abbastanza completo per illustrare l'utilizzo dell'array bidimensionale `ore`, il quale è una variabile di istanza della classe.

In aggiunta all'array bidimensionale `ore`, la classe `LibroDelTempo` utilizza due array monodimensionali come variabili di istanza. L'array `oreSettimana` memorizza le ore totali lavorate da ogni dipendente in una settimana. Quindi, `oreSettimana[0]` è il numero totale di ore lavorate dal dipendente 1 in una settimana, `oreSettimana[1]` è il numero totale di ore lavorate dal dipendente 2 in una settimana e così via. L'array `oreGiorno` memorizza il numero totale di ore lavorate da tutti dipendenti in ciascun giorno della settimana. Quindi, `oreGiorno[LUN]` è il numero totale di ore lavorate di lunedì da tutti i dipendenti, `oreGiorno[MAR]` è il numero totale di ore lavorate di martedì da tutti i dipendenti e così via.

I metodi `calcolaOreSettimana` e `calcolaOreGiorno` calcolano, rispettivamente, i valori per `oreSettimana` e `oreGiorno`, recuperando i valori dall'array bidimensionale `ore`. La Figura 9.10 mostra un esempio di dati inseriti nell'array `ore` e illustra la relazione tra `ore` e gli array `oreSettimana` e `oreGiorno`.

Va sicuramente notato il modo con cui il metodo `calcolaOreSettimana` utilizza gli indici dell'array bidimensionale `ore`. Un ciclo `for` è innestato all'interno di un altro ciclo `for`. Il ciclo `for` più esterno scandisce tutti i dipendenti, mentre il ciclo `for`

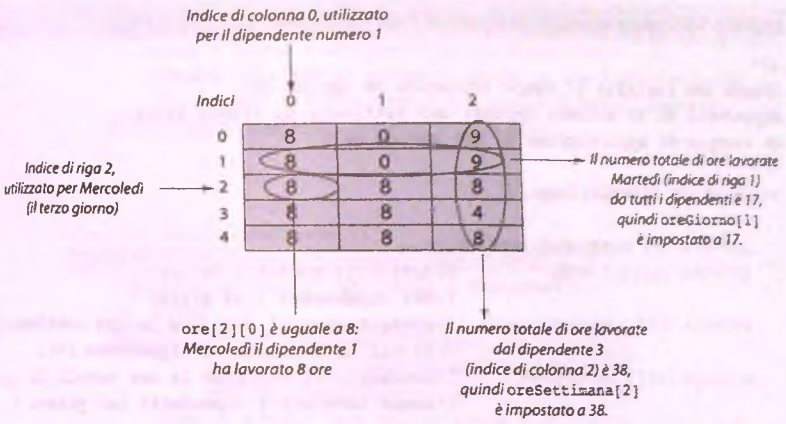


Figura 9.10 Array per la classe `LibroDelTempo`.

interno viene eseguito una volta per ogni giorno della settimana. Ecco l'aspetto del ciclo for più interno (unito a un'istruzione di assegnamento eseguita prima e dopo):

```
int somma = 0;
for (int giorno = LUN; giorno <= VEN; giorno++)
    somma = somma + ore[giorno][numeroDipendente - 1];
    //somma contiene la somma di tutte le ore lavorate in una
    //settimana dal dipendente con numero numeroDipendente.
oreSettimana[numeroDipendente - 1] = somma;
```

Si noti che, quando viene calcolata la somma delle ore per un dipendente, il secondo indice, che rappresenta un particolare dipendente, viene mantenuto costante.

Il metodo `calcolaOreGiorno` funziona in modo analogo per calcolare il numero totale di ore lavorate da tutti i dipendenti in ogni giorno della settimana. Tuttavia, in questo caso il ciclo `for` più interno scandisce il secondo indice, mentre il primo indice viene mantenuto costante. In altre parole, il ruolo dell'indice del dipendente e del giorno della settimana sono invertiti.

Sebbene la classe `LibroDelTempo` sia scritta correttamente, non è ancora un pezzo finito di un software pronto per essere usato più volte. Il metodo `setOre` è solo un prototipo e necessita di essere sostituito da un metodo più generico che ottiene le ore tramite, per esempio, l'input inserito da un utente. Il metodo `visualizzaTabella` non sempre produrrà un output chiaro come quello mostrato nel Listato 9.17, a meno che le ore abbiano tutte lo stesso numero di cifre. Quindi, il metodo `visualizzaTabella` deve essere raffinato, così che possa visualizzare sempre in maniera chiara ogni combinazione di ore lavorate. Infine, la classe `LibroDelTempo` dovrebbe avere più metodi, così che possa essere utile in un'ampia gamma di situazioni. Il Progetto 14 chiede di completare la definizione della classe `LibroDelTempo` in tutti questi modi.

MyLab

LISTATO 9.17 Programma che memorizza il tempo lavorato.

```
/**
Classe che registra il tempo impiegato da ognuno dei
dipendenti di un'azienda durante una settimana di cinque giorni.
Un esempio di applicazione e' nel metodo main.
*/
public class LibroDelTempo {

    private int numeroDiDipendenti;
    private int[][] ore;          //ore[i][j] contiene le ore
                                  //del dipendente j al giorno i.
    private int[] oreSettimana;   //oreSettimana[i] contiene le ore settimanali
                                  //in cui ha lavorato il dipendente i+1.
    private int[] oreGiorno;      //oreGiorno[i] contiene le ore totali in cui
                                  //hanno lavorato i dipendenti nel giorno i.

    private static final int NUMERO_GIORNI_LAVORATIVI = 5;
    private static final int LUN = 0;
    private static final int MAR = 1;
    private static final int MER = 2;
    private static final int GIO = 3;
    private static final int VEN = 4;

    public LibroDelTempo(int ilNumeroDiDipendenti) {
        numeroDiDipendenti = ilNumeroDiDipendenti;
        ore = new int[NUMERO_GIORNI_LAVORATIVI][numeroDiDipendenti];
        oreSettimana = new int[numeroDiDipendenti];
        oreGiorno = new int[NUMERO_GIORNI_LAVORATIVI];
    }

    public void setOre() { //Questo e' un prototipo (stub).
        ore[0][0] = 8; ore[0][1] = 0; ore[0][2] = 9;
        ore[1][0] = 8; ore[1][1] = 0; ore[1][2] = 9;
        ore[2][0] = 8; ore[2][1] = 8; ore[2][2] = 8;
        ore[3][0] = 8; ore[3][1] = 8; ore[3][2] = 4;
        ore[4][0] = 8; ore[4][1] = 8; ore[4][2] = 8;
    }

    public void aggiorna() {
        calcolaOreSettimana();
        calcolaOreGiorno();
    }

    private void calcolaOreSettimana() {
        for (int numeroDipendente = 1; numeroDipendente <= numeroDiDipendenti;
                numeroDipendente ++) {//Elabora un dipendente:
            int somma = 0;
```

Il programma completo, dovrebbe sostituire `setOre` con un metodo più completo che ottiene direttamente dall'utente i valori delle ore lavorate.

```
        for (int giorno = LUN; giorno <= VEN; giorno++)
            somma = somma + ore[giorno][numeroDipendente - 1];
            //somma contiene la somma di tutte le ore lavorate in una
            //settimana dal dipendente con numero numeroDipendente.
        oreSettimana[numeroDipendente - 1] = somma;
    }
}

private void calcolaOreGiorno () {
    for (int giorno = LUN; giorno <= VEN; giorno++) {
        //Elabora un giorno (per tutti i dipendenti):
        int somma = 0;
        for (int numeroDipendente = 1;
                 numeroDipendente <= numeroDiDipendenti;
                 numeroDipendente++)
            somma = somma + ore[giorno][numeroDipendente - 1];
            //somma contiene la somma di tutte le ore lavorate da
            //tutti i dipendenti in un giorno.
        oreGiorno[giorno] = somma;
    }
}

public void visualizzaTabella() {
    // intestazione
    System.out.print("Dipendente ");

    for (int numeroDipendente = 1;
             numeroDipendente <= numeroDiDipendenti;
             numeroDipendente++)
        System.out.print(numeroDipendente + " ");
    System.out.println("Totali");
    System.out.println();

    // valori riga
    for (int giorno = LUN; giorno <= VEN; giorno++) {
        System.out.print(getNomeGiorno(giorno) + " ");
        for (int colonna = 0; colonna < ore[giorno].length; colonna++)
            System.out.print(ore[giorno][colonna] + " ");
        System.out.println(oreGiorno[giorno]);
    }

    System.out.println();
    System.out.print("Totale = ");
    for (int colonna = 0; colonna < numeroDiDipendenti; colonna++)
        System.out.print(oreSettimana[colonna] + " ");
    System.out.println();
}
```

Il metodo `visualizzaTabella` dovrebbe essere reso più solido. Si rimanda al Progetto 14.

```
    //Converte 0 in "Lunedi'", 1 in "Martedi'" e così via.
    //Gli spazi sono inseriti per avere tutte le stringhe della stessa lunghezza.
    private String getNomeGiorno(int giorno) {
        String nomeGiorno = null;

        switch (giorno) {
            case LUN:
                nomeGiorno = "Lunedi'    ";
                break;
            case MAR:
                nomeGiorno = "Martedi'   ";
                break;
            case MER:
                nomeGiorno = "Mercoledi'";
                break;
            case GIO:
                nomeGiorno = "Giovedi'   ";
                break;
            case VEN:
                nomeGiorno = "Venerdi'   ";
                break;
            default:
                System.out.println("Errore Fatale.");
                System.exit(0);
                break;
        }

        return nomeGiorno;
    }

    /**
    Legge le ore lavorate da ogni dipendente in ogni giorno della
    settimana lavorativa nell'array bidimensionale delle ore. Il metodo
    per l'input e' solo un prototipo (stub) in questa versione preliminare.
    Computa il totale delle ore settimanali per ogni dipendente e
    il totale delle ore giornaliere per tutti i dipendenti.
    */
    public static void main(String[] args) {
        final int NUMERO_DI_DIPENDENTI = 3;
        LibroDelTempo libro = new LibroDelTempo(NUMERO_DI_DIPENDENTI);
        libro.setOre();
        libro.aggiorna();
        libro.visualizzaTabella();
    }
}
```

Una classe possiede generalmente più metodi. Qui sono stati definiti solo quelli utilizzati nel metodo `main`.

Esempio di output

```
Dipendente 1  2  3  Totali

Lunedi'    8  0  9  17
Martedi'   8  0  9  17
Mercoledi' 8  8  8  24
Giovedi'   8  8  4  20
Venerdi'   8  8  8  24

Totale =   40 24 38
```

9.6.2 Array di riferimenti

Come anticipato, il tipo base di un array può essere un tipo qualunque e, in particolare, può essere anche una classe. La seguente istruzione crea un array di nome `esemplare` i cui elementi sono riferimenti (*reference*) a oggetti di tipo `Specie`, dove `Specie` è una classe:

```
Specie[] esemplare = new Specie[3];
```

Questo array è una collezione di tre variabili (`esemplare[0]`, `esemplare[1]` ed `esemplare[2]`), tutte di tipo `Specie`. Così come gli elementi di un array il cui tipo base è un tipo primitivo sono inizializzati a valori di default, anche gli elementi di un array il cui tipo base è un tipo classe sono inizializzati al valore di default che è la costante `null`.

Il caso di studio che segue mostra un esempio di utilizzo di array di riferimenti per realizzare le relazioni associative tra classi.

CASO DI STUDIO
REPORT DELLE VENDITE

In questo caso di studio si scriverà un programma per la creazione dei report delle vendite di un'azienda. La compagnia vuole sapere quali dei suoi venditori garantiscono più vendite e paragonare le vendite di ciascun venditore rispetto al valore medio delle vendite.

Poiché è necessario registrare nome e vendite di ogni venditore, si definisce una classe che rappresenta questi dati per un singolo venditore. La classe ha metodi per l'input e l'output e altri metodi per l'accesso e la modifica delle variabili di istanza. La definizione della classe è riportata nel Listato 9.18.

LISTATO 9.18 Classe che rappresenta un venditore.

MyLab

```
import java.util.Scanner;

/**
Classe che rappresenta un venditore.
*/
public class Venditore {
    private String nome;
    private double vendite;
```

```
    public Venditore() {
        nome = "Nessun nome";
        vendite = 0;
    }

    public Venditore(String nomeIniziale, double venditeIniziali) {
        setValori(nomeIniziale, venditeIniziali);
    }

    public void setValori(String nuovoNome, double nuoveVendite) {
        nome = nuovoNome;
        vendite = nuoveVendite;
    }

    public void leggiValoriDaTastiera() {
        System.out.print("Inserire il nome: ");
        Scanner tastiera = new Scanner(System.in);
        nome = tastiera.nextLine();
        System.out.print("Inserire le vendite: € ");
        vendite = tastiera.nextDouble();
    }

    public void scriviOutput() {
        System.out.println("Nome: " + nome);
        System.out.println("Vendite: € " + vendite);
    }

    public String getNome() {
        return nome;
    }

    public double getVendite() {
        return vendite;
    }
}
```

Il programma necessita di un array per registrare i dati relativi ai venditori e di due variabili che tengano traccia, rispettivamente, della vendita più alta e della vendita media. È quindi necessario definire una nuova classe che conterrà le seguenti variabili di istanza:

```
private double venditaPiuAlta;
private double mediaVendite;
private Venditore[] team;
```

È inoltre necessario conoscere il numero di venditori. Questo valore è memorizzato in `team.length`, ma è preferibile creare una variabile con un nome significativo dedicata a memorizzare il numero di venditori. Si introduce, quindi, la seguente variabile di istanza:

```
private int numeroDiVenditori; //Equivalente a team.length
```

I compiti del programma possono essere riassunti nelle seguenti fasi principali.

1. Preparazione.
2. Recupero dei dati.
3. Calcolo di alcune statistiche (aggiornamento delle variabili di istanza).
4. Visualizzazione dei risultati.

Si deve ora fornire un nome alla classe e ai suoi metodi. Utilizzando UML, si può identificare ciò che è necessario ai fini del programma. Il diagramma delle classi risultante è mostrato nella Figura 9.11 in cui la relazione associativa tra le classi si traduce nel definire un array di elementi di tipo `Venditore` all'interno della classe `ReportVendite` il cui nome è `team`. La classe è quindi simile a quella riportata di seguito:

```
public class ReportVendite {
    private double venditaPiuAlta;
    private double mediaVendite;
    private Venditore[] team;
    private int numeroDiVenditori; //Uguale a team.length

    <Qui si deve aggiungere altro codice>

    public static void main(String[] args){
        // 1) Preparazione
        ReportVendite report = new ReportVendite();
        // 2) Recupero dei dati
        report.recuperaDati();
        // 3) Calcolo di alcune statistiche
        report.calcolaStatistiche();
        // 4) Visualizzazione dei risultati
        report.visualizzaRisultati();
    }
}
```

Si devono adesso definire i tre metodi `recuperaDati`, `calcolaStatistiche` e `visualizzaRisultati` ed eseguire il collaudo e il debugging del programma. Si inizia con la definizione dei tre metodi.

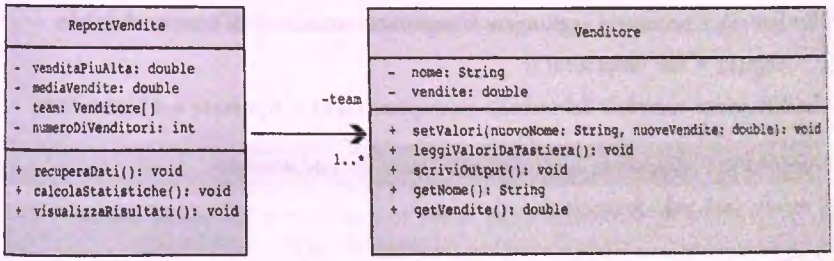


Figura 9.11 Diagramma delle classi per il caso di studio 'report delle vendite'.

Il metodo `recuperaDati` è piuttosto immediato, considerando anche la presenza di un metodo per l'inserimento dell'input per gli oggetti della classe `Venditore`. Una volta letto il numero di venditori, è possibile scrivere il seguente semplice ciclo per l'inserimento dei dati di input:

```
for (int i = 1; i <= numeroDiVenditori; i++) {
    System.out.println("Inserire i dati per il venditore " + i);
    team[i].leggiValoriDaTastiera();
}
```

Sebbene il primo indice di un array sia 0 e i venditori vengano numerati a partire da 1, si è deciso di utilizzare `team[i]` per il venditore `i`. In altre parole, si ignora l'elemento `team[0]`. Si deve quindi allocare memoria per un elemento aggiuntivo, come mostrato nel seguente codice:

```
team = new Venditore[numeroDiVenditori + 1];
```

Leggendo `numeroDiVenditori` nel metodo `recuperaDati`, la precedente istruzione verrà posta all'interno del medesimo metodo.

Rimane ancora un problema da gestire. Quando si collauda il precedente ciclo, si ottiene un messaggio di errore che attesta la presenza di un puntatore nullo (*null pointer*). Questo problema si presenta in quanto il tipo base dell'array `team` è una classe. Per capire meglio il problema, si consideri prima il seguente esempio:

```
Venditore s;
s.leggiValoriDaTastiera();
```

Questo codice produce nuovamente un errore del tipo puntatore nullo. Il problema è che la variabile `s` è semplicemente un nome: non si riferisce ad alcun oggetto della classe `Venditore`. Il codice precedente ha omesso l'utilizzo della parola chiave `new`. Il codice corretto è il seguente:

```
Venditore s = new Venditore();
s.leggiValoriDaTastiera();
```

La variabile indicizzata `team[i]` è anch'essa una variabile di tipo classe e, di conseguenza, è soltanto un nome. È necessario, quindi, assegnarle un oggetto di tipo `Venditore` prima dell'esecuzione di:

```
team[i].leggiValoriDaTastiera();
```

Per fare ciò, è necessario aggiungere la seguente istruzione all'interno del ciclo:

```
team[i] = new Venditore();
```

La definizione completa del metodo `recuperaDati` è riportata nel Listato 9.19.

LISTATO 9.19 Programma per la produzione del report delle vendite.

Il metodo `main` è alla fine della classe.

```
import java.util.Scanner;

/**
Programma per la produzione del report delle vendite.
*/
public class ReportVendite {
```

```
private double venditaPiuAlta;
private double mediaVendite;
private Venditore[] team;        //L'oggetto array e' creato in recuperaDati.
private int numeroDiVenditori; //Equivalente a team.length

/**
Lettura del numero di venditori e dei relativi dati.
*/
public void recuperaDati() {
    Scanner tastiera = new Scanner(System.in);
    System.out.println("Inserire il numero di venditori:");
    numeroDiVenditori = tastiera.nextInt();
    team = new Venditore[numeroDiVenditori + 1];
    for (int i = 1; i <= numeroDiVenditori; i++) {
        team[i] = new Venditore();
        System.out.println("Inserire i dati per il venditore " + i);
        team[i].leggiValoriDaTastiera();
        System.out.println();
    }
}

/**
Calcolo della vendita piu' alta e della vendita media.
Precondizione: deve esistere almeno un venditore.
*/
public void calcolaStatistiche() {
    double venditaSuccessiva = team[1].getVendite();
    venditaPiuAlta = venditaSuccessiva;
    double somma = venditaSuccessiva;
    for (int i = 2; i <= numeroDiVenditori; i++) {
        venditaSuccessiva = team[i].getVendite();
        somma = somma + venditaSuccessiva;
        if (venditaSuccessiva > venditaPiuAlta)
            venditaPiuAlta = venditaSuccessiva; //vendita piu' alta finora
    }
    mediaVendite = somma / numeroDiVenditori;
}

/**
Visualizza il report su schermo.
*/
public void visualizzaRisultati() {
    System.out.println("La vendita media per venditore e' € " +
                       mediaVendite);
    System.out.println("La vendita piu' alta e' pari a € " +
                    venditaPiuAlta);
    System.out.println();
    System.out.println("Il seguente venditore ha le vendite maggiori:");
```

L'array viene creato qui.

Gli oggetti di tipo `Venditore` vengono creati qui.

`team[1]` già elaborato, quindi il ciclo inizia da `team[2]`

```
        for (int i = 1; i <= numeroDiVenditori; i++) {
            double venditaSuccessiva = team[i].getVendite();

            if (venditaSuccessiva == venditaPiuAlta) {
                team[i].scriviOutput();
                System.out.println("€" + (venditaSuccessiva - mediaVendite)
                                   + " sopra la media.");
                System.out.println();
            }
        }

        System.out.println("Le performance dei restanti sono le seguenti:");

        for (int i = 1; i <= numeroDiVenditori; i++) {
            double venditaSuccessiva = team[i].getVendite();

            if (team[i].getVendite() != venditaPiuAlta) {
                team[i].scriviOutput();

                if (venditaSuccessiva >= mediaVendite)
                    System.out.println("€ " + (venditaSuccessiva - mediaVendite)
                                       + " sopra la media.");
                else
                    System.out.println("€ " + (mediaVendite - venditaSuccessiva)
                                       + " sotto la media.");
                System.out.println();
            }
        }
    }

    public static void main(String[] args) {
        // 1) Preparazione
        ReportVendite report = new ReportVendite();
        // 2) Recupero dei dati
        report.recuperaDati();
        // 3) Calcolo di alcune statistiche
        report.calcolaStatistiche();
        // 4) Visualizzazione dei risultati
        report.visualizzaRisultati();
    }
}
```

Esempio di output

```
Inserire il numero di venditori:
3
Inserire dati per il venditore 1
Inserire il nome: Mario Rossi
Inserire le vendite: € 36000
```

```
Inserire dati per il venditore 2
Inserire il nome: Marco Verdi
Inserire le vendite: € 50000

Inserire dati per il venditore 3
Inserire il nome: Maria Bianchi
Inserire le vendite: € 10000

La vendita media per venditore e' € 32000.0
La vendita piu' alta e' pari a € 50000.0

Il seguente venditore ha le vendite maggiori:
Nome: Marco Verdi
Vendite: € 50000.0
€ 18000.0 sopra la media.

Le performance dei restanti sono le seguenti:
Nome: Mario Rossi
Vendite: € 36000.0
€ 4000.0 sopra la media.

Nome: Maria Bianchi
Vendite: € 10000.0
€ 22000.0 sotto la media.
```

Ora si consideri il metodo `calcolaStatistiche` che può essere implementato tramite il seguente codice:

```
for (int i = 1; i <= numeroDiVenditori; i++) {
    somma = somma + team[i].getVendite();
    if (team[i].getVendite() > venditaPiuAlta)
        venditaPiuAlta = team[i].getVendite(); //Vendita più alta finora
}
mediaVendite = somma / numeroDiVenditori;
```

Il ciclo è fondamentalmente corretto, ma si devono inizializzare le variabili `somma` e `venditaPiuAlta` prima dell'inizio del ciclo. È possibile inizializzare `somma` con il valore 0, ma quale valore usare per inizializzare la variabile `venditaPiuAlta`? Si potrebbe usare un valore negativo, dato che le vendite non possono assumere valori negativi. In realtà, è anche possibile che le vendite siano negative. Per esempio, merci restituite possono rappresentare introiti negativi e, in tal caso, le vendite possono assumere valori negativi. Partendo dalla considerazione che esiste almeno un venditore all'interno della compagnia, è possibile inizializzare sia `somma` sia `venditaPiuAlta` con i dati del primo venditore. Il codice risultante è il seguente:

```
venditaPiuAlta = team[1].getVendite();
double somma = team[1].getVendite();
for (int i = 2; i <= numeroDiVenditori; i++) {
    somma = somma + team[i].getVendite();
```

```
        if (team[i].getVendite() > venditaPiuAlta)
            venditaPiuAlta = team[i].getVendite();  //Vendita più alta finora
    }
    mediaVendite = somma / numeroDiVenditori;
```

Il ciclo presentato funziona correttamente, ma si noti la ripetizione di alcune computazioni. Ci sono tre invocazioni identiche di `team[i].getVendite()`. Per evitare questa duplicazione, è possibile memorizzare il risultato dell'invocazione del metodo in una variabile, come mostrato di seguito:

```
double venditaSuccessiva = team[1].getVendite();
venditaPiuAlta = venditaSuccessiva;
double somma = venditaSuccessiva;
for (int i = 2; i <= numeroDiVenditori; i++) {
    venditaSuccessiva = team[i].getVendite();
    somma = somma + venditaSuccessiva;
    if (venditaSuccessiva > venditaPiuAlta)
        venditaPiuAlta = venditaSuccessiva;   //Vendita più alta finora
}
mediaVendite = somma / numeroDiVenditori;
```

La definizione completa del metodo `calcolaStatistiche` è riportata nel Listato 9.19. La definizione del restante metodo, `visualizzaRisultati`, utilizza concetti già visti e, per questo motivo, si rimanda direttamente alla sua definizione riportata nel Listato 9.19.

MyLab

deo 9.3
are array
i metodi

privacy leak con gli array

Una variabile di istanza, privata, di tipo array, può essere modificata al di fuori della classe di appartenenza se un metodo pubblico della classe restituisce tale array. Questo problema è lo stesso discusso precedentemente in questo capitolo e la soluzione è del tutto analoga.

ESEMPIO DI PROGRAMMAZIONE
UNA CLASSE SPECIALIZZATA PER LISTE

Un modo di utilizzare un array per uno specifico scopo è quello di rendere tale array una variabile di istanza di una classe. All'array si accede, quindi, solo tramite i metodi della classe che possono fornire controlli e processi automatici a piacere. Questo permette di definire classi i cui oggetti sono qualcosa di simile ad array, ma con uno scopo preciso (*special-purpose*). In questo esempio, verrà presentata una dimostrazione di classe di questo tipo.

Sarà definita una classe i cui oggetti potranno essere utilizzati per memorizzare liste di elementi, come per esempio una lista dei prodotti da acquistare o una lista di attività da svolgere. La classe avrà il nome di `ListaSenzaRipetizioni`.

La classe `ListaSenzaRipetizioni` avrà un metodo per aggiungere gli elementi alla lista. Un elemento della lista è una stringa, come per esempio "Comprare il latte". Questa classe non ha metodi per cambiare un singolo elemento o cancellare un elemento dalla lista. Ha, però, un metodo che consente di cancellare l'intera lista e cominciare con una lista vuota. Ogni oggetto della classe `ListaSenzaRipetizioni` può contenere un numero massimo di elementi. In ogni istante la lista può contenere un numero di elementi compreso tra 0 e questo numero massimo.

Un oggetto della classe `ListaSenzaRipetizioni` ha un array di stringhe come variabile di istanza. Questo array contiene gli elementi della lista. Tuttavia non si accede direttamente all'array: si utilizzano i metodi *set* e *get*. È possibile utilizzare variabili `int` per memorizzare i numeri che indicano le posizioni all'interno della lista. Ognuna di queste variabili `int` corrisponde a un indice, ma le posizioni sono numerate partendo da 1 anzichè da 0. Per esempio, un metodo chiamato `getElementoIn` consente di restituire un elemento in una data posizione. Se `listaDiCoseDaFare` è un oggetto della classe `ListaSenzaRipetizioni`, la seguente istruzione assegna alla variabile stringa `successivo` l'elemento in seconda posizione:

```
String successivo = listaDiCoseDaFare.getElementoIn(2);
```

Altri metodi permettono di aggiungere elementi alla fine della lista o di cancellare l'intera lista. Queste sono le uniche modifiche consentite alla lista. Non è pertanto possibile modificare o cancellare un particolare elemento nella lista, né aggiungere un elemento in una posizione diversa dall'ultima.

Nel Capitolo 8 è stato discusso l'incapsulamento. La classe `ListaSenzaRipetizioni` è un buon esempio di una classe ben incapsulata dal momento che nasconde i propri dettagli al programmatore che la impiega. Chiaramente il programmatore deve però sapere come utilizzare la classe. Quindi, è sensato fornire una descrizione del suo funzionamento prima di illustrare la sua definizione.

Il Listato 9.20 contiene un programma che mostra come utilizzare alcuni dei metodi definiti nella classe `ListaSenzaRipetizioni`. È da sottolineare che il costruttore accetta come argomento un intero. Quest'ultimo specifica il massimo numero di elementi che possono essere inseriti nella lista. Nell'esempio è stato utilizzato un numero piccolo.

LISTATO 9.20 Utilizzo della classe `ListaSenzaRipetizioni`.

MyLab

```
import java.util.Scanner;

public class ListaDemo {

    public static final int DIMENSIONE_MAX = 3; //Supponendo > 0

    public static void main(String[] args) {
        ListaSenzaRipetizioni listaDiCoseDaFare =
            new ListaSenzaRipetizioni(DIMENSIONE_MAX);
        System.out.println("Inserire gli elementi per la lista " +
                           "quando richiesto.");
        boolean altriElementi = true;
        String successivo = null;
```

```
        Scanner tastiera = new Scanner(System.in);

        while (altriElementi && !listaDiCoseDaFare.piena()) {
            System.out.println("Inserire un elemento:");
            successivo = tastiera.nextLine();
            listaDiCoseDaFare.aggiungiElemento(successivo);

            if (listaDiCoseDaFare.piena()) {
                System.out.println("La lista e' piena.");
            } else {
                System.out.print("Altri elementi per la lista? ");
                String risposta = tastiera.nextLine();
                if (risposta.trim().equalsIgnoreCase("no"))
                    altriElementi = false; //Utente indica
                                           //nessun altro elemento
            }
        }
        System.out.println("La lista contiene:");
        int posizione = listaDiCoseDaFare.POSIZIONE_INIZIALE;
        successivo = listaDiCoseDaFare.getElementoIn(posizione);

        while (successivo != null) {  //null indica la fine della lista
            System.out.println(successivo);
            posizione++;
            successivo = listaDiCoseDaFare.getElementoIn(posizione);
        }
    }
}
```

Esempio di output

```
Inserire gli elementi per la lista quando richiesto.
Inserire un elemento:
Comprare il latte
Altri elementi per la lista? si
Inserire un elemento:
Portare il cane a passeggio
Altri elementi per la lista? si
Inserire un elemento:
Comprare il latte
Altri elementi per la lista? si
Inserire un elemento:
Scrivere un programma
La lista e' piena.
La lista contiene:
Comprare il latte
Portare il cane a passeggio
Scrivere un programma
```

Il metodo `aggiungiElemento` aggiunge una stringa alla fine della lista. Per esempio, la seguente istruzione aggiunge la stringa contenuta nella variabile `successivo` in fondo alla lista `listaDiCoseDaFare`:

```
listaDiCoseDaFare.aggiungiElemento(successivo);
```

Se si osserva l'output mostrato nel Listato 9.20, è possibile notare che, sebbene `"Comprare il latte"` sia stato aggiunto due volte, compare una sola volta nella lista. Se l'elemento che si tenta di inserire è già presente nella lista, il metodo `aggiungiElemento` non ha effetto. In questo modo si evitano le duplicazioni.

È possibile utilizzare una variabile `int` per scorrere la lista dall'inizio alla fine. Tale tecnica è illustrata alla fine del Listato 9.20. Si inizia assegnando a una variabile `int` la prima posizione della lista con l'istruzione che segue:

```
int posizione = listaDiCoseDaFare.POSIZIONE_INIZIALE;
```

La costante `listaDiCoseDaFare.POSIZIONE_INIZIALE` è semplicemente un sinonimo di 1. Si utilizza questa costante perché il suo nome rende l'idea che ci si sta riferendo alla prima posizione della lista. Il numero 1 è molto meno espressivo. È possibile invocare il metodo `getElementoIn` per ottenere l'elemento a una specifica posizione della lista. Per esempio, la seguente istruzione assegna alla variabile stringa `successivo` lo stesso valore contenuto dalla stringa presente alla posizione data dalla variabile `posizione`:

```
successivo = listaDiCoseDaFare.getElementoIn(posizione);
```

Per ottenere l'elemento successivo nella lista, il programma incrementa semplicemente il valore della variabile `posizione`. Il seguente codice, estratto dal Listato 9.20, illustra come scorrere gli elementi della lista:

```
int posizione = listaDiCoseDaFare.POSIZIONE_INIZIALE;
successivo = listaDiCoseDaFare.getElementoIn(posizione);
while (successivo != null) {
    System.out.println(successivo);
    posizione++;
    successivo = listaDiCoseDaFare.getElementoIn(posizione);
}
```

Quando il valore di `posizione` viene incrementato oltre l'ultima posizione della lista che contiene degli elementi, occorre fermarsi, poiché non ci sono più elementi. Occorre quindi esprimere il raggiungimento della fine della lista. Se si procedesse, si rischierebbe di accedere a un valore "senza senso" (*garbage*) in una porzione inutilizzata dell'array. Per ovviare a questo problema, quando non ci sono elementi in una data posizione il metodo `getElementoIn` restituisce il valore `null`. Si noti che `null` è diverso da qualsiasi stringa e quindi non può apparire in nessuna lista. Di conseguenza, il programma può verificare la fine della lista ricercando il valore `null`. Si ricordi che per verificare l'uguaglianza o la disuguaglianza con `null` è possibile utilizzare `==` o `!=`. Non si utilizza il metodo `equals`.

La definizione completa della classe `ListaSenzaRipetizioni` è fornita nel Listato 9.21. Gli elementi nella lista sono memorizzati nella variabile di istanza `elemento`, un array di tipo base `String`. Quindi, il numero massimo di elementi che la lista può memorizzare è `elemento.length`. Tuttavia, la lista potrebbe non essere sempre piena, ma potrebbe contenere un numero di elementi minore di `elemento.length`.

Per tenere traccia di quanto l'array `elemento` è realmente utilizzato, la classe ha una variabile di istanza chiamata `numeroElementi`. Gli elementi veri e propri sono memorizzati nelle variabili indicizzate `elemento[0]`, `elemento[1]`, `elemento[2]` e così via, fino a `elemento[numeroElementi - 1]`. I valori degli elementi i cui indici sono `numeroElementi` o superiori, sono valori "senza senso" e non fanno parte della lista. Quindi, quando si scandiscono gli elementi della lista, bisogna fermarsi dopo `elementi[numeroElementi - 1]`.

Per esempio, la definizione del metodo `nellaLista` contiene un ciclo `while` che scandisce tutto l'array, verificando se l'argomento è uguale a qualche elemento della lista:

```
while (!trovato && (i < numeroElementi)) {
    if (elementoDaRicercare.equalsIgnoreCase(elemento[i]))
        trovato = true;
    else
        i++;
}
```

Il codice controlla solo gli elementi dell'array i cui indici sono minori di `numeroElementi`. Non controlla, infatti, il resto dell'array, poiché non vi sono elementi.

La classe completa `ListaSenzaRipetizioni` ha anche altri metodi che sono utilizzati dal programma nel Listato 9.20. Questi metodi aggiuntivi rendono la classe utilizzabile per una maggior varietà di applicazioni.

MyLab

LISTATO 9.21 Uso di un array all'interno di una classe per rappresentare una lista.

```
/**
Un oggetto di questa classe e' un caso speciale di lista di stringhe.
E' possibile creare la lista solo dall'inizio alla fine. E' possibile aggiungere
elementi solo alla fine della lista. Non e' possibile cambiare singoli elementi,
ma e' possibile cancellare l'intera lista e ricominciare. Nessun elemento può
comparire piu' di una volta nella lista. E' possibile utilizzare delle
variabili intere come indicatori della posizione nella lista. Gli indicatori
della posizione sono simili agli indici dell'array, ma sono numerati da 1.
*/
public class ListaSenzaRipetizioni {

    public static int POSIZIONE_INIZIALE = 1;
    public static int DIMENSIONE_DEFAULT = 50;

    //elemento.length e' il numero totale di elementi inseribili nella lista
    //(la sua capacita');
    //numeroElementi e' il numero di elementi attualmente nella lista
    private int numeroElementi; //puo' essere minore di elemento.length.
    private String[] elemento;

    /**
    Crea una lista vuota di una data capacita'.
    */
    public ListaSenzaRipetizioni(int massimoNumeroDiElementi) {
        elemento = new String[massimoNumeroDiElementi];
```

```
    numeroElementi = 0;
}

/**
Crea una lista vuota con capacita' DIMENSIONE_DEFAULT.
*/
public ListaSenzaRipetizioni() {
    elemento = new String[DIMENSIONE_DEFAULT];
    numeroElementi = 0;
    // e' possibile sostituire queste due istruzioni con
    // this(DIMENSIONE_DEFAULT);
}

public boolean piena() {
    return numeroElementi == elemento.length;
}

public boolean vuota() {
    return numeroElementi == 0;
}

/**
Precondizione: la lista non e' piena
Postcondizione: se un elemento non e' nella lista
deve essere aggiunto alla lista
*/
public void aggiungiElemento(String nuovoElemento) {
    if (!nellaLista(nuovoElemento)) {
        if (numeroElementi == elemento.length) {
            System.out.println("La lista e' piena!");
            System.exit(0);
        } else {
            elemento[numeroElementi] = nuovoElemento;
            numeroElementi++;
        }
    } //altrimenti non fare nulla. L'elemento e' gia' nella lista.
}

/**
Se l'argomento indica una posizione nella lista,
viene restituito l'elemento in quella specifica posizione;
altrimenti viene restituito null.
*/
public String getElementoIn(int posizione) {
    String risultato = null;
```

```
        if ((1 <= posizione) && (posizione <= numeroElementi))
            risultato = elemento[posizione - 1];

        return risultato;
    }

    /**
    Restituisce true se la posizione passata come argomento
    indica l'ultima posizione nella lista; altrimenti restituisce false.
    */
    public boolean ultimoElemento(int posizione) {
        return posizione == numeroElementi;
    }

    /**
    Restituisce true se l'elemento e' nella lista;
    altrimenti restituisce false. Non distingue
    tra lettere maiuscole e minuscole.
    */
    public boolean nellaLista(String elementoDaRicercare) {
        boolean trovato = false;
        int i = 0;
        while (!trovato && (i < numeroElementi)) {
            if (elementoDaRicercare.equalsIgnoreCase(elemento[i]))
                trovato = true;
            else
                i++;
        }
        return trovato;
    }

    public int getMassimoNumeroDiElementi() {
        return elemento.length;
    }

    public int getNumeroDiElementi() {
        return numeroElementi;
    }

    public void cancellaLista() {
        numeroElementi = 0;
    }
}
```

Sebbene l'array `elemento` abbia gli indici che partono da 0, se si utilizza una variabile `int` come indicatore della posizione (come la variabile `posizione` nel Listato 9.21), la numerazione partirebbe da 1 e non da 0. I metodi della classe aggiustano automaticamente gli indici. Quindi, quando si chiede l'elemento che si trova in `posizione`, viene restituito quello alla posizione `elemento[posizione - 1]`. È possibile tuttavia compensare la differenza tra gli indici dell'array e le posizioni della lista allocando degli elementi aggiuntivi nell'array `elemento` e ignorando `elemento[0]`, come suggerito in precedenza.

L'Esercizio 18 alla fine di questo capitolo chiederà di fare quanto appena detto.

Si noti che la classe `ListaSenzaRipetizioni` offre tre modi per individuare la fine della lista, supponendo che si utilizzi una variabile posizione `int` per accedere agli elementi della lista.

- Se `posizione` è uguale a `getNumeroDiElementi()`, `posizione` è sull'ultimo elemento.
- Se `ultimoElemento(posizione)` restituisce `true`, `posizione` è sull'ultimo elemento.
- Se `getElementoIn(posizione)` restituisce `null`, `posizione` è oltre l'ultimo elemento.

Si conclude questo programma menzionando che la Java Class Library contiene la classe `ArrayList`. Tale classe permette di creare delle liste più generali e flessibili rispetto a quella utilizzata in precedenza. Come la classe `ListaSenzaRipetizioni`, `ArrayList` utilizza un array per rappresentare una lista. Il Capitolo 12 introdurrà questa classe.

9.7 Enumerazioni come classi

Il Capitolo 3 ha presentato le enumerazioni spiegando che ogni volta che incontra un'enumerazione, il compilatore crea una nuova classe. Questo paragrafo tratta proprio questo tipo di classi.

Si consideri, per esempio, una semplice enumerazione: i semi delle carte da gioco.

```
enum Semi {FIORI, QUADRI, CUORI, PICCHE}
```

Il compilatore, in questo caso, crea la classe `Semi`. I valori elencati sono nomi di oggetti pubblici e statici il cui tipo è `Semi`. Tutti questi valori possono essere impiegati nel programma usando il termine `Semi` seguito da un punto, come nell'esempio che segue:

```
Semi s = Semi.QUADRI;
```

La classe `Semi` ha diversi metodi, fra cui `equals`, `compareTo`, `ordinal`, `toString` e `valueOf`. Gli esempi che seguono mostrano come si possano usare questi metodi. Negli esempi si suppone che la variabile `s` sia `Semi.QUADRI`.

- `s.equals(Semi.CUORI)` verifica se `s` è uguale a `CUORI`. In questo caso restituisce `false`, poiché `QUADRI` non è uguale a `CUORI`.

- `s.compareTo(Semi.CUORI)` verifica se `s` precede, è uguale o segue `CUORI` nella definizione di `Semi`. Come il metodo `compareTo` della classe `String`, descritto nel Capitolo 3, questo metodo restituisce un intero che è negativo, zero o positivo a seconda del risultato del confronto. In questo caso il metodo `compareTo` restituisce un intero negativo poiché `QUADRI` compare prima di `CUORI` nell'enumerazione.
- `s.ordinal()` restituisce la posizione, o **valore ordinale**, di `QUADRI` nell'enumerazione. Gli oggetti all'interno dell'enumerazione sono numerati a partire da 0. Quindi, in questo esempio, l'invocazione del metodo `s.ordinal()` restituisce 1.
- `s.toString()` restituisce la stringa "`QUADRI`". Restituisce, cioè, il nome dell'oggetto chiamante del metodo come una stringa.
- `Semi.valueOf("CUORI")` restituisce l'oggetto `Semi.CUORI`. La corrispondenza tra la stringa passata in input e il nome dell'oggetto nell'enumerazione deve essere esatta; non vengono ignorati nemmeno gli spazi all'interno della stringa. Il metodo `valueOf` è statico, pertanto deve essere preceduto dal nome dell'enumerazione di appartenenza, `Semi`, e dal punto.

In ogni enumerazione si possono definire variabili di istanza private e metodi pubblici, costruttori compresi. Definendo una variabile di istanza, si possono assegnare facilmente dei valori agli oggetti dell'enumerazione. L'aggiunta di un metodo *get* permetterebbe di accedere a questi valori. Queste operazioni sono state effettuate nella nuova definizione dell'enumerazione `Semi`, mostrata nel Listato 9.22.

MyLab

LISTATO 9.22 Enumerazione rivisitata.

```
/**
 Un'enumerazione di Semi di carte
*/
enum Semi {

    FIORI("nero"), QUADRI("rosso"), CUORI("rosso"), PICCHE("nero");

    private final String colore;

    private Semi(String coloreSeme) {
        colore = coloreSeme;
    }

    public String getColore() {
        return colore;
    }
}
```

Sono stati scelti valori stringa come valori per gli oggetti enumerati, quindi è stata introdotta la variabile di istanza `colore` come un oggetto di tipo `String`, così come un costruttore che ne imposta il valore. L'istruzione `PICCHE("nero")` invoca il costruttore e imposta il valore della variabile di istanza privata di `PICCHE` a "`nero`". Si osservi che il

valore di `colore` non può cambiare dal momento che la variabile è stata dichiarata `final`. Si osservi, inoltre, che il costruttore è privato e che quindi non può essere invocato direttamente: il costruttore può essere invocato solo all'interno della definizione di `Seme`. Il metodo `getColore` fornisce un accesso pubblico al valore della variabile `colore`.

Alle enumerazioni possono essere assegnati dei modificatori di visibilità, come `public` o `private`. Se vengono omessi, un'enumerazione è considerata privata. Si può definire un'enumerazione all'interno di un file distinto, così come si definirebbe una qualsiasi altra classe.

9.8 Package

Un **package** (letteralmente "pacchetto") è una collezione di classi correlate a cui viene assegnato un nome. Un package svolge il ruolo di libreria di classi, le quali possono essere utilizzate all'interno di un qualsiasi programma. Grazie ai package non occorre posizionare tutte queste classi nella stessa cartella del programma. Sebbene questo sia un argomento molto importante e utile, il resto del testo non utilizza i package. Di conseguenza, è possibile leggere questo paragrafo in modo completamente indipendentemente rispetto al resto del libro.

Per capire al meglio questo paragrafo è necessario sapere cosa sono le cartelle (o directory); che cosa sono i nomi di percorso (*path name*) delle cartelle e come il sistema operativo utilizza una variabile di percorso (*variabile path*). Se non si conoscono questi aspetti, è bene riprendere questo paragrafo in un secondo tempo dopo aver chiarito questi concetti. Questi argomenti non riguardano solo Java, ma sono parte del sistema operativo e pertanto i dettagli dipendono proprio dal sistema operativo utilizzato. Se si sa come modificare una variabile di percorso si può procedere con la lettura dei prossimi paragrafi.

9.8.1 Package e istruzione `import`

Un package è semplicemente una collezione di classi che sono state raggruppate in una cartella. Il nome della cartella deve corrispondere al nome del package. All'interno del package le classi sono disposte in file distinti, i cui nomi corrispondono al nome della classe, come si è visto nei capitoli precedenti. L'unica differenza rispetto a quanto visto in precedenza è che ciascun file del package deve contenere la seguente riga all'inizio del file stesso:

```
package nome_package;
```

Niente può precedere questa riga a eccezione di righe bianche e commenti. *nome_package* tipicamente è formato da lettere minuscole, spesso separate da punti. Per esempio, se il nome di un certo package è `generale.utilita`, ciascun file all'interno del package conterrà la seguente istruzione all'inizio del file:

```
package generale.utilita;
```

Qualsiasi programma o classe può usare tutte le classi contenute in un package inserendo un'opportuna istruzione `import` all'inizio del file contenente il programma o la definizione della classe. Questa regola vale anche se il programma o la definizione della classe non sono contenuti nella stessa cartella delle classi del package. Per esempio, per usare la

`ClasseAusiliaria` contenuta nel package `generale.utilita`, basta inserire l'istruzione seguente all'inizio del file che si sta sviluppando:

```
import generale.utilita.ClasseAusiliaria;
```

Si noti che si scrive il nome della classe, `ClasseAusiliaria`, e non il nome del file in cui questa classe è definita, `ClasseAusiliaria.java`. Per importare tutte le classi del package `generale.utilita`, si può usare un'istruzione come la seguente:

```
import generale.utilita.*;
```

L'asterisco indica l'intenzione di importare tutti i file del package. Sebbene questa istruzione venga usata da diversi programmatori, si consiglia di importare le singole classi.

L'istruzione package

Un package è una collezione di classi raggruppate in una cartella a cui viene dato il nome del package stesso. Ciascuna classe è definita in un file distinto che ha lo stesso nome della classe. Ciascun file che appartiene a un package deve iniziare con l'istruzione `package`, eventualmente preceduta da righe vuote o commenti.

Sintassi di una classe in un package

```
<Righe vuote o commenti.>
package nome_package;
<Definizione della classe.>
```

Esempi

```
package generale.utilita;
package java.io;
```

L'istruzione import

Si possono usare tutte le classi contenute in un package inserendo l'istruzione `import` che indichi il package che contiene la classe che si vuole utilizzare. L'istruzione `import` deve trovarsi all'inizio del file. Il programma o la classe che intende usare una classe definita in un package non deve necessariamente risiedere nella stessa cartella del package.

Sintassi

```
import nome_package.nome_classe_o_asterisco;
```

Se si scrive il nome di una classe, viene importata solo la classe specificata; se si scrive il carattere asterisco, vengono importate tutte le classi contenute in quel package.

Esempi

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

9.8.2 Nomi di package e cartelle

Il nome di un package non è un identificatore qualsiasi, ma indica al compilatore dove può trovare le classi contenute nel package. In realtà, il nome del package indica al compilatore il percorso della cartella contenente le classi del package. Per individuare la cartella che corrisponde a un package Java, il compilatore ha bisogno di due informazioni: il nome del package e le cartelle elencate all'interno della variabile di percorso chiamata *class path*.

Il valore della **variabile class path** indica a Java dove iniziare la propria ricerca per individuare un certo package. La variabile *class path* non è una variabile Java, ma è una variabile che fa parte del sistema operativo utilizzato e che contiene i nomi di percorsi di un certo elenco di directory. Quando Java cerca un package, inizia da queste cartelle. Queste cartelle sono dette **cartelle base del class path**. I prossimi paragrafi indicano sia come Java usa le cartelle base del *class path*, sia come definire la variabile *class path*.

Il nome di un package specifica il percorso relativo di una cartella che contiene le classi del package. Questo è un percorso relativo, in quanto presuppone che le classi si trovino in una certa cartella contenuta all'interno di una cartella base del *class path*. Si supponga, per esempio, che quella che segue sia una cartella base del *class path* (il sistema operativo potrebbe usare il carattere / invece del carattere \):

```
\programmijava\librerie
```

e si supponga che le classi del package siano contenute nella cartella:

```
\programmijava\librerie\generale\utilita
```

In questo caso, il package deve avere il nome `generale.utilita`.

Si noti che il nome di un package non è arbitrario, ma deve corrispondere a un elenco di cartelle contenute all'interno di una cartella base del *class path*. Il nome del package, quindi, indica a Java quali sono le sotto-cartelle in cui deve entrare, a partire dalla cartella base del *class path*, per trovare le classi del package. La Figura 9.12 mostra proprio questa organizzazione. Il punto nel nome del package ha lo stesso significato dei caratteri \ e / o comunque del simbolo usato dal sistema operativo per indicare i percorsi sul file system.

Le cartelle base del *class path* vengono specificate attraverso la variabile d'ambiente (*environment variable*) `CLASSPATH`. Il modo in cui si assegna il *class path* dipende dal sistema operativo utilizzato. Nei sistemi UNIX si usa un comando come il seguente:

```
export CLASSPATH=/home/io/programmijava/librerie/
```

Se si sta usando un sistema Windows, si può definire il *class path* usando il Pannello di controllo per creare (o modificare) la variabile d'ambiente `CLASSPATH`.
Si possono elencare più cartelle base in una variabile `CLASSPATH` separandole con il carattere ";" nei sistemi Windows o con il carattere ":" nei sistemi `UNIX`. Per esempio, la seguente riga potrebbe corrispondere a un *class path* in un sistema Windows.

```
c:\programmijava\librerie;f:\altriprogrammi
```

Questo vuol dire che si possono creare package sia all'interno di:

```
c:\programmijava\librerie\
```

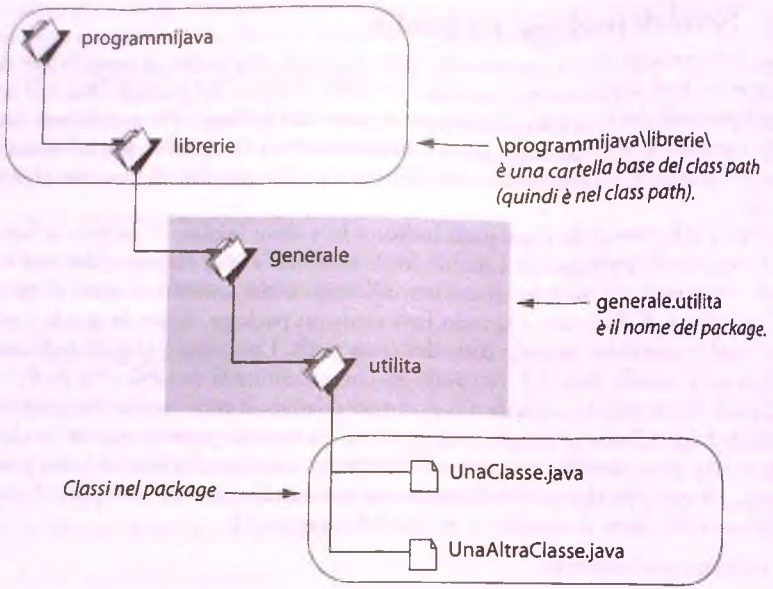


Figura 9.12 Il nome di un package.

sia all'interno di:

```
f:\altriprogrammi
```

Per individuare i package da usare, Java cerca dapprima nelle sottocartelle di:

```
c:\programmijava\librerie\
```

e quindi, se non trova il package, cerca nelle sottocartelle di:

```
f:\altriprogrammi
```

Ogni volta che si assegna o si modifica la variabile `CLASSPATH`, è bene includere la cartella corrente come una delle possibili alternative. La cartella corrente è la cartella in cui si trova il programma. Nella maggior parte dei sistemi la cartella corrente è rappresentata con il carattere ".". Per esempio, si potrebbe usare il seguente *class path*:

```
c:\programmijava\librerie;f:\altriprogrammi;.
```

In questo *class path*, è stato aggiunto il punto, cioè la cartella corrente al *class path*. In questo modo, se il package non viene trovato nelle prime due cartelle, Java lo cerca nelle sottocartelle della cartella corrente, cioè nelle sottocartelle del programma che si sta compilando. Se si desidera che Java cerchi prima nella cartella corrente e poi nelle altre, basta specificare il punto all'inizio della lista.

Nomi di package

Il nome di un package deve corrispondere al nome del percorso di una cartella che contiene le classi del package. Il nome del package però usa il carattere ".", al posto dei caratteri \ o /, per separare le cartelle. Quando si nomina un package, si utilizza un percorso relativo che parte da una delle cartelle contenute nella variabile d'ambiente `CLASSPATH`.

Esempi

```
generale.utilita;
java.io;
```

Non includere la cartella corrente nel *class path*

Omettere la cartella corrente dal *class path* non limita solamente il numero di posti utilizzabili per trovare i package, ma potrebbe interferire con i programmi che non usano package. Se non si creano package, ma si posizionano tutte le classi nella stessa cartella, come peraltro viene fatto in questo testo, Java non sarà in grado di individuare le classi, a meno che la cartella corrente non sia nel *class path*. Si noti che questo problema non si verifica se non si ha alcuna variabile *class path*, ma solo se si decide di utilizzare la variabile `CLASSPATH`.

9.8.3 Conflitti tra nomi

I package rappresentano un modo conveniente per raggruppare e usare librerie di classi, ma esiste anche un altro motivo per utilizzare i package. I package, infatti, possono aiutare lo sviluppatore nel gestire eventuali conflitti tra i nomi delle classi. Potrebbero cioè aiutare a gestire casi in cui due classi abbiano lo stesso nome. Per esempio, se due programmatori differenti hanno usato lo stesso nome per una classe, ma l'hanno posizionata in package diversi, l'ambiguità del nome della classe viene risolta proprio grazie al nome del package.

Si supponga, per esempio, che il package `utilitaMie` contenga la classe `ClasseUtile` e che un altro package `utilitaTue` contenga una classe differente, che tuttavia si chiama anch'essa `ClasseUtile`. Entrambe le classi potrebbero essere utilizzate all'interno di uno stesso programma usando i nomi `utilitaMie.ClasseUtile` e `utilitaTue.ClasseUtile` come viene mostrato in questo esempio:

```
utilitaMie.ClasseUtile oggetto1 = new utilitaMie.ClasseUtile();
utilitaTue.ClasseUtile oggetto2 = new utilitaTue.ClasseUtile();
```

Se si elenca il nome del package seguito dal nome della classe, non è necessario importare il package in quanto il nome esteso della classe include già il nome del package.

9.9 Riepilogo

- Un costruttore è un metodo che, invocato con l'operatore new, crea e inizializza un oggetto di una classe. Un costruttore deve avere lo stesso nome della classe.
- Un costruttore che non riceve alcun parametro è detto costruttore di default. A una classe che non definisce alcun costruttore viene assegnato automaticamente un costruttore di default. Una classe che definisce uno o più costruttori, nessuno dei quali corrisponde a un costruttore di default, non avrà alcun costruttore di default.
- Quando si definisce un costruttore per una classe si può usare la parola chiave `this` per indicare un altro costruttore della stessa classe. Qualsiasi invocazione a `this` deve essere la prima azione svolta dal costruttore che effettua l'invocazione.
- La dichiarazione di una variabile statica deve contenere la parola chiave `static`. Una variabile statica è condivisa da tutti gli oggetti di una classe.
- L'intestazione di un metodo statico contiene la parola chiave `static`. Un metodo statico è un metodo che viene tipicamente invocato usando il nome della classe invece del nome di un oggetto. Un metodo statico non può far riferimento a una variabile di istanza della classe, né può invocare un metodo di istanza se non utilizzando un'istanza della classe.
- In Java ciascun tipo primitivo ha una corrispondente classe *wrapper* che fornisce una versione di tipo classe per il tipo primitivo. Le classi *wrapper* contengono anche una serie di costanti e metodi predefiniti molto utili.
- Java effettua la conversione automatica di tipo tra un tipo primitivo e la corrispondente classe *wrapper* ogni volta che è necessario.
- Due o più metodi all'interno di una stessa classe possono avere lo stesso nome se presentano un diverso numero di parametri o se presentano parametri di tipo diverso. Cioè se i metodi hanno firme differenti. Questa caratteristica è detta *overloading* del nome del metodo.
- Gli array possono essere utilizzati come variabili di istanza di una classe.
- Una classe può essere il tipo base di un array. Gli array di reference possono essere utilizzati per realizzare associazioni uno-a-molti.
- Un'enumerazione è una classe. Per questo motivo, all'interno di un'enumerazione si possono definire variabili di istanza, costruttori e metodi.
- Si può definire una collezione contenente le classi usate più di frequente. Questa collezione prende il nome di package. Ogni classe nel package deve essere definita nel proprio file contenuto all'interno della stessa cartella e deve iniziare con un'istruzione di tipo `package`.
- Si possono usare le classi contenute in un package in un qualsiasi programma senza doverle spostare nella stessa cartella del programma, ma semplicemente inserendo un'istruzione di `import` all'inizio del programma stesso.

9.10 Esercizi

1. Si crei una classe che include più metodi statici che computano l'ammontare delle imposte. Questa classe non dovrebbe avere un costruttore. I suoi attributi sono i seguenti:
   - `impostaBase` – l'imposta di base, un variabile statica di tipo `double` che ha un valore iniziale del 4%;
   - `impostaLusso` – l'imposta di lusso, una variabile statica di tipo `double` inizializzata al 10%.

   I metodi di questa classe sono i seguenti:
   - `computaCostoBase(prezzo)` – un metodo statico che restituisce il prezzo sommato all'imposta di base e arrotondato al centesimo più vicino;
   - `computaCostoLusso(prezzo)` – un metodo statico che restituisce il prezzo sommato all'imposta di lusso, arrotondato al centesimo più vicino;
   - `cambiaImpostaBase(nuovaImpostaBase)` – un metodo statico che cambia l'imposta di base;
   - `cambiaImpostaDiLusso(nuovaImpostaLusso)` – un metodo statico che cambia l'imposta di lusso;
   - `arrotondaACentesimoVicino(prezzo)` – un metodo statico privato che restituisce il prezzo arrotondato al centesimo più vicino. Per esempio, se il prezzo è 12.567 questo metodo restituirà il valore 12.57.
2. Si consideri la classe `Ora` che rappresenta una certa ora del giorno. Questa classe ha delle variabili di istanza per rappresentare l'ora e i minuti. Il valore delle ore varia da 0 a 23. I minuti variano da 0 a 59.
   a. Si scriva un costruttore di default che inizializza l'ora a 0 ore e 0 minuti.
   b. Si scriva un metodo privato `valida(ore, minuti)` che restituisce il valore `true` se i valori passati sono validi.
   c. Si scriva il metodo `setOra(ore, minuti)` che assegna l'ora data se i valori passati sono validi.
   d. Si scriva un altro metodo `setOra(ore, minuti, AM)` che assegna l'ora data se i valori sono validi. Il parametro `ore` deve essere nel range 1-12. Il parametro AM è `true` se le ore sono mattutine, altrimenti deve essere `false`.
3. Si scrivano un costruttore di default e un secondo costruttore per la classe `Punteggio`, descritta nell'Esercizio 9 del capitolo precedente.
4. Si scriva un costruttore per la classe `ProgettoScienzaPunteggio` descritta nell'Esercizio 10 del capitolo precedente. Si assegnino a questo costruttore tre parametri corrispondenti ai primi tre attributi descritti dall'esercizio. Il costruttore dovrebbe assegnare valori di default agli altri attributi.

5. Si consideri la classe `Caratteristiche` da utilizzare in un servizio di appuntamenti on-line e che permette di capire quanto siano compatibili due persone. Gli attributi sono i seguenti:
   - `descrizione` – una stringa che identifica le caratteristiche;
   - `punteggio` – un intero da 1 a 10 che indica quanto una persona ricerchi questa caratteristica in un'altra persona.

   a. Si scriva un costruttore che assegni una stringa data alla descrizione e che assegni il valore 0 al punteggio per indicare che questo non è stato ancora indicato.
   b. Si scriva il metodo privato `valido(punteggio)` che restituisce vero se il punteggio dato è valido e cioè se è compreso tra 1 e 10.
   c. Si scriva il metodo `setPunteggio(punteggio)` che assegna il punteggio dato se questo è valido.
   d. Si scriva il metodo `setPunteggio` che legge il punteggio inserito da tastiera, continuando a richiederlo se il punteggio inserito non è valido.

6. Si crei la classe `OccupazioneStanza` che può essere usata per memorizzare il numero di persone presenti in una stanza di un edificio. Questa classe presenta i seguenti attributi:
   - `numeroNellaStanza` – numero di persone nella stanza;
   - `numeroTotale` – variabile statica che indica il numero totale di persone in tutte le stanze.

   La classe deve presentare i seguenti metodi:
   - `aggiungiUnoAllaStanza` – aggiunge una persona alla stanza e incrementa il valore di `numeroTotale`;
   - `rimuoviUnoDallaStanza` – rimuove una persona dalla stanza, assicurandosi che `numeroNellaStanza` non diventi minore di 0 e decrementa il valore di `numeroTotale` come richiesto;
   - `getNumero` – restituisce il numero di perone nella stanza;
   - `getTotale` – metodo statico che restituisce il numero di persone totali.

7. Si scriva un programma che collaudi la classe `OccupazioneStanza` descritta nell'esercizio precedente.

8. Alle volte è necessario avere classi che hanno una sola istanza. Si crei una classe `Merlino` che ha una variabile, `mago`, che è statica e di tipo `Merlino`. La classe ha solo un costruttore e due metodi.
   - `Merlino` – un costruttore privato. Solo questa classe può invocare questo costruttore e nessun'altra classe o programma può farlo.
   - `chiama` – un metodo statico che restituisce il valore dell'attributo `mago` se non è nullo. Altrimenti, se `mago` è `null`, questo metodo crea un'istanza di `Merlino` usando il costruttore privato e lo assegna alla variabile `mago` prima di restituirlo al chiamante.
   - `consulta` – un metodo di istanza che restituisce la stringa `"Estrai la spada dalla roccia"`.

9. Si scriva un programma che verifichi la correttezza della classe `Merlino`. Si utilizzi il metodo `toString` per verificare che sia stata creata una sola istanza.
10. Si consideri una classe `Persona` che descrive una generica persona. Questa classe ha una variabile di istanza di tipo stringa nome che indica il nome della persona e una variabile di istanza di tipo intero `eta`, che rappresenta l'età di una persona.
    a. Si scriva un costruttore di default per la classe `Persona` che assegni `"Nessun nome"` a `nome` e `0` a `eta`.
    b. Si scriva un secondo costruttore che assegni la stringa fornita in ingresso a `nome` e l'intero fornito a `eta`.
    c. Si scriva un metodo statico `creaPersonaAdulta` che restituisce un'istanza speciale di questa classe. L'istanza restituita rappresenta un generico individuo adulto che ha come `nome` la stringa `"Un adulto"` e come `eta` 21.
11. Si crei una classe `Androide` i cui oggetti hanno valori univoci. La classe deve avere le seguenti variabili:
    - `tag` – un intero statico che inizia per `1` e cambia ogni volta che viene creata un'istanza;
    - `nome` – una stringa univoca per ciascuna istanza.

    La classe `Androide` ha i seguenti metodi:
    - `Androide` – un costruttore di default che assegna il nome `"Bob"` seguito dal valore di `tag`; dopo aver impostato il nome, questo costruttore cambia il valore di `tag` invocando il metodo privato `cambiaTag`;
    - `getNome` – restituisce il nome;
    - `isPrimo(n)` – metodo statico che restituisce vero se `n` è un numero primo, cioè se non è divisibile per nessun numero compreso tra 2 e `n - 1`;
    - `cambiaTag` – un metodo statico privato che sostituisce `tag` con il numero primo che segue il valore corrente di `tag`.
12. Si crei un programma che collaudi la classe `Androide` realizzata.
13. Scrivere un programma in una classe `ContaPoveri` che conti il numero di famiglie che vengono considerate povere. Scrivere e utilizzare una classe `Famiglia` che ha i seguenti attributi:
    - `reddito` – un valore `double` che è il reddito della famiglia;
    - `dimensione` – il numero di componenti della famiglia;

    e i seguenti metodi:
    - `Famiglia(reddito, dimensione)` – il costruttore che inizializza gli attributi;
    - `povera(costoCasa, costoCibo)` – un metodo che restituisce vero se `costoCasa + costoCibo * dimensione` è maggiore della metà del reddito della famiglia (`costoCibo` è il costo medio del cibo per ogni individuo, mentre `costoCasa` è unico per la famiglia);
    - `toString` – un metodo che restituisce una stringa contenente le informazioni della famiglia;

Il programma deve leggere da tastiera un intero `k` e, successivamente, creare un array di dimensione `k` il cui tipo base è `Famiglia`. Deve inoltre creare `k` oggetti di tipo `Famiglia` e inserirli nell'array, leggendo da tastiera il reddito e la dimensione di ogni famiglia. Dopo aver letto da tastiera un costo medio familiare e un costo medio del cibo, visualizzare le famiglie che sono povere.

14. Creare una classe `LibroMastro` per registrare le vendite di un negozio. Essa deve avere i seguenti attributi:
    - `vendite` – un array di valori `double` che corrisponde agli importi di tutte le vendite;
    - `venditeEffettuate` – il numero di vendite effettuate;
    - `massimoVendite` – il massimo numero di vendite che può essere registrato;

    e i seguenti metodi:
    - `LibroMastro(massimo)` – un costruttore che inizializza a `massimo` il massimo numero di vendite;
    - `aggiungiVendita(d)` – aggiunge una vendita il cui valore è `d`;
    - `getNumeroDiVendite` – restituisce il numero di vendite effettuate;
    - `getTotaleVendite` – restituisce il valore totale delle vendite.

15. Definire i seguenti metodi per la classe `LibroMastro`, come descritto nel precedente esercizio:
    - `getMediaVendite()` – restituisce il valore medio di tutte le vendite;
    - `getVenditeAlDiSopra(v)` – restituisce il numero di vendite che hanno un valore superiore a `v`.

16. Creare una classe `Polinomio` utilizzata per valutare una funzione polinomiale in x:

    $P(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1} + a_n x^n$

    I coefficienti a_i sono numeri in virgola mobile, gli esponenti di x sono interi e l'esponente più grande n (il grado del polinomio) è maggiore o uguale a 0. La classe ha i seguenti attributi:
    - `grado` – valore del più grande esponente del polinomio n;
    - `coefficienti` – array dei coefficienti a_i;

    e i seguenti metodi:
    - `Polinomio(massimo)` – un costruttore che crea un polinomio di grado `massimo` i cui coefficienti sono tutti 0;
    - `setCostante(i, valore)` – imposta il coefficiente a_i a `valore`;
    - `valuta(x)` – restituisce il valore del polinomio per un dato valore `x`.

    Per esempio, il polinomio:

    $P(x) = 3 + 5 x + 2 x^3$

    è di grado 3 e ha coefficienti $a_0 = 3$, $a_1 = 5$, $a_2 = 0$ e $a_3 = 2$. L'invocazione del metodo `valuta(7)` calcola l'espressione $3 + 5 * 7 + 0 * 7^2 + 2 * 7^3$, che è 3 + 35 + 0 + 686 e restituisce il risultato 724.

17. Scrivere un metodo `oltreUltimoElemento(posizione)` per la classe `ListaSenzaRipetizioni`, come fornita nel Listato 9.21, che restituisce vero quando `posizione` è oltre l'ultimo elemento della lista.
18. Correggere la classe `ListaSenzaRipetizioni`, come fornita nel Listato 9.21, in modo che venga allocato un elemento aggiuntivo all'array `elemento` e ignori `elemento[0]`, come suggerito verso la fine del Paragrafo 6.1.4.
19. Correggere la classe `LibroDelTempo` che appare nel Listato 9.17 in modo che utilizzi una enumerazione per i giorni della settimana anziché delle costanti.

9.11 Progetti

1. Si modifichi la definizione della classe `Specie` nel Listato 8.16 del Capitolo 8 rimuovendo il metodo `setSpecie` e aggiungendo i metodi seguenti.
   - Cinque costruttori: uno per ciascuna variabile di istanza, uno con tre parametri per le tre variabili di istanza, un costruttore di default. Ci si accerti che ciascun costruttore assegni valori a tutte le variabili di istanza.
   - Quattro metodi *set* per resettare i valori: uno deve corrispondere al metodo `setSpecie` del Listato 8.15, mentre gli altri tre devono resettare ciascuna variabile di istanza. Si scriva quindi un programma di test per verificare ciascuno dei metodi implementati.

   Si ripeta il Progetto 1 del Capitolo 8, ma assicurandosi che venga utilizzato un costruttore diverso da quello di default quando si istanziano nuovi oggetti della classe `Specie`.
2. Si ripeta il Progetto 4 del Capitolo 8. Questa volta si aggiungano i seguenti costruttori:
   - uno per ciascuna variabile di istanza;
   - uno che riceve due parametri per le due variabili di istanza;
   - un costruttore di default.

   Ci si assicuri che ciascun costruttore assegni valori a tutte le variabili di istanza. Si scriva un programma *driver* per collaudare tutti i metodi del programma.
3. Si usi la classe `Animale` del Listato 9.1 per scrivere un programma che legge i dati di cinque animali e mostri le seguenti informazioni: nome del più piccolo, nome del più grande, nome del più vecchio, nome del più giovane, peso medio dei cinque animali ed età media dei cinque animali.
4. Si completi e si collaudi a fondo la classe `Ora` descritta nell'Esercizio 2. Si aggiungano altri due costruttori analoghi ai metodi `setOra` descritti nelle parti *c* e *d* dell'esercizio. Inoltre, si includano i seguenti metodi:
   - `getOra24` – restituisce una stringa che rappresenta l'ora del giorno in una notazione a 24 ore: hhmm. Per esempio, se i valori di ore e minuti sono rispettivamente 7 e 25, deve restituire "0725". Se i valori di ore e minuti sono rispettivamente 0 e 5, restituisce "0005". Se i valori di ore e minuti sono rispettivamente 15 e 30, restituisce "1530";

- `getOra12` – restituisce le ore in una notazione a 12 ore: h:mm xx. Per esempio, se le ore e i minuti valgono rispettivamente 7 e 25, restituisce "7:25 am". Se il valore delle ore è 0 e quello dei minuti è 5, restituisce "12:05 am". Se il valore delle ore è 15 e quello dei minuti è 30, restituisce "3:30 pm".

5. Si completi e collaudi a fondo la classe `Caratteristiche` dell'Esercizio 5. Si includano inoltre i seguenti metodi:
   - `getDescrizione` – restituisce la descrizione della caratteristica;
   - `getPunteggio` – restituisce il punteggio della caratteristica;
   - `getCompatibilita(Caratteristica altraCratteristica)` – restituisce la misura di compatibilità tra due caratteristiche o 0 se le descrizioni con coincidono;
   - `getMisuraDiCompatibilita(Caratteristica altraCaratteristica)` – un metodo privato che restituisce una misura di compatibilità come un valore `double` usando la formula $m = 1 - (r1 - r2)^2 / 81$. Nel caso in cui uno dei due punteggi (*r1* o *r2*) sia uguale 0, deve restituire 0. Si ricordi l'Esercizio 5, in cui il costruttore assegna il valore 0 al punteggio, indicando che non è stato determinato;
   - `corrispondenza(Caratteristica altraCaratteristica)` – un metodo privato che restituisce `true` se le descrizioni corrispondono.
6. Si scriva un'enumerazione `VotoLettere` che rappresenta i punteggi da A a F, includendo punteggi positivi e negativi. Si definisca una variabile di istanza privata che vale `true` se il punteggio è positivo. Inoltre si definisca un costruttore che inizializza la variabile di istanza, un metodo che restituisce il valore di questa variabile e un metodo `toString` che restituisce il voto come una stringa. Infine, si scriva un programma che mostri il funzionamento dell'enumerazione.
7. Si completi e collaudi a fondo la classe `Persona` descritta nell'Esercizio 10. Si includano i seguenti metodi:
   - `getNome` – restituisce il nome di una persona come stringa;
   - `getEta` – restituisce l'età della persona;
   - `setNome(nome, cognome)` – imposta il nome e il cognome di una persona;
   - `setNome(nomeCompleto)` – imposta il nome di una persona, dati il nome e il cognome in una sola stringa;
   - `setEta(eta)` – imposta l'età di una persona;
   - `creaInfante` – metodo statico che restituisce una speciale istanza di questa classe che rappresenta un infante. L'istanza ha il nome "Un infante" e un'età che vale 2;
   - `creaBambino` – metodo statico che crea una speciale istanza di questa classe che rappresenta un bambino in età pre-scolare e ha il nome "Un bambino" e un'età che vale 5;
   - `creaPreAdolescente` – un metodo statico che restituisce una speciale istanza di questa classe che rappresenta un pre-adolescente. L'istanza ha il nome "Un pre-adolescente" e un età che vale 9;

- `creaAdolescente` – un metodo statico che restituisce una speciale istanza di questa classe che rappresenta un adolescente. L'istanza ha il nome "Un adolescente" e un età che vale 15.

8. Si scriva una classe `Temperatura` che rappresenta le temperature in gradi Celsius e Fahrenheit. Si usi un numero in virgola mobile per le temperature e un carattere per la scala: "`C`" per Celsius e "`F`" per Fahrenheit. La classe dovrebbe avere i seguenti elementi:

   - quattro costruttori: uno che riceve in input i gradi, uno la scala, uno entrambi e un costruttore di default. Per ciascuno di questi costruttori si supponga che se non viene specificata la scala, sia di tipo Celsius e che se non vengono forniti i gradi, siano 0 gradi;
   - due metodi *get*: uno che restituisce la temperatura in gradi Celsius, l'altro in gradi Fahrenheit. Si usi la formula del Progetto 5 del Capitolo 3 e si arrotondi al decimo di grado;
   - tre metodi *set*: uno che imposta i gradi, uno che imposta la scala e uno entrambi;
   - tre metodi di confronto: uno che verifica se due temperature sono uguali, uno che verifica se una temperatura è minore di un altra, uno che verifica se una temperatura è maggiore di un'altra.

   Si scriva un programma *driver* che verifichi tutti questi metodi. Si faccia attenzione a invocare tutti i costruttori, i metodi e a controllare le seguenti coppie di valori nell'uguaglianza:

   - 0.0 gradi C e 32.0 gradi F;
   - –40.0 gradi C e –40.0 gradi F;
   - 100.0 gradi C e 212.0 gradi F.

9. Si ripeta il Progetto 10 del Capitolo 8, ma si includano i costruttori.

10. Si scriva e si collaudi a fondo una classe che rappresenta i numeri razionali. Un numero razionale può essere rappresentato come il rapporto fra due valori interi, *a* e *b*, dove *b* è un valore diverso da 0. La classe deve avere attributi che rappresentano rispettivamente il numeratore e il denominatore. Il rapporto deve essere sempre rappresentato nella sua forma più semplice, cioè devono essere rimossi i fattori comuni. Per esempio, il numero razionale 40 / 12 deve essere memorizzato come 10 / 3.

    La classe deve avere i seguenti metodi e costruttori:

    - un costruttore di default che assegna al numero razionale il valore 0 / 1;
    - un costruttore che riceve come parametri il numeratore e il denominatore e converte il rapporto risultante nella sua forma più semplice;
    - `semplifica` – un metodo privato che converte un numero razionale nella sua forma più semplice;
    - `getGCD(x,y)` – un metodo statico privato che restituisce il massimo comune divisore di due interi positivi x e y. Per esempio, il massimo comune divisore di 40 e 12 è 4;
    - `getValore` – restituisce un numero razionale sotto forma di valore `double`;
    - `toString` – restituisce il numero razionale sotto forma di una stringa *a* / *b*.

11. Si scriva un programma che registra i voti per uno tra due candidati usando la classe `RegistratoreVoti`, che deve essere progettata e implementata secondo le specifiche di seguito fornite. `RegistratoreVoti` deve avere delle variabili statiche per tracciare il numero totale di voti per i candidati e variabili di istanza per tracciare i voti fatti da una singola persona. La classe avrà le seguenti variabili:
    - `nomeCandidatoPresidente1` – una variabile statica di tipo `String` che contiene il nome del primo candidato presidente;
    - `nomeCandidatoPresidente2` – una variabile statica di tipo `String` che contiene il nome del secondo candidato presidente;
    - `nomeCandidatoVicePresidente1` – una variabile statica di tipo `String` che contiene il nome del primo candidato alla carica di vicepresidente;
    - `nomeCandidatoVicePresidente2` – una variabile statica di tipo `String` che contiene il nome del secondo candidato alla carica di vicepresidente;
    - `votiPerCandidatoPresidente1` – una variabile statica che conta i voti per il primo candidato presidente;
    - `votiPerCandidatoPresidente2` – una variabile statica che conta i voti per il secondo candidato presidente;
    - `votiPerCandidatoVicePresidente1` – una variabile statica che conta i voti per il primo candidato alla carica di vicepresidente;
    - `votiPerCandidatoVicePresidente2` – una variabile statica che conta i voti per il secondo candidato alla carica di vicepresidente;
    - `mioVotoPerPresidente` – un intero che contiene il voto di una singola persona per la carica di presidente (0 per nessuna scelta, 1 per il primo candidato, 2 per il secondo candidato);
    - `mioVotoPerVicePresidente` – un intero che contiene il voto di una singola persona per la carica di vicepresidente (0 per nessuna scelta, 1 per il primo candidato, 2 per il secondo candidato).

    Oltre a opportuni costruttori, `RegistratoreVoti` deve definire i seguenti metodi:
    - `setCandidatiPresidente(String nome1, String nome2)` – metodo statico per impostare i nomi dei candidati alla carica di presidente;
    - `setCandidatiVicePresidente(String nome1, String nome2)` – metodo statico per impostare i nomi dei candidati alla carica di vicepresidente;
    - `resettaVoti` – metodo statico che resetta il conteggio dei voti a 0;
    - `getVotiPresidenti` – metodo statico che restituisce una stringa contenente i voti per entrambi i candidati alla carica di presidente;
    - `getVotiVicePresidenti` – metodo statico che restituisce una stringa contenente i voti per entrambi i candidati alla carica di vicepresidente;
    - `restituisciEConfermaVoti` – un metodo di istanza che restituisce i voti di un individuo, li conferma e li registra;

- `getVoto(String nome1, String nome2)` – un metodo privato che restituisce una scelta di voto tra due candidati fatta da un individuo (0 per nessuna scelta, 1 per il primo candidato, 2 per il secondo candidato);
- `getVoti` – un metodo privato che restituisce una scelta di voto per i candidati di presidente e vicepresidente fatta da un individuo;
- `confermaVoti` – un metodo privato che mostra i voti di una persona e chiede al votante se è contento della scelta e restituisce `true` se questi a risposto sì e `false` se ha risposto no;
- `registraVoti` – un metodo privato che aggiunge i voti di un individuo alle variabili statiche appropriate.

Si crei un programma che gestisca un'elezione. I candidati a presidente sono Mauro e Leonardo. I candidati alla carica di vicepresidente sono Giovanni e Pietro. Si usi un ciclo per registrate i voti dei votanti. Si crei un nuovo oggetto `RegistratoreVoti` per ciascun votante. Dopo che sono stati collezionati tutti i voti presenta i risultati.

12. Si ripeta il Progetto 12 del Capitolo 8, ma si includano i costruttori.
13. Si ripeta il Progetto 12 del Capitolo 8 utilizzando un array per immagazzinare le valutazioni sui film anziché variabili distinte. Tutti i cambiamenti dovranno essere interni alla classe, così che il metodo `main` nella classe di prova funzioni esattamente allo stesso modo sia con la vecchia versione della classe `Film` che con la nuova.
14. La classe `LibroDelTempo` nel Listato 9.17 non è ancora completa. Completare la definizione di questa classe come descritto nel testo. In particolare, occorre aggiungere un costruttore di default, così come i metodi *set* e *get* che modificano e restituiscono ogni variabile di istanza e ogni variabile indicizzata di ogni istanza di array. Occorre, inoltre, sostituire il prototipo `setOre` con un metodo che prenda in input i valori dalla tastiera. Inoltre, si deve definire un metodo privato con due parametri `int` che visualizza il primo parametro lasciando esattamente il numero di spazi bianchi iniziali specificati dal secondo parametro. Questo consentirà di scrivere ogni elemento dell'array esattamente in quattro spazi, per esempio, e di conseguenza consentirà di visualizzare gli elementi dell'array in una struttura rettangolare. Si deve verificare che il metodo `main` nel Listato 9.17 funzioni correttamente con questi nuovi metodi. Inoltre, si scriva, a parte, un programma di test dei nuovi metodi. *Suggerimento*: per visualizzare un valore `int n` in un numero fissato di spazi, si utilizzi `Integer.toString(n)` che converte il numero in una stringa e consente, quindi, di lavorare con un valore di tipo stringa. Questo metodo è discusso nel Paragrafo 9.2.5.
15. Definire una classe `GiocoDelTris`. Un oggetto di tipo `GiocoDelTris` è una singola partita del gioco del tris. Memorizzare la griglia di gioco come un array bidimensionale di tipo base `char` formato da tre righe e tre colonne. Includere un metodo per aggiungere una mossa, per visualizzare la griglia di gioco, per indicare che turno è ("X" o "O"), per indicare se c'è un vincitore, per dire chi è il vincitore e per ricominciare con una nuova partita. Scrivere un metodo `main` per la classe che consenta a due giocatori di inserire le loro mosse a turno dalla stessa tastiera.

16. Il Sudoku è un gioco ampiamente diffuso basato sulla logica che utilizza un array di 9×9 caselle suddivise in 3×3 sotto-array. Il solutore deve riempire le caselle bianche inserendo numeri che vanno da 1 a 9, in modo che la cifra inserita non si ripeta né nella riga, né nella colonna e neanche nel sottogruppo cui appartiene la cifra. All'inizio alcune celle hanno già un valore e non possono essere modificate. Per esempio, la figura seguente rappresenta lo schema iniziale di un Sudoku:

1	2	3	4	9	7	8	6	5
4	5	9						
6	7	8						
3				1				
2					5			
9								
8								
7								
5			9					

Si crei una classe Sudoku che possiede i seguenti attributi:

- `scacchiera` – un array 9×9 di interi che rappresenta lo stato attuale del gioco e in cui gli zeri rappresentano le celle ancora non riempite;
- `inizio` – un array 9×9 di valori booleani che specifica quali elementi dell'array `scacchiera` possiedono un valore che non può essere cambiato;

e i seguenti metodi:

- `Sudoku` – un costruttore che crea un nuovo gioco in cui tutte le caselle sono vuote;
- `toString` – restituisce una stringa stampabile che rappresenta il gioco;
- `aggiungiaIniziali(riga, colonna, valore)` – aggiunge nella posizione specificata da `riga` e `colonna` il valore iniziale dato da `valore` che non può essere modificato;
- `aggiungiMossa(riga, colonna, valore)` – aggiunge nella posizione specificata da `riga` e `colonna` il valore specificato da `valore`. Tale valore può essere modificato;
- `verificaGioco()` – restituisce vero se i valori inseriti non violano le regole del gioco;
- `getValoreIn(riga, colonna)` – restituisce il valore contenuto nella posizione specificata da `riga` e `colonna`;
- `getValoriValidi(riga, colonna)` – restituisce un array monodimensionale di nove valori booleani, ognuno dei quali corrisponde a una cifra e risulta vero se la cifra può essere posta alla posizione specificata da `riga` e `colonna` senza violare le regole del gioco;
- `pieno` – restituisce vero se ogni cella possiede un valore;
- `reset` – imposta a zero tutte le celle che non contengono valori immutabili.

Scrivere un metodo `main` nella classe `Sudoku` che crea un'istanza di `Sudoku` e imposta la configurazione iniziale. Quindi utilizzare un ciclo per permettere all'utente di giocare. Visualizzare la configurazione corrente e chiedere all'utente una riga, una colonna e un valore. Aggiornare la scacchiera di gioco e visualizzarla. Si avvisi l'utente qualora la nuova configurazione non rispetti le regole del gioco. Visualizzare un messaggio quando il gioco è stato completato correttamente. In questo caso, sia `verificaGioco` sia `pieno` devono restituire `true`. Si deve dare all'utente la possibilità di riavviare il gioco e di visualizzare i valori che possono essere inseriti nelle celle.

Capitolo 10

Ereditarietà

OBIETTIVI

- Descrivere in generale l'ereditarietà.
- Definire e utilizzare le classi derivate in Java.

Questo capitolo tratta l'ereditarietà, uno dei concetti chiave della programmazione orientata agli oggetti. L'ereditarietà permette di definire una classe in una forma molto generale e, in un secondo momento, di utilizzarla come base di partenza per definire nuove classi, che sono specializzazioni della classe generale. Queste nuove classi ereditano i metodi e le variabili di istanza della classe generale e definiscono nuove variabili di istanza e nuovi metodi. Per questo motivo, l'ereditarietà rafforza il riutilizzo del software. Questo capitolo illustra l'ereditarietà in generale e la sua realizzazione in Java.

Prerequisiti

Per poter comprendere gli argomenti trattati in questo capitolo, occorre aver letto il materiare presentato nei Capitoli da 1 a 5, nei Capitoli 8 e 9. I Capitoli 6 e 7 non sono necessari.

10.1 Concetti di base sull'ereditarietà

Si supponga di dover definire una classe per rappresentare dei veicoli. Un veicolo è caratterizzato dall'avere definite le variabili di istanza per memorizzare il numero di ruote e il numero massimo di occupanti. Nella classe dovranno essere definiti anche i metodi *get* e *set*. Si immagini ora di definire una nuova classe per rappresentare delle automobili. La classe è caratterizzata dall'avere le stesse variabili di istanza e gli stessi metodi definiti nella classe veicolo. In più, la classe automobile dovrebbe aggiungere le variabili di istanza per rappresentare la quantità di carburante presente nel serbatoio e il numero di targa, più alcuni altri metodi. Invece di ripetere nella automobile le definizioni delle variabili di istanza e dei metodi nella classe veicolo, si può sfruttare il meccanismo dell'ereditarietà di Java in modo che la classe automobile erediti tutte le variabili di istanza e i metodi definiti dalla classe veicolo.

L'**ereditarietà** (*inheritance*) permette di definire una classe più generale e di definire in seguito classi specializzate che aggiungono nuovi dettagli alla classe generale. Questo permette di risparmiare molto lavoro, e quindi tempo, perché la classe specializzata *eredita* tutte le proprietà della classe generale e il programmatore deve solo realizzare le nuove caratteristiche.

Prima di presentare un esempio di ereditarietà realizzato in Java, occorre impostare uno scenario. Si definirà una semplice classe chiamata `Persona`. Questa classe, mostrata nel Listato 10.1, è così semplice da rappresentare una persona esclusivamente mediante un attributo, il suo nome. La classe `Persona` non ha molta utilità di per sé, ma sarà utilizzata per definire nuove classi. Molti dei metodi della classe `Persona` sono semplici. Per esempio, il metodo `haLoStessoNome` è simile ai metodi `equals` che sono stati visti nei capitoli precedenti, tranne per il fatto che considera come uguali le versioni maiuscole e minuscole di una lettera nei confronti fra nomi.

MyLab

LISTATO 10.1 La classe `Persona`.

```
public class Persona {
    private String nome;

    public Persona() {
        nome = "Ancora nessun nome";
    }

    public Persona(String nomeIniziale) {
        nome = nomeIniziale;
    }

    public void setNome(String nuovoNome) {
        nome = nuovoNome;
    }

    public String getNome() {
        return nome;
    }

    public void scriviOutput() {
        System.out.println("Nome: " + nome);
    }

    public boolean haLoStessoNome(Persona altraPersona) {
        return this.nome.equalsIgnoreCase(altraPersona.nome);
    }
}
```

10.1.1 Classi derivate

Si supponga di dover progettare un programma per l'archiviazione di informazioni di un'Università. Il sistema gestisce le schede informative degli studenti, dei docenti e di altro personale.

Esiste una gerarchia naturale con cui raggruppare questi tipi di schede: a prescindere dal fatto che sia uno studente o un docente, una scheda gestisce comunque informazioni su una persona. Le schede gestite dal sistema sono quindi tutte schede di persone. Gli studenti sono una sottoclasse di persone. Un'altra sottoclasse è rappresentata dai dipendenti, i quali includono sia i docenti, sia gli impiegati (lo staff). Gli studenti si dividono ulteriormente in due sottoclassi: gli studenti non ancora laureati e gli studenti della laurea triennale. Queste sottoclassi potrebbero essere ulteriormente suddivise in sottoclassi ancora più specifiche.

La Figura 10.1 descrive una parte di questa organizzazione gerarchica. Sebbene il programma possa non aver bisogno di classi che corrispondono a persone o a dipendenti, pensare in termini di tali classi può essere utile. Per esempio, tutte le persone possiedono un nome e i metodi d'inizializzazione, visualizzazione e modifica del nome saranno gli stessi per studenti, impiegati e docenti.

In Java, è possibile definire una classe chiamata `Persona` che include la definizione di tutte quelle variabili di istanza che rappresentano le proprietà comuni a tutte le sottoclassi di persone. La definizione della classe può inoltre contenere tutti i metodi che gestiscono le variabili di istanza definite nella classe `Persona`. Di fatto, la classe `Persona` è già stata definita nel Listato 10.1.

Il Listato 10.2 contiene la definizione di una classe che rappresenta gli studenti. Uno studente è una persona e pertanto si può definire la classe `Studente` come una **classe derivata**, o **sottoclasse**, della classe `Persona`. Una classe derivata è una classe definita aggiungendo variabili di istanza e metodi a una classe esistente. La classe esistente, dalla quale è stata definita la classe derivata, è chiamata **classe base**, o **superclasse**. Nell'esempio proposto, `Persona` è la classe base e `Studente` è la classe derivata. Nella definizione di `Studente` del Listato 10.2, ciò è specificato includendo l'espressione `extends Persona` sulla prima riga della definizione della classe. La definizione della classe `Studente` inizia quindi come segue:

```
public class Studente extends Persona
```

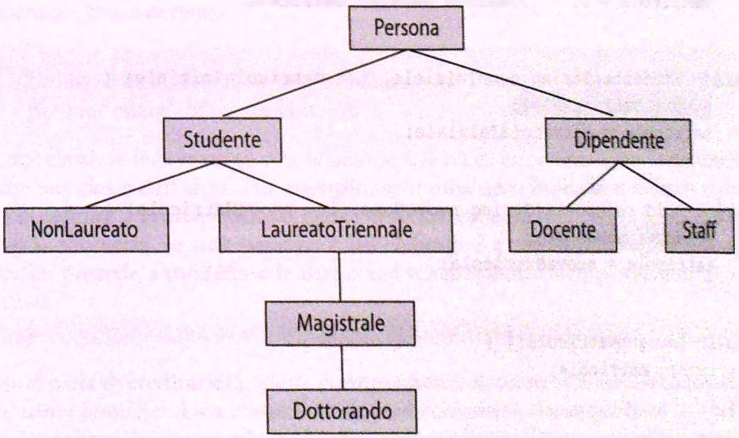


Figura 10.1 Una gerarchia di classi.

La classe `Studente`, come ogni altra classe derivata, **eredita** le variabili di istanza e i metodi pubblici della classe base che estende. Quando si definisce una classe derivata, si definiscono *esclusivamente* le variabili di istanza aggiuntive e i metodi aggiuntivi. Nell'esempio proposto, anche se uno studente è caratterizzato da un nome (e dai relativi metodi), la classe `Studente` *non* definisce tali proprietà, poiché le eredita dalla classe `Persona`. Quindi, la classe `Studente` possiede già tutte le variabili di istanza e tutti i metodi pubblici definiti nella classe `Persona`, senza doverli ridefinire. Per esempio, si immagini di istanziare un nuovo oggetto della classe `Studente` come segue:

```
Studente s = new Studente();
```

Il nome dell'oggetto `s` può essere modificato utilizzando il metodo `setNome` che la classe `Studente` eredita dalla classe `Persona`. Il metodo ereditato `setNome` può essere utilizzato come un qualsiasi altro metodo:

```
s.setNome("Stefano Rampoldi");
```

Una classe derivata, come `Studente`, può inoltre aggiungere nuove variabili di istanza e nuovi metodi a quelli ereditati dalla sua classe base. Per esempio, `Studente` definisce la variabile di istanza `matricola` e i metodi `reimposta`, `getMatricola`, `setMatricola`, `scriviOutput` ed `equals`, così come alcuni costruttori (si rimanda la discussione sui costruttori a quando sarà terminata la spiegazione inerente le altre parti di queste definizioni di classe). Il Listato 10.3 contiene un semplice programma dimostrativo per illustrare l'ereditarietà.

MyLab

LISTATO 10.2 Una classe derivata.

```
public class Studente extends Persona {
    private int matricola;

    public Studente() {
        super();
        matricola = 0;     //Ancora nessuna matricola
    }

    public Studente(String nomeIniziale, int matricolaIniziale) {
        super(nomeIniziale);
        matricola = matricolaIniziale;
    }

    public void reimposta(String nuovoNome, int nuovaMatricola) {
        setNome(nuovoNome);
        matricola = nuovaMatricola;
    }

    public int getMatricola() {
        return matricola;
    }

    public void setMatricola(int nuovaMatricola) {
        matricola = nuovaMatricola;
    }
```

`super` viene spiegato più avanti. Per il momento non occorre preoccuparsene.

```
    public void scriviOutput() {
        System.out.println("Nome: " + getNome());
        System.out.println("Matricola: " + matricola);
    }

    public boolean equals(Studente altroStudente) {
        return this.haLoStessoNome(altroStudente) &&
            (this.matricola == altroStudente.matricola);
    }
}
```

LISTATO 10.3 Una dimostrazione dell'ereditarietà utilizzando `Studente`.

MyLab

```
public class EreditarietaDemo {
    public static void main(String[] args) {
        Studente s = new Studente();
        s.setNome("Stefano Rampoldi");
        s.setMatricola(1234);
        s.scriviOutput();
    }
}
```

`setNome` è ereditato dalla classe `Persona`.

Esempio di output

```
Nome: Stefano Rampoldi
Matricola: 1234
```

In precedenza si è osservato che uno studente è una persona. Le classi `Studente` e `Persona` realizzano questa relazione del mondo reale, facendo sì che `Studente` abbia tutti i comportamenti di `Persona`. Questa relazione è nota come **relazione is-a** (letteralmente *è-un*). Si dovrebbe utilizzare l'ereditarietà solo se esiste una relazione *is-a* tra una classe e una potenziale classe derivata.

L'utilizzo dell'ereditarietà esclusivamente per modellare relazioni is-a

Date due classi, se non sussiste una relazione *is-a* tra di esse, non si usa l'ereditarietà per derivare una classe dall'altra. Per esempio, se si dovessero modellare elefanti e persone, prima di definire le corrispondenti classi, ci si dovrebbe domandare se un elefante è una persona o, viceversa, se una persona è un elefante. La risposta è ovviamente negativa e quindi si procede a modellare le due classi senza stabilire alcuna relazione gerarchica tra di esse.

Quando si parla di ereditarietà, viene comunemente adottata una terminologia che deriva dalle relazioni familiari. Una classe base è spesso chiamata **classe genitore** (o anche **classe padre**). Una classe derivata è allora chiamata **classe figlia**. Questo semplifica notevolmente il linguaggio dell'ereditarietà. Per esempio, si può dire che una classe figlia eredita le variabili di istanza e i metodi pubblici dalla sua classe genitore.

Questa analogia viene spesso portata anche un passo oltre. Una classe genitore di una classe che a sua volta è genitore di un'altra classe (ma la gerarchia si può estendere indefinitamente) è detta **classe antenato**. Se la classe A è un antenato della classe B, allora la classe B è chiamata **discendente** della classe A.

Membri ereditati

Una classe derivata include automaticamente tutte le variabili di istanza, le variabili statiche e tutti i metodi pubblici della classe base. I membri ottenuti dalla classe base si dicono ereditati. Le variabili di istanza, le variabili statiche e i metodi pubblici ereditati dalla classe base non sono dichiarati esplicitamente nella definizione della classe derivata, ma diventano automaticamente suoi membri. Esiste una sola eccezione a questa regola: come spiegato nel prossimo paragrafo, in una classe derivata è possibile modificare un metodo ereditato. Questa nuova definizione ridefinirà il comportamento del metodo solo nella classe derivata.

Classe derivata

Si definisce una classe derivata, o sottoclasse, partendo dalla definizione di un'altra classe già definita e aggiungendo (o modificando) i metodi e le variabili di istanza necessari. La classe di partenza è detta classe base o superclasse. La classe derivata eredita tutte le variabili di istanza, le variabili statiche e tutti i metodi pubblici dalla classe base e può aggiungere variabili proprie e metodi propri.

Sintassi

```
public class nome_della_classe_derivata extends nome_della_classe_base {
    dichiarazione_di_variabili_aggiuntive
    definizoni_di_metodi_aggiuntivi_e_di_metodi_modificati
}
```

Esempio

Vedere il Listato 10.2.

Come si vedrà nel prossimo paragrafo, i metodi modificati prendono il nome di metodi ridefiniti.

10.1.2 Metodi ridefiniti (overriding)

La classe `Studente` del Listato 10.2 definisce il metodo `scriviOutput`, senza parametri. Ma anche la classe `Persona` definisce un metodo con lo stesso nome e senza parametri. Se la classe `Studente` avesse ereditato il metodo `scriviOutput` dalla classe base `Persona`, `Studente` si ritroverebbe con due metodi `scriviOutput`, entrambi senza parametri. In altre parole, la classe `Studente` avrebbe due metodi con la stessa firma, cosa impossibile in Java, come sottolineato nei capitoli precedenti. Se una classe derivata definisce un metodo che ha lo stesso nome, gli stessi parametri (in termini di tipo, ordine e numero) e anche lo stesso tipo di ritorno di un metodo della classe base, il metodo della classe derivata **ridefinisce** il metodo presente nella classe base (*overriding*). In altre parole,

per gli oggetti creati dalla classe derivata viene usata la definizione del metodo presente nella classe derivata. Esiste una sola eccezione a questa regola, che sarà descritta nel prossimo paragrafo.

Per esempio, l'invocazione:

```
s.scriviOutput();
```

del Listato 10.3 userà la definizione di `scriviOutput` nella classe `Studente`, non la definizione nella classe `Persona`, poiché `s` è un oggetto della classe `Studente`.

Quando si ridefinisce un metodo, si può cambiare a piacere il corpo della sua definizione, ma non si può modificare l'intestazione. Esiste un solo caso, discusso nel prossimo paragrafo, in cui è possibile modificare il tipo di ritorno.

10.1.3 Cambiare il tipo di ritorno di un metodo ridefinito

In una classe derivata, quando si ridefinisce un metodo ereditato dalla classe base, in generale *non* è possibile modificare il tipo di ritorno. Per esempio, non è possibile cambiare un metodo `void` in un metodo che restituisce un valore (di un qualsiasi tipo); non è possibile cambiare un metodo che restituisce un valore (di un qualsiasi tipo) in un metodo `void`. L'eccezione a questa regola è la seguente: se il tipo di ritorno è una classe, il metodo ridefinito può restituire una qualsiasi delle sue classi derivate. Per esempio, se nella classe base un metodo restituisce il tipo `Persona` (Listato 10.1), lo stesso metodo ridefinito in una sua classe derivata, può restituire il tipo `Studente` (Listato 10.2) o qualsiasi altra classe derivata direttamente (figlia) o indirettamente (discendente) da `Persona`. Il tipo di ritorno così modificato prende il nome di **tipo di ritorno covariante** ed è stato introdotto a partire dalla versione 5.0 di Java; il tipo di ritorno covariante non poteva essere utilizzato nelle versioni precedenti di Java. Di seguito un semplice esempio.

Si supponga che una classe includa le seguenti definizioni:

```
public class ClasseBase {
    ...
    public Persona getIndividuo(int identificatore) {
        ...
```

In questo caso, la classe derivata può lecitamente includere la seguente dichiarazione:

```
public class ClasseDerivata extends ClasseBase {
    ...
    public Studente getIndividuo(int identificatore) {
        ...
```

La ridefinizione del metodo `getIndividuo` in `ClasseDerivata` cambia il tipo di ritorno da `Persona` a `Studente`.

È importante notare che la ridefinizione del tipo di ritorno, come quella da `Persona` a `Studente` nell'esempio precedente, non introduce un tipo di ritorno più restrittivo del tipo di ritorno dal metodo dichiarato nella classe base. Infatti, uno `Studente` è una `Persona` con alcune proprietà aggiuntive. Qualsiasi frammento di codice scritto supponendo che il metodo `getIndividuo` della classe base restituisca un valore di tipo `Persona` sarà corretto anche per il metodo `getIndividuo` ridefinito nella classe derivata e che restituisce un valore di tipo `Studente`. Questo è vero perché ogni `Studente` è anche una `Persona`.

10.1.4 Cambiare i modificatori d'accesso di un metodo ridefinito

Un metodo dichiarato come privato nella classe base può essere ridefinito come pubblico in una classe derivata (in generale il modificatore d'accesso può essere ridefinito in un qualsiasi modo che renda più permissivo l'accesso). Per esempio, se una classe base comprendesse il seguente metodo:

```
private void faiQualcosa()
```

tale metodo potrebbe essere ridefinito nel seguente modo in una classe derivata:

```
public void faiQualcosa()
```

Si noti che non è possibile restringere i permessi d'accesso nella classe derivata. Quindi, mentre è possibile cambiare un modificatore d'accesso da `private` a `public`, non è possibile cambiarlo da `public` a `private`. Questa regola deve essere rispettata perché il codice scritto per i metodi nella classe base deve funzionare anche per i metodi nelle classi derivate. È possibile invocare un metodo pubblico in tutti i punti del codice dove veniva invocato un metodo privato, ma non è possibile invocare un metodo privato dove prima veniva invocato un metodo pubblico.

Overriding delle definizioni dei metodi

In una classe derivata, se si include la ridefinizione di un metodo che ha lo stesso nome, gli stessi parametri (in termini di tipo, ordine e numero) e lo stesso tipo di ritorno di un metodo già definito nella classe base, questa nuova definizione sostituirà la vecchia definizione per le invocazioni del metodo ricevute dagli oggetti della classe derivata.

Quando si ridefinisce un metodo, non si può cambiare il tipo di ritorno, tranne per il tipo classe: si può usare un discendente della classe di partenza.

È possibile, infine, cambiare il modificatore d'accesso, a patto di non restringere la visibilità. In altre parole il nuovo modificatore d'accesso può solo rendere più permissivo l'accesso al metodo.

10.1.5 Overriding vs. overloading

Non si deve confondere l'*overriding* di un metodo con l'*overloading* di metodo. Quando si effettua l'*overriding* della definizione di un metodo, la nuova definizione del metodo nella classe derivata ha lo stesso nome, lo stesso tipo di ritorno (a parte l'eccezione precedentemente trattata) e gli stessi parametri in termini di tipo, ordine e numero. Se invece il metodo nella classe derivata avesse lo stesso nome e lo stesso tipo di ritorno, ma un numero differente di parametri o anche un solo parametro di un tipo differente dal metodo nella classe base, si sarebbe di fronte a un caso di *overloading*. In questa situazione, la classe derivata avrebbe entrambi i metodi. Per esempio, si supponga di aggiungere il seguente metodo alla definizione della classe `Studente` del Listato 10.2:

```
public String getNome(String titolo) {
    return titolo + getNome();
}
```

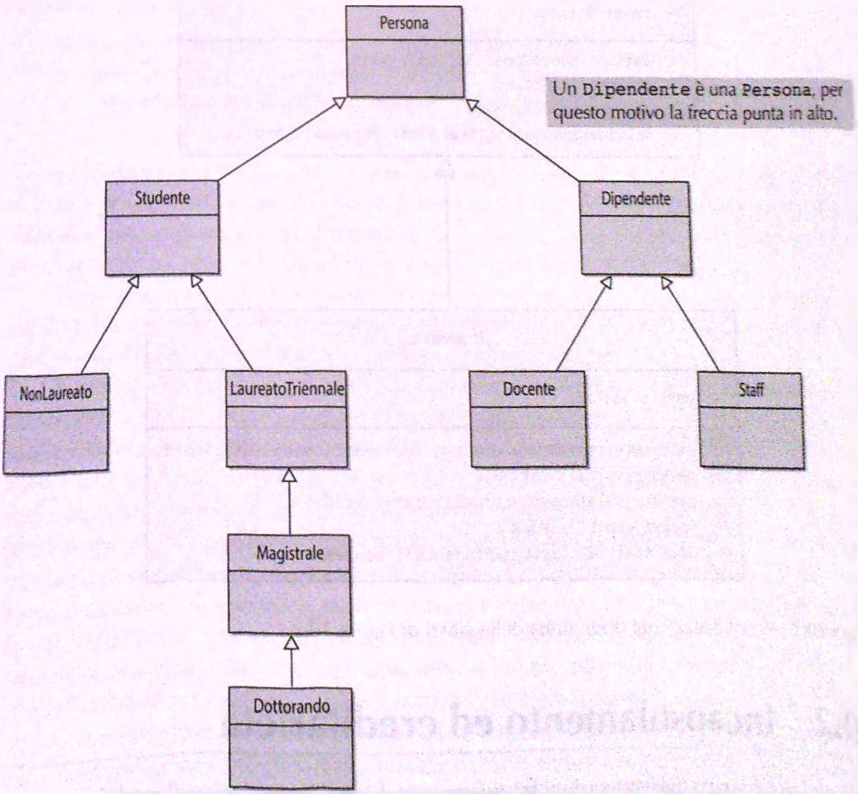
In questo caso, la classe `Studente` avrebbe due metodi `getNome`: erediterebbe dalla classe base `Persona` (Listato 10.1) il metodo `getNome` senza parametri e in più avrebbe anche il metodo `getNome` con un parametro. Ciò avviene perché i due metodi `getNome` hanno un numero di parametri differente e sono, quindi, il risultato di un *overloading*.

Per distinguere *overloading* e *overriding*, si ricordi che: *overloading* aggiunge un "carico" (*load*) sul nome di un metodo utilizzandolo per un'ulteriore attività, mentre l'*overriding* sostituisce una definizione del metodo.

10.1.6 Ereditarietà nei diagrammi UML

La Figura 10.2 mostra una porzione della gerarchia delle classi fornita nella Figura 10.1, ma usa la notazione UML. Si sottolinea che i diagrammi delle classi presentati nella Figura 10.2 sono incompleti: lo scopo è esclusivamente quello di illustrare la notazione UML che rappresenta la relazione di ereditarietà fra le classi.

L'unica differenza importante tra la notazione della Figura 10.2 e quella della Figura 10.1 consiste nel fatto che le linee che indicano una relazione di ereditarietà fra classi nella Figura 10.2 sono delle frecce vuote. Si noti che le frecce puntano dalla classe derivata alla classe base. Queste frecce mostrano la relazione *is-a* (letteralmente "è-un"). Per esempio, uno `Studente` è una (*is-a*) `Persona`. In termini Java, un oggetto di tipo `Studente` è anche di tipo `Persona`.



Le frecce sono utili anche per individuare la posizione delle definizioni dei metodi, vale a dire in quali classi sono definiti. Se si sta cercando la definizione di un metodo per una certa classe, le frecce mostrano il percorso che il programmatore (o il computer) dovrebbe seguire. Per esempio, se si sta cercando la definizione di un metodo utilizzato da un oggetto della classe `NonLaureato`, si guarda prima nella definizione della classe `NonLaureato`; se non c'è, si guarda nella definizione di `Studente`; se non c'è si guarda nella definizione della classe `Persona`.

La Figura 10.3 mostra il dettaglio completo di due classi in una gerarchia di ereditarietà: `Persona` e una delle sue classi derivate, `Studente`. Si supponga che `s` sia un oggetto della classe `Studente`. Il diagramma nella Figura 10.3 mostra che la definizione del metodo `scriviOutput` nell'invocazione:

```
s.scriviOutput();
```

si trova nella classe `Studente`, ma che la definizione di `setNome` in:

```
s.setNome("Luca Studente");
```

è nella classe `Persona`.

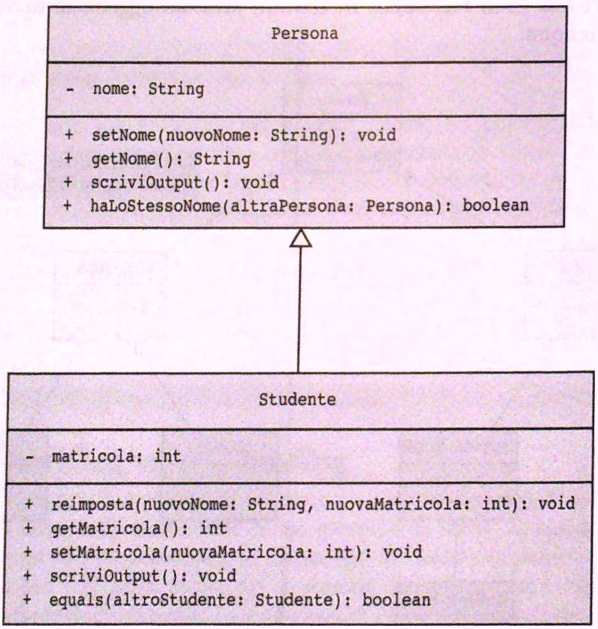


Figura 10.3 Alcuni dettagli del diagramma delle classi di Figura 10.2.

10.2 Incapsulamento ed ereditarietà

Questo paragrafo tratta delle interazioni fra *information-hiding*, in particolare il modificatore d'accesso `private`, ed ereditarietà.

10.2.1 Uso delle variabili di istanza private della classe base

Uno dei membri che un oggetto di tipo `Studente` eredita dalla classe `Persona` è la variabile di istanza nome. Per esempio, il seguente frammento di codice assegna alla variabile di istanza nome dell'oggetto `gio` di tipo `Studente` il valore `"Giovanni"`:

```
gio.setNome("Giovanni");
```

Bisogna sempre gestire con prudenza le variabili di istanza come nome. La variabile di istanza nome della classe `Studente` è stata ereditata dalla classe `Persona`, ma si tratta di una variabile di istanza privata della classe `Persona`. Questo significa che si può accedere alla variabile di istanza nome solamente attraverso i metodi definiti nella classe `Persona`. Le variabili di istanza (e i metodi) che sono dichiarati privati in una classe base non sono accessibili per nome dai metodi di nessun'altra classe, incluse le classi derivate.

Si consideri, per esempio, la seguente definizione di metodo estratta dalla classe `Studente` definita nel Listato 10.2:

```
public void scriviOutput() {
    System.out.println("Nome: " + getNome());
    System.out.println("Matricola: " + matricola);
}
```

Sarebbe lecito chiedersi perché si è usato il metodo `getNome()` invece di implementare il metodo come segue:

```
public void scriviOutput() {    //versione errata
    System.out.println("Nome: " + nome);
    System.out.println("Matricola: " + matricola);
}
```

Come specificato dal commento, questa seconda versione è errata. nome è una variabile di istanza privata della classe `Persona` e, benché la classe derivata `Studente` la erediti da `Persona`, non può accedervi direttamente. È necessario utilizzare un metodo pubblico per riuscire ad accedere alla variabile di istanza nome; per esempio `getNome`.

Nella definizione di una classe derivata, non è possibile accedere per nome alle variabili di istanza private che vengono ereditate. Bisogna utilizzare i metodi *get* e *set* (per esempio `getNome()` e `setNome()`) definiti nella classe base.

Il principio in base al quale dalla definizione di un metodo di una classe derivata non sia possibile accedere a una variabile di istanza privata di una classe base potrebbe apparire errato. Dopotutto, se uno studente volesse cambiare nome, nessuno direbbe "Mi spiace, nome è una variabile di istanza privata della classe `Persona`". In fondo, uno studente è anche una persona e ha quindi tutti i diritti di cambiare nome. Questo è vero anche in Java: un oggetto di tipo `Studente` è anche un oggetto di tipo `Persona`. Nonostante questo, l'accesso alle variabili di istanza e ai metodi privati deve rispettare le regole precedentemente descritte, altrimenti la correttezza del programma potrebbe essere compromessa. Se una variabile di istanza privata di una classe fosse accessibile dalla definizione di un metodo di una classe derivata, ogniqualvolta si volesse accedere a una variabile di istanza privata basterebbe creare una classe derivata e accedervi da un metodo di questa classe. Questo vorrebbe dire rendere accessibili (a chiunque voglia fare lo sforzo di definire una classe derivata) le variabili di istanza private definite in una classe. Questo scenario illustra la problematica, ma il reale problema che si verrebbe a creare è l'involontaria introduzione

di errori. Se le variabili di istanza private di una classe fossero accessibili dalle definizioni dei metodi delle classi derivate, il programmatore potrebbe modificarne il valore inavvertitamente o in modo inappropriato (si ricorda che i metodi *get* e *set* proteggono le variabili di istanza private da cambiamenti inappropriati).

Si discuterà come aggirare la limitazione d'accesso alle variabili di istanza private nel Paragrafo 10.2.3.

Le variabili di istanza private non sono direttamente accessibili nelle classi derivate

Una classe derivata non può accedere direttamente alle variabili di istanza private della sua classe base. La classe derivata conosce infatti il comportamento pubblico della classe base, ma si presuppone che non conosca (e che non vi sia interessata) il modo in cui la classe base gestisce i propri dati. Tuttavia, la classe derivata potrebbe ereditare metodi pubblici che contengono riferimenti alle variabili private.

10.2.2 I metodi privati non sono accessibili

Come si è fatto notare nel paragrafo precedente, una variabile di istanza o un metodo privato di una classe base non sono direttamente accessibili al di fuori della definizione della classe base, *incluse le classi derivate*. Relativamente alle modalità d'accesso, i metodi privati di una classe base si comportano come le variabili di istanza private. Nel caso dei metodi, però, la restrizione è più forte. Si può accedere a una variabile privata attraverso i metodi *get* e *set*, mentre i metodi privati sono del tutto indisponibili: è come se i metodi privati non venissero ereditati. In realtà, i metodi privati di una classe base possono essere indirettamente disponibili nella classe derivata. Se un metodo privato viene utilizzato nella definizione di un metodo pubblico della classe base, il metodo pubblico può essere invocato nella classe derivata, o in qualsiasi altra classe, che quindi accede indirettamente al metodo privato.

Questo non dovrebbe rappresentare un problema. I metodi privati dovrebbero essere utilizzati come metodi di supporto ad altri metodi, quindi il loro utilizzo dovrebbe essere limitato alla classe nella quale sono definiti. Se si desidera utilizzare nei metodi delle classi derivate un metodo di supporto definito nella classe base, significa che non è un semplice metodo di supporto, ma deve essere dichiarato pubblico.

I metodi privati non sono direttamente accessibili nelle classi derivate

Una classe derivata non può invocare metodi privati della sua classe base. Tuttavia, la classe derivata può chiamare metodi pubblici che a loro volta chiamano metodi privati, a patto che entrambi i metodi siano stati definiti nella classe base.

10.2.3 Modalità d'accesso `protected` (opzionale)

Come visto in precedenza, non è possibile accedere (per nome) alle variabili di istanza o ai metodi privati della classe base dalla definizione dei metodi delle classi derivate. Esistono due tipi di modificatori per le variabili di istanza e per i metodi che permettono l'accesso per nome dalle classi derivate (a parte `public`, ovviamente). I due modificatori sono `protected` (protetto), che permette sempre l'accesso alle classi derivate, e `package`, che permette l'accesso se la classe derivata appartiene allo stesso package della classe base. Quest'ultimo non sarà trattato in questo testo.

Un metodo (o una variabile di istanza) dichiarato con modificatore d'accesso `protected` (invece di `public` o `private`) è accessibile per nome dalla classe alla quale appartiene, dalle classi derivate dalla classe a cui appartiene e da qualsiasi classe (anche non derivata) contenuta nello stesso package della classe alla quale appartiene. I metodi e le variabili di istanza `protected` non sono invece accessibili per nome da qualsiasi altra classe che non appartenga alla casistica appena riportata. Quindi, se una variabile di istanza è dichiarata come `protected` in una classe `Padre` e la classe `Figlio` è derivata dalla classe `Padre`, la variabile di istanza è accessibile da qualsiasi definizione di metodo della classe `Figlio`. Invece per le classi che non sono né nello stesso package della classe `Padre` né estendono la classe `Padre`, la variabile di istanza `protected` è equivalente a una variabile di istanza `private`.

Si consideri la classe `Studente` derivata dalla classe base `Persona`. È necessario utilizzare i metodi *get* e *set* per gestire le variabili di istanza ereditate da `Persona` poiché sono state dichiarate private. A titolo esemplificativo, si riporta la definizione del metodo `scriviOutput` della classe `Studente`:

```
public void scriviOutput() {
    System.out.println("Nome: " + getNome());
    System.out.println("Matricola: " + matricola);
}
```

Se la variabile di istanza nome fosse specificata come `protected` nella classe `Persona`:

```
public class Persona {
    protected String nome;
    ...
```

la definizione del metodo `scriviOutput` nella classe `Studente` potrebbe essere semplificata come segue:

```
public void scriviOutput() {    //corretto se nome è dichiarato
                                //protected nella classe Persona
    System.out.println("Nome: " + nome);
    System.out.println("Matricola: " + matricola);
}
```

Il modificatore d'accesso `protected`

Un metodo (o una variabile di istanza) dichiarato con modificatore d'accesso `protected` è accessibile per nome dalla classe cui appartiene, dalle classi derivate dalla classe cui appartiene e da qualsiasi classe nello stesso package della classe cui appartiene.

È meglio evitare di utilizzare il modificatore d'accesso `protected` per le variabili di istanza

Il modificatore d'accesso `protected` garantisce un basso livello di protezione rispetto al modificatore d'accesso `private` perché qualsiasi programmatore può accedere direttamente a un membro `protected` definendo una classe derivata. Per tale motivo l'utilizzo del modificatore `protected` è spesso sconsigliato e le variabili di istanza non dovrebbero essere specificate come `protected`. Solo in rare occasioni si potrebbe voler dichiarare un metodo come `protected`.

10.3 Programmare con l'ereditarietà

Questo paragrafo presenta alcune tecniche base di programmazione utili quando si definiscono o si utilizzano le classi derivate.

10.3.1 Costruttori nelle classi derivate

Una classe derivata, come la classe `Studente` del Listato 10.2, ha i suoi costruttori. Essa non eredita alcun costruttore dalla classe base. Una classe base, come `Persona`, ha anch'essa i suoi costruttori. Nella definizione di un costruttore per la classe derivata, la prima tipica azione è di invocare un costruttore della classe base.

Si consideri, per esempio, di dover definire un costruttore per la classe `Studente`. Poiché il compito principale di un costruttore consiste nell'inizializzare l'oggetto (e quindi le sue variabili di istanza), un costruttore di `Studente` dovrebbe inizializzare la variabile di istanza `nome` (oltre a `matricola`). Dato che la variabile di istanza `nome` è definita in `Persona`, è inizializzata dai costruttori della classe base `Persona`. Occorre, quindi, delegare a uno dei costruttori di `Persona` l'inizializzazione di `nome`.

Si consideri la seguente definizione di costruttore nella classe derivata `Studente` (Listato 10.2):

```
public Studente(String nomeIniziale, int matricolaIniziale) {
    super(nomeIniziale);
    matricola = matricolaIniziale;
}
```

Questo costruttore utilizza la parola riservata `super` come un nome di metodo per invocare un costruttore della classe base. Sebbene la classe base `Persona` definisca due costruttori, l'invocazione:

```
super(nomeIniziale);
```

rimanda al costruttore della classe `Persona` che ha un parametro di tipo stringa. Si noti che si utilizza la parola chiave `super`, non il nome del costruttore. Cioè, non si utilizza:

```
Persona(nomeIniziale);     //ILLEGALE
```

FAQ Come si può ricordare che super invoca un costruttore della classe base e non della classe derivata?

Una classe base è detta anche superclasse. Così super invoca un costruttore nella superclasse della classe.

L'uso di super implica alcuni dettagli: super deve essere sempre la prima azione specificata nella definizione di un costruttore, non può essere specificato più avanti nella definizione del costruttore. Se in ogni costruttore della classe derivata non si include un'invocazione esplicita al costruttore della classe base, Java includerà automaticamente un'invocazione al costruttore di default della classe base. Per esempio, la definizione del costruttore di default per la classe Studente data nel Listato 10.2:

```
public Studente() {
    super();
    matricola = 0;     //ancora nessun numero
}
```

è completamente equivalente alla seguente definizione:

```
public Studente() {
    matricola = 0;     //ancora nessun numero
}
```

Chiamare un costruttore della classe base

Quando si definisce un costruttore per una classe derivata, si può usare super come un nome per il costruttore della classe base. Qualsiasi invocazione a super deve essere la prima azione eseguita dal costruttore.

Esempio

```
public Studente(String nomeIniziale, int matricolaIniziale) {
    super(nomeIniziale);
    matricola = matricolaIniziale;
}
```

Omettere un'invocazione a super in un costruttore

Quando si omette un'invocazione al costruttore della classe base in un costruttore di una classe derivata, il costruttore di default della classe base è invocato come prima azione del costruttore della classe derivata. Questo costruttore di default, senza parametri, potrebbe non essere quello che doveva essere invocato. Di conseguenza, spesso è opportuno esplicitare la chiamata al costruttore della classe base in modo tale da scegliere il costruttore più idoneo.

Per esempio, omettere `super(nomeIniziale)` dal secondo costruttore nella classe `Studente`, causerebbe l'invocazione del costruttore di default di `Persona`. Quest'azione imposterebbe il nome dello studente a "`Ancora nessun nome`" invece della stringa `nomeIniziale`.

Omettendo un'invocazione a `super` in un costruttore si ottengono errori di compilazione

Se la classe base non ha definito il costruttore di default e se si omette l'invocazione a uno dei costruttori della classe base nella definizione di un costruttore della classe derivata, si avrà un errore in compilazione. Infatti, `super()` non esiste nella classe base.

10.3.2 Ancora sul metodo `this`

Un'altra azione comune quando si definisce un costruttore consiste nell'invocare un altro costruttore della stessa classe. Il Capitolo 9 ha introdotto l'argomento spiegando l'utilizzo della parola chiave `this`. Ora che si conosce `super`, è chiaro che si possono utilizzare `this` e `super` in modi simili.

Il costruttore di default nella classe `Persona` (Listato 10.1) può essere riveduto in modo tale che invochi un altro costruttore definito nella classe stessa utilizzando `this`. La nuova definizione è la seguente:

```
public Persona() {
    this("Ancora nessun nome");
}
```

In questo modo il costruttore di default invoca il costruttore:

```
public Persona(String nomeIniziale) {
    nome = nomeIniziale;
}
```

impostando pertanto la variabile di istanza `nome` con la stringa "`Ancora nessun nome`".

Allo stesso modo di `super`, ogni utilizzo di `this` deve essere la prima azione nella definizione di un costruttore. Così, la definizione di un costruttore non può contenere sia un'invocazione che utilizza `super` sia un'invocazione che utilizza `this`. Cosa occorre fare se si vogliono includere entrambe le invocazioni? Si utilizza `this` per invocare un costruttore che ha `super` come sua prima azione.

`this` e `super` all'interno di un costruttore

Quando è usato in un costruttore, `this` invoca un costruttore della stessa classe, mentre `super` invoca un costruttore della classe base.

10.3.3 Invocare un metodo ridefinito

Si è appena visto come un costruttore di una classe derivata può utilizzare `super` come un nome per un costruttore della classe base. Un metodo di una classe derivata che ridefinisce un metodo nella classe base può utilizzare `super` per invocare il metodo ridefinito, ma in modo leggermente differente.

Per esempio, si consideri il metodo `scriviOutput` della classe `Studente` nel Listato 10.2. Questo contiene l'istruzione:

```
System.out.println("Nome: " + getNome());
```

per visualizzare il nome dello `Studente`. Una maniera alternativa per ottenere lo stesso risultato dell'istruzione sopra riportata, consiste nell'invocare il metodo `scriviOutput` della classe `Persona` del Listato 10.1, il quale mostra il nome della persona. L'unico problema è che se si utilizza il nome del metodo `scriviOutput` nella classe `Studente`, verrà invocato proprio il metodo `scriviOutput` della classe `Studente`. C'è bisogno di un modo per richiamare `scriviOutput` così come è definito nella classe base. Il modo di dire ciò è `super.scriviOutput()`. Di conseguenza, una definizione alternativa del metodo `scriviOutput` per la classe `Studente` è la seguente:

```
public void scriviOutput() {
    super.scriviOutput();     //Visualizza il nome
    System.out.println("Matricola: " + matricola);
}
```

Se si sostituisce la definizione di `scriviOutput` nella definizione di `Studente` (Listato 10.2) con la definizione precedente, la classe `Studente` si comporterà esattamente come prima.

Invocare un metodo ridefinito

Nella definizione di un metodo di una classe derivata si può invocare un metodo ridefinito della classe base facendolo precedere da `super` e un punto.

Sintassi

```
super.nome_del_metodo_ridefinito(elenco_argomenti)
```

Esempio

```
public void scriviOutput() {
    super.scriviOutput();     //Invoca scriviOutput nella classe base
    System.out.println("Matricola: " + matricola);
}
```

MyLab

Video 10.1 Definire classi sfruttando l'ereditarietà

ESEMPIO DI PROGRAMMAZIONE
UNA CLASSE DERIVATA DI UNA CLASSE DERIVATA

Si può definire una classe derivata da una classe derivata. Per esempio, si è derivata la classe `Studente` (Listato 10.2) dalla classe `Persona` (Listato 10.1). Ora si deriverà

una classe `NonLaureato` dalla classe `Studente`, come mostrato nel Listato 10.4. La Figura 10.4 contiene un digramma UML che mostra le relazioni tra le classi `Persona`, `Studente` e `NonLaureato`.

Un oggetto della classe `NonLaureato` possiede tutte le variabili di istanza e i metodi pubblici della classe `Studente`. Ma `Studente` è una classe derivata di `Persona`. Questo significa che un oggetto della classe `NonLaureato` possiede anche tutte le variabili di istanza e i metodi pubblici della classe `Persona`. Sebbene un oggetto della classe `NonLaureato` non erediti direttamente le variabili di istanza `nome` e `matricola`, in quanto private, può accedervi utilizzando i corrispondenti metodi ereditati *get* e *set*. Infatti, le classi `Studente` e `NonLaureato`, come qualsiasi altra classe derivata da ognuna di queste, riutilizzano il codice fornito nella definizione della classe `Persona` in quanto ereditano tutti i metodi pubblici della classe `Persona`.

MyLab

LISTATO 10.4 Una classe derivata di una classe derivata.

```
public class NonLaureato extends Studente {
    private int annoDiCorso;     //1 per primo anno, 2 per secondo anno,
                                 //3 per terzo anno, o 4 per fuori corso.
    public NonLaureato() {
        super();
        annoDiCorso = 1;
    }

    public NonLaureato(String nomeIniziale,
                       int matricolaIniziale, int annoDiCorsoIniziale) {
        super(nomeIniziale, matricolaIniziale);
        //Verifica 1 <= annoDiCorsoIniziale <= 4
        setAnnoDiCorso(annoDiCorsoIniziale);
    }

    public void reimposta(String nuovoNome, int nuovaMatricola,
                          int nuovoAnnoDiCorso) {
        reimposta(nuovoNome, nuovaMatricola);     //reimposta di Studente
        //Verifica 1 <= nuovoAnnoDiCorso <= 4
        setAnnoDiCorso(nuovoAnnoDiCorso);
    }

    public int getAnnoDiCorso() {
        return annoDiCorso;
    }

    public void setAnnoDiCorso(int nuovoAnnoDiCorso) {
        if ((1 <= nuovoAnnoDiCorso) && (nuovoAnnoDiCorso <= 4))
            annoDiCorso = nuovoAnnoDiCorso;
        else {
            System.out.println("Anno di corso illegale!");
            System.exit(0);
        }
    }
}
```

```
    public void scriviOutput() {
        super.scriviOutput();
        System.out.println("Anno di corso: " + annoDiCorso);
    }

    public boolean equals(NonLaureato altroNonLaureato) {
        return equals((Studente)altroNonLaureato) &&
            (this.annoDiCorso == altroNonLaureato.annoDiCorso);
    }
}
```

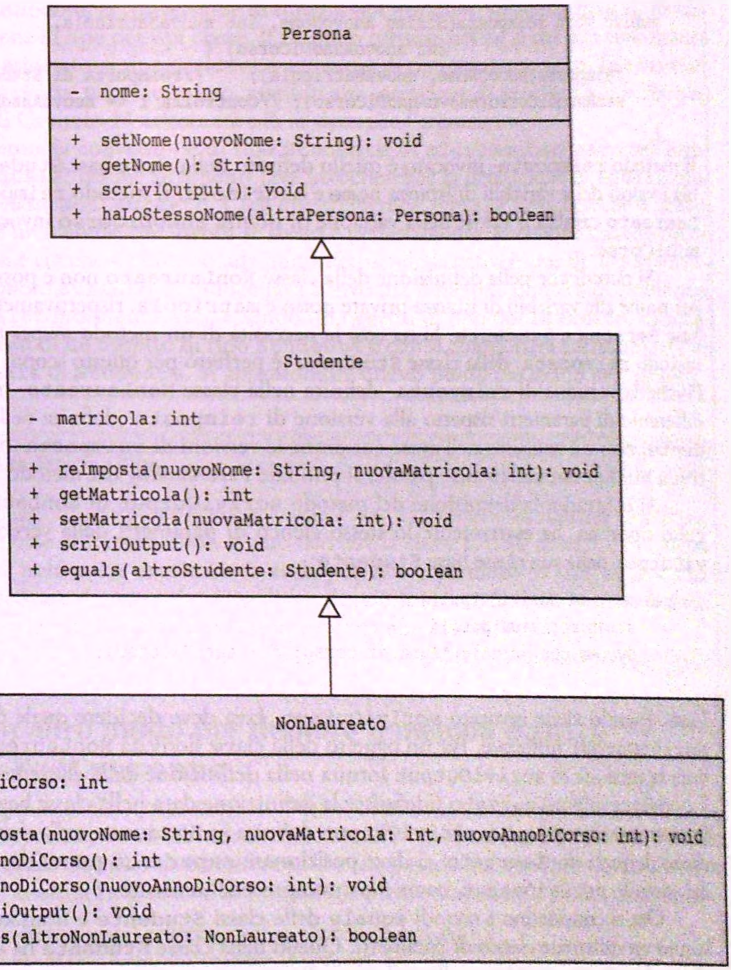


Figura 10.4 Ulteriori dettagli del diagramma delle classi di Figura 10.2.

Ogni costruttore nella classe NonLaureato inizia con un'invocazione a super, che, in questo contesto, rappresenta un costruttore della classe base Studente. Tuttavia anche i costruttori della classe Studente iniziano con un'invocazione a super, che, in questo caso, rappresentano un costruttore della classe base Persona. Così, quando si utilizza new per invocare un costruttore in NonLaureato, vengono invocati, nell'ordine, i costruttori di Persona e Studente e solo dopo vengono eseguite le istruzioni che seguono super nel costruttore di NonLaureato.

Le classi Studente e NonLaureato definiscono entrambe il metodo reimposta. In Studente, reimposta ha due parametri, mentre in NonLaureato reimposta ha tre parametri, così reimposta è frutto di un *overloading*. La definizione del metodo reimposta nella classe NonLaureato, qui riportata, inizia con l'invocare reimposta passando solo due argomenti:

```
public void reimposta(String nuovoNome, int nuovaMatricola,
                      int nuovoAnnoDiCorso) {
    reimposta(nuovoNome, nuovaMatricola);    //reimposta di Studente
    setAnnoDiCorso(nuovoAnnoDiCorso); //Controlla 1 <= nuovoAnnoDiCorso <= 4
}
```

Il metodo reimposta invocato è quello definito nella classe base Studente che cambia i valori delle variabili di istanza nome e matricola. Il metodo reimposta di NonLaureato cambia il valore della variabile di istanza annoDiCorso invocando setAnnoDiCorso.

Si ricordi che nella definizione della classe NonLaureato non è possibile accedere per nome alle variabili di istanza private nome e matricola, rispettivamente delle classi base Persona e Studente. Si ha così la necessità di un metodo *set* per modificarle. Il metodo reimposta della classe Studente è perfetto per questo scopo.

Poiché la versione di reimposta definita nella classe NonLaureato ha un numero differente di parametri rispetto alla versione di reimposta definita nella classe Studente, non c'è conflitto nell'avere entrambe le versioni di reimposta nella classe derivata NonLaureato. In altre parole, si può fare l'*overloading* del metodo reimposta.

Al contrario, la definizione del metodo scriviOutput in NonLaureato, di seguito riportata, ha esattamente lo stesso elenco di parametri della versione di scriviOutput nella sua classe base Studente:

```
public void scriviOutput() {
    super.scriviOutput();
    System.out.println("Anno di corso: " + annoDiCorso);
}
```

Così, quando viene invocato scriviOutput, Java deve decidere quale definizione di scriviOutput utilizzare. Per un oggetto della classe derivata NonLaureato, Java utilizza la versione di scriviOutput fornita nella definizione della classe NonLaureato. La versione in NonLaureato ridefinisce la definizione data nella classe base Studente. Per invocare la versione di scriviOutput definita in Studente nella definizione della classe derivata NonLaureato, si deve posizionare super e un punto davanti al nome del metodo scriviOutput, come mostrato precedentemente.

Ora si considerino i metodi equals delle classi Studente e NonLaureato. Essi hanno un differente elenco di parametri. Quello nella classe Studente ha un parametro di tipo Studente, mentre quello nella classe NonLaureato ha un parametro di tipo NonLaureato. Questi metodi hanno lo stesso numero di parametri, cioè uno, ma quell'unico

parametro è di tipo differente in ognuna delle due definizioni. Si ricordi che una differenza nel tipo è sufficiente per caratterizzare un *overloading*. Per aiutare ad analizzare la situazione, si riporta di seguito la definizione di `equals` nella classe derivata `NonLaureato`:

```
public boolean equals(NonLaureato altroNonLaureato) {
    return equals((Studente)altroNonLaureato) &&
        (this.annoDiCorso == altroNonLaureato.annoDiCorso);
}
```

Come si può notare, il frammento di istruzione:

```
(Studente)altroNonLaureato
```

effettua una conversione di tipo. Questo tipo di sintassi è stata presentata nel Capitolo 2 quando si è introdotta la conversione di tipo fra tipi primitivi. In questo caso, si effettua una conversione di tipo per tipi classe. Il risultato consiste nel far sì che `altroNonLaureato` sia considerato di tipo `Studente` e non più di tipo `NonLaureato`. Tale istruzione è legittima poiché `NonLaureato`, essendo una sottoclasse di `Studente`, è anche uno `Studente`. Il Capitolo 11 affronterà più in dettaglio l'argomento.
Perché si effettua la conversione di tipo `Studente` su `altroNonLaureato` nell'invocazione di `equals`? Perché altrimenti Java invocherebbe la definizione di `equals` contenuta nella classe `NonLaureato`. Cioè questo metodo `equals` invocherebbe se stesso. L'esistenza di un metodo che invoca se stesso è perfettamente accettabile quando necessario, ma non è ciò che si vuole accada qui. Effettuando la conversione di tipo `Studente` su `altroNonLaureato`, si induce Java a invocare il metodo `equals` di `Studente`. Si noti che `altroNonLaureato`, essendo un oggetto di tipo `NonLaureato`, ha tutti i comportamenti di un oggetto di tipo `Studente` e così si comporterà correttamente nel metodo `equals` di `Studente`. Si approfondirà questo punto più avanti nel capitolo.

Ereditarietà multipla

Alcuni linguaggi di programmazione, come C++, consentono di derivare una classe da due classi base differenti. Cioè, si può derivare la classe C dalle classi A e B. Questa caratteristica, nota come **ereditarietà multipla**, non è permessa in Java. In Java, una classe può derivare da una sola classe base. Si può, tuttavia, derivare la classe B dalla classe A e poi derivare C dalla classe B; ma questa non è eredità multipla.

10.3.4 Un altro modo per definire il metodo `equals` in `NonLaureato`

Il metodo `equals` della classe `NonLaureato` effettua la conversione di tipo sul parametro `altroNonLaureato` da `NonLaureato` alla sua classe base `Studente` quando lo passa al metodo `equals` di `Studente`. Un altro modo per obbligare Java a invocare la definizione di `equals` presente in `Studente` è usare `super` e un punto, come segue:

```
public boolean equals(NonLaureato altroNonLaureato) {
    return super.equals(altroNonLaureato) &&
        (this.annoDiCorso == altroNonLaureato.annoDiCorso);
}
```

Non si può usare super ripetutamente

Come si è fatto notare in precedenza, dalla definizione di metodo in una classe derivata, utilizzando il prefisso `super` è possibile invocare un metodo della classe base che è stato ridefinito nella classe derivata. Non è però possibile ripetere l'utilizzo di `super` per invocare un metodo di una classe che nella gerarchia di ereditarietà occupa una posizione diversa dalla classe padre. Per esempio, si supponga che la classe `NonLaureato` derivi da `Studente`, la quale a sua volta deriva da `Persona`. Si potrebbe pensare di invocare un metodo della classe `Persona` (che è stato ridefinito sia da `Studente` che da `NonLaureato`) dalla classe `NonLaureato` nel seguente modo:

```
super.super.scriviOutput();     //ILLEGALE!
```

In Java è errato utilizzare più volte in sequenza la parola chiave `super`.

10.3.5 Compatibilità di tipo

Si consideri la classe `NonLaureato` del Listato 10.4, derivata della classe `Studente`. Nel mondo reale, ogni individuo iscritto all'Università non laureato è anche uno studente. Questa relazione è vera anche nell'esempio Java proposto. Ogni oggetto della classe `NonLaureato` è anche un oggetto della classe `Studente`. Così se si ha un metodo che ha un parametro formale di tipo `Studente`, l'argomento in un'invocazione di questo metodo può essere un oggetto di tipo `NonLaureato`. In questo caso, il metodo potrebbe utilizzare solo metodi definiti nella classe `Studente`, ma ogni oggetto della classe `NonLaureato` ha tutti questi metodi.

Per esempio, si supponga che le classi `Studente` e `NonLaureato` siano definite come nei Listati 10.2 e 10.4 e si consideri la seguente definizione di metodo appartenente a un'altra classe:

```
public class UnaClasse {
    public static void confrontaMatricole(Studente s1, Studente s2) {
        if (s1.getMatricola() == s2.getMatricola())
            System.out.println(s1.getNome() +
                               " ha la stessa matricola di " +
                               s2.getNome());
        else
            System.out.println(s1.getNome() +
                               " ha una matricola differente da " +
                               s2.getNome());
    }
    . . .
}
```

Un programma che utilizza `UnaClasse` dovrebbe contenere il seguente codice:

```
Studente oggettoStudente = new Studente("Michela Papalini", 1234);
NonLaureato oggettoNonLaureato =
    new NonLaureato("Maria Aureli", 1234, 1);
UnaClasse.confrontaMatricole(oggettoStudente, oggettoNonLaureato);
```

Se si osserva l'intestazione del metodo `confrontaMatricole`, si vedrà che entrambi i parametri sono di tipo `Studente`. Tuttavia, l'invocazione:

```
UnaClasse.confrontaMatricole(oggettoStudente, oggettoNonLaureato);
```

utilizza un argomento di tipo `Studente` e un altro argomento di tipo `NonLaureato`. Come si può utilizzare un oggetto di tipo `NonLaureato` dove è richiesto un argomento di tipo `Studente`? La risposta è che ogni oggetto di tipo `NonLaureato` è anche di tipo `Studente`. Per complicare la questione, si noti che si possono rovesciare i due argomenti e l'invocazione del metodo sarà ancora valida:

```
UnaClasse.confrontaMatricole(oggettoNonLaureato, oggettoStudente);
```

Si osservi che non viene operata alcuna conversione di tipo. Infatti, un oggetto della classe `NonLaureato` è un oggetto della classe `Studente`, e così esso è di tipo `Studente`.

Un oggetto può comportarsi come se fosse di più tipi in virtù dell'ereditarietà. La classe `NonLaureato` deriva dalla classe `Studente`, che a sua volta è derivata dalla classe `Persona` nel Listato 10.1. Questo significa che ogni oggetto della classe `NonLaureato` è anche un oggetto di tipo `Studente`, ma anche un oggetto di tipo `Persona`. In questo modo, tutto ciò che funziona per gli oggetti della classe `Persona` funziona anche per gli oggetti della classe `NonLaureato`.

Per esempio, si supponga che le classi `Persona` e `NonLaureato` siano definite come nei Listati 10.1 e 10.4 e si consideri il seguente codice che potrebbe trovarsi in un programma:

```
Persona lucaPersona = new Persona("Luca Studente");
System.out.println("Inserire nome:");
Scanner tastiera = new Scanner(System.in);
String nuovoNome = tastiera.nextLine();
NonLaureato unNonLaureato = new NonLaureato(nuovoNome, 222, 3);
if (lucaPersona.haLoStessoNome(unNonLaureato))
   System.out.println("Wow, stessi nomi!");
else
   System.out.println("Nomi differenti");
```

Se si osserva l'intestazione del metodo `haLoStessoNome` nel Listato 10.1, si vedrà che ha un solo parametro, di tipo `Persona`. Tuttavia, l'invocazione nella precedente istruzione `if-else`,

```
lucaPersona.haLoStessoNome(unNonLaureato)
```

è perfettamente valida, anche se l'argomento `unNonLaureato` è un oggetto della classe `NonLaureato`, cioè il suo tipo è `NonLaureato`, e il parametro corrispondente in `haLoStessoNome` è di tipo `Persona`. Ogni oggetto della classe `NonLaureato` è anche un oggetto della classe `Persona`.
Anche la seguente invocazione è valida:

```
unNonLaureato.haLoStessoNome(lucaPersona)
```

Il metodo `haLoStessoNome` appartiene a `Persona`, ma è ereditato dalla classe `NonLaureato`. Così l'oggetto `unNonLaureato` di tipo `NonLaureato` possiede questo metodo. Ogni cosa che funziona per gli oggetti di una classe antenata funziona anche per gli oggetti di ogni classe discendente. In altre parole, un oggetto di una classe discendente può fare le stesse cose di un oggetto di una classe antenata.

Come si è appena visto, se la classe C deriva dalla classe B la quale, a sua volta deriva dalla classe A, allora un oggetto di classe C è di tipo C, ma anche di tipo B e di tipo A. Questo funziona per ogni catena di classi derivate, indipendentemente dalla sua lunghezza.

Poiché un oggetto di una classe derivata ha i tipi di tutte le classi dei suoi antenati in aggiunta al suo tipo, si può assegnare un oggetto di una classe a una variabile di ogni tipo di antenato, ma non il contrario. Per esempio, poiché `Studente` è una classe derivata di `Persona`, e `NonLaureato` è una classe derivata di `Studente`, il seguente codice è valido:

```
Studente s = new Studente();
NonLaureato nl = new NonLaureato();
Persona p1 = s;
Persona p2 = nl;
```

Si può anche non utilizzare le variabili `s` e `nl` e assegnare i nuovi oggetti direttamente alle variabili `p1` e `p2`, come segue:

```
Persona p1 = new Studente();
Persona p2 = new NonLaureato();
```

Al contrario, tutte le seguenti istruzioni sono illecite:

```
Studente s = new Persona();          //ILLEGALE!
NonLaureato nl = new Persona();      //ILLEGALE!
NonLaureato nl2 = new Studente();    //ILLEGALE!
```

E se si definiscono `p` e `s` come segue:

```
Persona p = new Persona();       //valida
Studente s = new Studente();     //valida
```

anche le seguenti istruzioni, che potrebbero sembrare più innocenti, diventano illecite:

```
NonLaureato nl = p;      //ILLEGALE!
NonLaureato nl2 = s;     //ILLEGALE!
```

Tutto ciò ha perfettamente senso. Per esempio, uno `Studente` è una `Persona`, ma una `Persona` non è necessariamente uno `Studente`. Alcuni programmatori trovano la frase "è-un" (dall'inglese *is-a*) utile per decidere quali tipi può avere un oggetto e quali assegnamenti di variabili sono validi. Per esempio, se `Dipendente` è una classe derivata di `Persona`, un `Dipendente` è una `Persona`, così si può assegnare un oggetto `Dipendente` a una variabile di tipo `Persona`. Tuttavia una `Persona` non è necessariamente un `Dipendente`, così non si può assegnare un oggetto creato come `Persona` a una variabile di tipo `Dipendente`.

Il Capitolo 11 tratterà ancora gli assegnamenti in relazione al polimorfismo, un altro elemento chiave della programmazione a oggetti.

Le compatibilità di assegnamento

Un oggetto di una classe derivata ha il tipo della classe derivata, ma può essere assegnato a una variabile il cui tipo è una qualsiasi delle sue classi antenato. Si può assegnare un oggetto di una classe derivata a una variabile di un tipo qualsiasi di antenato, ma non viceversa.

Le Relazioni *is-a* e *has-a*

Come si è già osservato, uno `Studente` è una `Persona`, in quanto la classe `Studente` è una classe derivata dalla classe `Persona`. Questo è un esempio di una relazione *is-a* tra classi.

Un altro tipo di relazione che può sussistere tra classi è noto come relazione *has-a*, già introdotta nel Capitolo 9. Per esempio, se si ha una classe `Data` che memorizza una data, si potrebbe aggiungere una data d'iscrizione alla classe `Studente` aggiungendo una variabile di istanza di tipo `Data` alla classe `Studente`.

In questo caso si dice che uno `Studente` "ha una" (dall'inglese "*has-a*") `Data`. E ancora, se si ha una classe `BraccioMeccanico` e si sta definendo una classe per simulare un robot, si può attribuire alla classe `Robot` una variabile di istanza di tipo `BraccioMeccanico`. In questo caso si dice che un `Robot` "ha un" `BraccioMeccanico`.

Decidere se realizzare una relazione tra classi come una relazione *has-a* o *is-a* a volte può essere difficile. Un elementare, ma efficace, metodo consiste di solito nel seguire semplicemente ciò che sembra più naturale in italiano. Ha più senso dire che "un `Robot` ha un `BraccioMeccanico`" piuttosto che dire "un `Robot` è un `BraccioMeccanico`". Così ha più senso nella programmazione avere un `BraccioMeccanico` come una variabile di istanza della classe `Robot`.

Si incontreranno spesso i termini *is-a* e *has-a* nella letteratura dedicata alle tecniche di programmazione.

Come si è visto nel Capitolo 9, una relazione *has-a* si esplicita in UML con una relazione associativa rappresentata da una linea che lega le due classi in associazione. Si presti attenzione al fatto che la freccia che rappresenta la navigabilità in un'associazione si differenzia stilisticamente da quella che rappresenta una generalizzazione dal fatto che è una freccia aperta.

10.3.6 La classe `Object`

Java ha una classe "Eva", ovvero antenata di ogni classe. In Java, ogni classe deriva dalla classe `Object`. Se una classe `C` ha come classe base la classe `B`, quest'ultima è derivata da `Object`; quindi anche `C` deriva da `Object`. Pertanto, ogni oggetto di ogni classe è di tipo `Object`, così come è del tipo della sua classe e di tutte le sue classi antenato. Anche tutte le classi che si definiscono senza utilizzare l'ereditarietà sono discendenti dalla classe `Object`. Se non si deriva la nuova classe da un'altra classe, Java la deriverà automaticamente da `Object`.

La classe `Object` consente di scrivere metodi Java che hanno parametri di tipo `Object`. In questo modo sarà possibile passare come argomenti oggetti di qualsiasi tipo classe. La classe `Object` ha alcuni metodi che sono ereditati da ogni classe Java. Per esempio, ogni classe eredita dalle classi antenate (se ne è stata fatta una ridefinizione) o direttamente dalla classe `Object` i metodi `equals` e `toString`. Tuttavia, questi due metodi non possono funzionare correttamente per tutte le classi, perché sono troppo generici. Sarà, quindi, necessario ridefinire questi metodi ereditati con nuove definizioni più appropriate.

Scrivere una corretta versione di `equals` è un po' complicato per programmatori alle prime armi. In ogni caso, i metodi `equals` che verranno definiti in questo testo costituiscono una buona base di partenza per acquisire un po' di dimestichezza. Il prossimo paragrafo mostra come si scrive una definizione pienamente completa e corretta di equals.

Il metodo ereditato `toString` non ha argomenti. Si suppone che restituisca tutti i dati di un oggetto impacchettati in una stringa. Tuttavia, la versione ereditata di `toString` è quasi sempre inutile poiché non produrrà un'adeguata rappresentazione della stringa dei dati. È necessario ridefinire `toString` così che produca una stringa appropriata per i dati della classe.

Per esempio, alla classe `Studente` nel Listato 10.2 potrebbe essere aggiunta la seguente definizione di `toString`:

```
public String toString() {
    return "Nome: " + getNome() + "\nMatricola: " + matricola;
}
```

Dopo aver aggiunto questo metodo `toString` alla classe `Studente`, lo si potrebbe utilizzare per mostrare l'output nel seguente modo:

```
Studente luca = new Studente("Luca Studente", 2010);
System.out.println(luca.toString());
```

L'output prodotto sarà:

```
Nome: Luca Studente
Matricola: 2010
```

Definire un nuovo metodo `toString`

Il metodo `toString` di `Object` non mostrerà alcun dato riferito alla propria classe. Di solito si dovrebbe ridefinire `toString` nelle nuove classi che si realizzano.

Un altro metodo ereditato dalla classe `Object` è il metodo `clone`. Questo metodo non ha argomenti e restituisce una copia dell'oggetto chiamante. Un clone è un oggetto che ha dati identici a quelli dell'oggetto che ha invocato il metodo. Come gli altri metodi ereditati dalla classe `Object`, il metodo `clone` deve essere ridefinito perché possa funzionare correttamente nella classe derivata.

10.3.7 Un metodo `equals` migliorato

Come menzionato nel paragrafo precedente, la classe `Object` definisce un metodo `equals` ereditato da ogni classe. Purtroppo, il più delle volte, se si vuole che la propria classe abbia un metodo `equals` che operi in maniera appropriata, occorre ridefinire la definizione di `equals` di `Object`. Nella classe `Studente` fornita nel Listato 10.2 è stato infatti definito un metodo `equals`. Osservando la sua intestazione, però, ci si può rendere conto che in realtà quello che è stato fatto è solo un *overloading*. L'intestazione del metodo `equals` nella definizione della classe `Studente` è infatti:

```
public boolean equals(Studente altroStudente)
```

ma l'intestazione del metodo `equals` nella classe `Object` è:

```
public boolean equals(Object altroOggetto)
```

Questi due metodi `equals` hanno differenti tipi di parametro. In altre parole, la classe `Studente` ha entrambi i metodi. In molte situazioni, ciò non sarà rilevante. Tuttavia, ci sono dei casi in cui è fondamentale aver definito una reale ridefinizione del metodo `equals`.

Si supponga di avere un metodo chiamato `faiQualcosa` che ha un parametro di tipo `Object` chiamato `parametroObject` e un altro parametro `parametroStudente` di tipo `Studente`. Si supponga che il suo corpo contenga l'istruzione `parametroStudente.equals(parametroObject)`. Il metodo è di seguito schematizzato:

```
public void faiQualcosa(Object parametroObject,
                        Studente parametroStudente) {
    ...
    parametroStudente.equals(parametroObject);
    ...
}
```

Se, quando lo si invoca, si passano al metodo due argomenti di tipo `Studente` per i parametri `parametroObject` e `parametroStudente`, Java userà la definizione di `equals` ereditata dalla classe `Object`, non quella che è stata definita nella classe `Studente`. Questo significa che in alcuni casi il metodo `equals` restituirà una risposta sbagliata.

Per risolvere questo problema, è necessario cambiare da `Studente` a `Object` il tipo del parametro del metodo `equals` nella classe `Studente`. Un primo tentativo potrebbe essere il seguente:

```
// Primo tentativo di un metodo equals migliorato
public boolean equals(Object altroOggetto) {
    Studente altroStudente = (Studente)altroOggetto;
    return this.haLoStessoNome(altroStudente) &&
        (this.matricola == altroStudente.matricola);
}
```

Si osservi che è necessaria una conversione di tipo per il parametro `altroOggetto` dal tipo `Object` al tipo `Studente`. Senza la conversione di tipo e la variabile `altroStudente`, si otterrebbe un errore di compilazione dell'istruzione:

```
altroOggetto.matricola
```

perché la classe `Object` non ha una variabile di istanza `matricola`. In realtà, anche l'invocazione `haLoStessoNome` causerebbe un errore.

Questo primo tentativo di miglioramento del metodo `equals` ridefinisce il metodo `equals` della classe `Object` e lavorerà bene in quasi tutti i casi. Tuttavia, ha ancora un difetto: la nuova definizione di `equals` ammette un argomento che può essere un oggetto qualsiasi. Cosa succede se il metodo `equals` viene usato con un argomento che non è uno `Studente`? La risposta è che si otterrà un errore a run-time quando verrà tentata una conversione di tipo a `Studente`.

Si dovrebbe modificare la definizione in modo che accetti un qualsiasi oggetto, ma se l'oggetto non è uno `Studente`, si restituirà semplicemente `false`. Dopo tutto, il metodo appartiene a un oggetto di `Studente` e così se l'argomento non è uno `Studente`,

i due oggetti non possono essere considerati uguali. Ma come si può affermare che il parametro non è di tipo `Studente`? Si può usare l'operatore `instanceof` per verificare se un oggetto è di tipo `Studente`. La sintassi è:

oggetto `instanceof` *nome_della_classe*

Questa espressione restituisce vero se *oggetto* è di tipo *nome_della_classe*; altrimenti restituisce falso. Così la seguente istruzione restituirà vero solo se `altroOggetto` è di tipo `Studente`:

```
altroOggetto instanceof Studente
```

Quindi, il metodo `equals` dovrebbe restituire falso se la precedente espressione booleana è falsa.

Nel Listato 10.5 è mostrata la versione finale del metodo `equals` per la classe `Studente`. Si noti che si è preso in considerazione un ulteriore possibile caso. La costante predefinita `null` può essere assegnata a un parametro di tipo `Object`. La documentazione Java dice che un metodo `equals` dovrebbe restituire falso quando si confronta un oggetto con il valore `null` e questo è quello che è stato fatto.

LISTATO 10.5 Un metodo `equals` migliorato per la classe `Studente`.

```
public boolean equals(Object altroOggetto) {
    boolean uguale = false;
    if ((altroOggetto != null) &&
            (altroOggetto instanceof Studente)) {
        Studente altroStudente = (Studente)altroOggetto;
        uguale = this.haLoStessoNome(altroStudente) &&
            (this.matricola == altroStudente.matricola);
    }
    return uguale;
}
```

10.4 Riepilogo

- Una classe derivata si ottiene da una classe base aggiungendo variabili di istanza e metodi. La classe derivata eredita tutte le variabili di istanza e tutti i metodi pubblici che sono definiti nella classe base.
- Quando si definisce un costruttore per una classe derivata, la definizione dovrebbe per prima cosa invocare un costruttore della classe base, utilizzando `super`. Se non viene fatta un'invocazione esplicita, Java invocherà automaticamente il costruttore di default della classe base.
- All'interno di un costruttore, `this` invoca un costruttore della stessa classe, mentre `super` invoca un costruttore della classe base.

- Si può ridefinire un metodo da una classe base in modo da avere una differente definizione nella classe derivata. Questo è detto *overriding* della definizione del metodo.
- Quando si ridefinisce un metodo, la nuova definizione nella classe derivata deve avere lo stesso nome, gli stessi parametri in termini di tipo, ordine e numero e lo stesso tipo di ritorno del metodo nella classe base.
- Esiste un'eccezione alla regola di *overriding*: se il tipo di ritorno è una classe, è possibile cambiarlo in una qualsiasi delle sue classi derivate.
- Se il metodo nella classe derivata si differenzia dal metodo nella classe base in quanto a elenco di parametri, si parla di *overloading* del metodo e non di *overriding*.
- All'interno della definizione di un metodo di una classe derivata, si può invocare un metodo della classe base ridefinito facendo precedere al nome del metodo `super` e un punto.
- Da una classe derivata non è possibile accedere direttamente per nome alle variabili di istanza private e ai metodi privati di una classe base.
- Un oggetto di una classe derivata ha il tipo della classe derivata e anche il tipo della classe base. Più in generale, una classe derivata ha il tipo di ognuna delle sue classi antenate.
- Si può assegnare un oggetto di una classe derivata a una variabile di qualsiasi tipo antenato, ma non viceversa.
- In Java, ogni classe è una discendente della classe predefinita `Object`. Così qualsiasi oggetto di qualsiasi classe è di tipo `Object`, come pure del tipo della sua classe e di qualsiasi classe antenata.

10.5 Esercizi

1. Si consideri un programma che archivierà gli articoli in una biblioteca scolastica. Si disegni una gerarchia di classi, che include una classe base, per i vari tipi di articoli. Si devono considerare anche gli articoli che non possono essere dati in prestito.
2. Si implementi la classe base per la gerarchia ottenuta dal precedente esercizio.
3. Si crei una classe `BambinoScuola` che è la classe base per i bambini di una scuola. Questa classe dovrebbe avere gli attributi per il nome e l'età del bambino, il nome dell'insegnante del bambino e una stringa di saluto. Questa classe dovrebbe avere appropriati metodi *get* e *set* per ognuno degli attributi.
4. Si derivi una classe `BambinoPrecoce` da `BambinoScuola`, come descritta nel precedente esercizio. La nuova classe dovrebbe ridefinire il metodo *get* per l'età, restituendo l'età attuale aumentata di 2. Essa dovrebbe anche ridefinire il metodo *set* per la stringa di saluto, restituendo la stringa di saluto del bambino concatenata con le parole "Io sono il migliore".
5. Si crei una classe `CalcoloPagamento` che ha un attributo `tariffa` dato in Euro per ora. La classe dovrebbe anche avere un metodo `calcolaPagamento(ore)` che restituisce l'importo da pagare per un dato arco di tempo.

6. Derivare una classe `PagamentoOrdinario` da `CalcoloPagamento`, come descritta nel precedente esercizio. Essa dovrebbe avere un costruttore che ha un parametro per la `tariffa`. Questa classe non ridefinisce alcun metodo. Si derivi poi una classe `PagamentoStraordinario` da `CalcoloPagamento` che ridefinisce il metodo `calcolaPagamento`. Il nuovo metodo dovrebbe restituire l'importo restituito dal metodo nella classe base moltiplicato per 1.5.

10.6 Progetti

1. Si definisca una classe `Dipendente` i cui oggetti rappresentano le schede dei dipendenti di un'azienda. Si derivi questa classe dalla classe `Persona` nel Listato 10.1. Un dipendente eredita il nome dalla classe `Persona`. In aggiunta, un dipendente possiede una retribuzione annuale rappresentata come un valore di tipo `double`, una data di assunzione che fornisce l'anno di assunzione come un valore di tipo `int` e un numero identificativo che è un valore di tipo `String`. Si definiscano gli appropriati costruttori, i metodi *get* e *set* e un metodo `equals`. Si scriva un programma per verificare la definizione della classe.
2. Si definisca una classe `Dottore` i cui oggetti rappresentano le schede dei dottori di una clinica. Si derivi questa classe dalla classe `Persona` fornita nel Listato 10.1. Un dottore ha un nome, definito nella classe `Persona`, una specializzazione descritta tramite una stringa (per esempio Pediatra, Ostetrico, Medico generale e così via) e una parcella per le visite in ufficio (si usi il tipo `double`). Si definiscano gli appropriati costruttori, i metodi *get* e un metodo `equals`. Si scriva un programma di prova per verificare tutti i metodi.
3. Si definiscano due classi, `Paziente` e `Fattura`, i cui oggetti sono schede per una clinica. Si derivi `Paziente` dalla classe `Persona` data nel Listato 10.1. Un `Paziente` ha un nome (definito nelle classe `Persona`) e un numero identificativo (si usi il tipo `String`). Un oggetto `Fattura` conterrà un oggetto `Paziente` e un oggetto `Dottore` (dal Progetto 2). Si definiscano i costruttori appropriati, i metodi *get* e un metodo `equals`. Si scriva inizialmente un programma *driver* per verificare tutti i metodi, si scriva poi un programma di prova che crei almeno due pazienti, almeno due dottori e almeno due schede `Fattura` e che visualizzi il guadagno totale dalle schede `Fattura`.
4. Si crei una classe base `Veicolo` che possiede il nome della casa automobilistica produttrice (di tipo `String`), il numero di cilindri del motore (di tipo `int`) e il proprietario (di tipo `Persona` fornita nel Listato 10.1). Si crei, quindi, una classe chiamata `Camion` che è derivata dalla classe `Veicolo` e possiede delle caratteristiche addizionali: la capacità di carico in tonnellate (di tipo `double` dal momento che può contenere cifre decimali) e la capacità di carico del rimorchio (di tipo `double`). Si dotino le classi di costruttori opportuni, di tutti i metodi *get* e del metodo `equals`. Si scriva un programma *driver* per verificare il funzionamento dei metodi definiti.

5. Si crei una nuova classe `Cane` che sia derivata dalla classe `Animale` fornita nel Listato 9.1 del Capitolo 9. La nuova classe avrà gli attributi aggiuntivi `razza` (tipo `String`) e `comandoDiRichiamo` (tipo `boolean`), il quale sarà vero se l'animale ha il suo comando di richiamo e falso altrimenti. Si dotino le classi di opportuni costruttori e di tutti i metodi *get*. Si scriva un programma *driver* per verificare tutti i metodi, poi si scriva un programma che legga le informazioni per cinque animali di tipo `Cane` e visualizzi il nome e la razza di tutti gli animali che siano oltre i due anni di età e non abbiano assegnati i loro comandi di richiamo.

6. Si definisca una nuova classe `Pagamento` che contenga una variabile di istanza di tipo `double` che memorizza l'importo del pagamento e si definiscano appropriati metodi *get* e *set*. Si crei inoltre un metodo `dettagliPagamento` che visualizza una frase in italiano per descrivere l'importo del pagamento.
   Si definisca poi una classe `PagamentoContanti` che sia derivata da `Pagamento`. Questa classe dovrebbe ridefinire il metodo `dettagliPagamento` per indicare che il pagamento è in contanti. Si includano appropriati costruttori (o un unico costruttore).
   Si definisca una classe `PagamentoCartaDiCredito` derivata da `Pagamento`. Questa classe dovrebbe contenere le variabili di istanza per il nome sulla carta, la data di scadenza e il numero della carta di credito. Si includano appropriati costruttori (o un unico costruttore). Infine, si ridefinisca il metodo `dettagliPagamento` per includere tutte le informazioni della carta di credito oltre all'importo del pagamento.
   Si crei un metodo `main` che crei almeno due oggetti di `PagamentoContanti` e due di `PagamentoCartaDiCredito` con valori differenti e si invochi `dettagliPagamento` per ognuno di essi.

7. Si definisca una classe `Documento` che contenga una variabile di istanza di tipo `String` chiamata `testo` che memorizza qualsiasi contenuto testuale per il documento. Si crei un metodo `toString` che restituisca il valore di `testo` e si includa anche un metodo per impostare questo valore.
   Si definisca poi una classe `Email` che sia derivata da `Documento` e che includa le variabili di istanza per il `mittente`, il `destinatario` e il `titolo` del messaggio. Si implementino metodi *get* e *set* appropriati. Il corpo del messaggio dell'e-mail dovrebbe essere memorizzato nella variabile ereditata `testo`. Si ridefinisca il metodo `toString` per concatenare tutti i campi di testo.
   Analogamente, si definisca una classe `File` che sia derivata da `Documento` e includa una variabile di istanza per il `nomePercorso`. I contenuti testuali del file dovrebbero essere memorizzati nella variabile ereditata `testo`. Si ridefinisca il metodo `toString` che concateni il nome del percorso e il testo.
   Infine, in un programma *driver*, si creino vari oggetti di tipo `Email` e `File`. Si provino gli oggetti passandoli al seguente metodo (incluso nel programma *driver*) che restituisce vero se l'oggetto contiene la parola chiave specificata nel proprio testo.

```
public static boolean contieneParolaChiave(Documento oggettoDoc,
                                           String parolaChiave) {
    if (oggettoDoc.toString().indexOf(parolaChiave, 0) >= 0)
        return true;
    return false;
}
```

8. Il seguente frammento di codice è stato progettato da J. Hacker per un video game. La classe `Alieno` rappresenta un mostro, mentre la classe `GruppoAlieni` rappresenta un gruppo di alieni e quanti danni questi possono infliggere. Le definizioni di tali classi sono fornite di seguito:

```
public class Alieno {
    public static final int ALIENO_SERPENTE = 0;
    public static final int ALIENO_ORCO = 1;
    public static final int ALIENO_UOMO_MARSHMALLOW = 2;
    public int tipo;    // Memorizza uno dei tre tipi sopra indicati
    public int salute;  // 0=morto, 100=forza piena
    public String nome;

    public Alieno(int tipo, int salute, String nome) {
        this.tipo = tipo;
        this.salute = salute;
        this.nome = nome;
    }
}

public class GruppoAlieni {
    private Alieno[] alieno;

    public GruppoAlieni(int alieniNum) {
        alieno = new Alieno[alieniNum];
    }

    public void aggiungiAlieno(Alieno nuovoAlieno, int indice) {
        alieno[indice] = nuovoAlieno;
    }

    public Alieno[] getAlieni() {
        return alieno;
    }

    public int calcolaDanno() {
        int danno = 0;
        for (int i = 0; i < alieno.length; i++) {
            if (alieno[i].tipo == Alieno.ALIENO_SERPENTE) {
                danno +=10;     //Il serpente procura un danno 10
            }
            else if (alieno[i].tipo == Alieno.ALIENO_ORCO) {
                danno +=6;      // L'orco procura un danno 6
            }
            else if (alieno[i].tipo == Alieno.ALIENO_UOMO_MARSHMALLOW) {
                danno +=1;      // L'Uomo Marshmallow procura un danno 1
            }
        }
        return danno;
    }
}
```

Il codice non è molto orientato agli oggetti e non supporta l'information hiding nella classe `Alieno`. *Si riscriva il codice in modo da usare l'ereditarietà per rappresentare i differenti tipi di alieni al posto del parametro "tipo". Si riscriva anche la classe* `Alieno` *per nascondere le variabili di istanza e per creare un metodo* `getDanno` *per ogni classe derivata e che restituisca il danno totale inflitto dall'alieno. Infine, si riscriva il metodo* `calcolaDanno` *e si scriva un metodo* `main` *che provi i codici.*

Capitolo 11

Polimorfismo, classi astratte e interfacce

OBIETTIVI

- Descrivere in generale il polimorfismo.
- Definire opportunamente i metodi per sfruttare appieno il polimorfismo.
- Definire e utilizzare le classi astratte.
- Definire un'interfaccia e utilizzarla per arricchire i comportamenti di una classe.

L'incapsulamento, l'ereditarietà e il polimorfismo costituiscono i tre principali meccanismi della programmazione a oggetti. I primi due sono stati trattati nei capitoli precedenti. Questo capitolo presenta il terzo: il polimorfismo. Con il termine polimorfismo si intende la possibilità di associare più significati a un nome di metodo per mezzo di un meccanismo conosciuto come *binding* dinamico (o *dynamic binding* o *late binding*). Il capitolo illustra anche le classi astratte, particolari classi in cui alcuni metodi possono non essere completamente definiti e servono come classi base su cui costruire classi più specifiche. Sia il polimorfismo sia le classi astratte considerano come esistente un metodo prima ancora che sia definito. Sebbene questo possa sembrare paradossale, tutto ciò funziona. L'ultima parte del capitolo è dedicata alle interfacce. Un'interfaccia specifica un insieme di metodi che ogni classe che implementa quell'interfaccia deve definire. Le interfacce sono tipi e, come tali, godono delle stesse proprietà dei tipi primitivi e dei tipi classe.

Prerequisiti

Occorre aver letto il materiale presentato nei Capitoli fino al 5 e dal Capitolo 8 al 10. Il capitolo non utilizza nessun concetto inerente gli array, trattati nel Capitolo 6.

11.1 Polimorfismo

MyLab

Video 11.1 Polimorfismo

L'ereditarietà permette di definire una classe base il cui codice può essere utilizzato non solo dagli oggetti creati da tale classe, ma anche dagli oggetti creati da ogni classe da essa derivata. Il polimorfismo (*polymorphism*) permette di modificare la definizione dei metodi nella classe derivata e far sì che questi cambiamenti siano effettivi anche per il codice *della classe base*. Tutto ciò avviene in modo automatico in Java, ma è importante capire come funziona. Nel prossimo paragrafo, sarà illustrato il polimorfismo mediante un esempio.

11.1.1 Binding dinamico

Si immagini di dover progettare un insieme di classi che rappresentano diverse tipologie di figure geometriche: rettangoli, cerchi e così via. Ogni figura può essere un oggetto di una classe differente. Per esempio, la classe `Rettangolo` può avere variabili di istanza che rappresentano l'altezza, la larghezza e un punto centrale; la classe `Cerchio` può avere variabili di istanza che rappresentano il centro e il raggio. In un buon progetto software, queste classi dovrebbero tutte derivare da un'unica classe, il cui nome potrebbe essere `Figura`.

Si supponga di voler definire un metodo che permetta di visualizzare la figura geometrica. Per visualizzare un cerchio occorre eseguire operazioni differenti rispetto a quelle necessarie per visualizzare un rettangolo. Di conseguenza, ogni classe ha bisogno di un proprio metodo per visualizzare la figura. Rispettando le buone norme di codifica dei nomi e sapendo che i metodi appartengono a classi diverse, è possibile assegnare ai metodi lo stesso nome: `visualizza`. Se `r` è un oggetto di tipo `Rettangolo` e `c` è un oggetto di tipo `Cerchio`, il comportamento dei metodi `r.visualizza()` e `c.visualizza()` sarà diverso, perché corrisponde all'invocazione di due metodi ben distinti che hanno implementazioni differenti. Quanto descritto corrisponde agli argomenti affrontati nel capitolo precedente; ora si introdurranno i nuovi concetti legati al polimorfismo.

La classe base `Figura` può avere dei metodi utilizzabili da tutte le figure. Per esempio, la classe `Figura` può implementare un metodo `centra` che sposta una figura al centro dello schermo, cancellandola dalla posizione corrente e ridisegnandola al centro. Il metodo `centra` della classe `Figura` può utilizzare il metodo `visualizza` per ridisegnare la figura al centro dello schermo.

Un'implementazione della classe `Figura` potrebbe essere:

```
public class Figura {

    public void centra() {
        //Istruzioni per spostare la figura
        visualizza();
    }

    public void visualizza() {
        // Implementazione vuota: in Figura non si sa come implementarlo
        // poiché dipende dalla specifica figura
    }
```

Mentre quella, per esempio, della classe `Rettangolo` potrebbe essere:

```
public class Rettangolo extends Figura {
    private double altezza;
    private double larghezza;
    private double puntoCentraleX;
    private double puntoCentraleY;

    public void visualizza() {
        //Istruzioni per visualizzare il rettangolo
    }
}
```

Quando si pensa a come usare il metodo `centra`, ereditato dalla classe `Figura`, per le classi `Rettangolo` e `Cerchio` emergono delle complicazioni.

Si consideri, per esempio, la classe `Rettangolo`. `Rettangolo` è una classe derivata da `Figura` e pertanto eredita il suo metodo `centra`. Questa è un'ottima notizia perché non occorre riscrivere il metodo `centra` per la classe `Rettangolo`. Sussiste, però, un problema. Il metodo `centra` usa il metodo `visualizza` che è implementato in modo diverso per ogni tipo di figura. Il metodo `centra` è definito nella classe `Figura` mentre si vorrebbe che la sua esecuzione invocasse il metodo `visualizza` specifico di ogni figura, cioè quello implementato nelle classi `Rettangolo` e `Cerchio`. Se, infatti, `r` è un'istanza della classe `Rettangolo`, si vorrebbe che la seguente istruzione:

```
r.centra();
```

causasse l'invocazione del metodo `visualizza` della classe `Rettangolo` e non quella di `Figura`.

È possibile far sì che accada? La risposta è sì: in Java tutto questo accade addirittura in modo automatico. Quando viene eseguito il metodo `centra` della classe `Rettangolo`, viene invocato il corrispondente metodo `visualizza` definito nella classe `Rettangolo` perché Java utilizza un meccanismo conosciuto come **binding dinamico** (o *dynamic binding* o *late binding*). Ecco il funzionamento del *binding* dinamico per le classi che rappresentano figure.

Il *binding* indica il processo con cui l'invocazione di un metodo viene associata a una definizione specifica del metodo. Si definisce *binding* statico (o *binding* in fase di compilazione o *early binding*) l'associazione tra invocazione e definizione di metodo che viene prodotta al momento della compilazione del codice. Se l'associazione tra invocazione e definizione di metodo viene prodotta quando il metodo è invocato (a run-time) si parla di *binding* dinamico (o *dynamic binding* o *late binding*). Java utilizza il *binding* dinamico per tutti i metodi, fatta eccezione per pochi casi discussi più avanti in questo capitolo. Ora ci si concentrerà sul funzionamento del *binding* dinamico per il metodo `centra`.

Si ricorda che il metodo `centra` è definito nella classe `Figura` e che la definizione del metodo `centra` include l'invocazione del metodo `visualizza`. Se, contrariamente a quanto accade, Java utilizzasse il *binding* statico, l'invocazione del metodo `visualizza` presente nel metodo `centra` sarebbe associata alla definizione di `visualizza` presente nella classe `Figura` e non alla definizione di `visualizza` presente nelle classi derivate da `Figura`. Quindi, se fosse utilizzato il *binding* statico, il metodo `centra` si comporterebbe esattamente allo stesso modo per tutte le figure, come specificato dalla definizione di `visualizza` fornita nella classe `Figura`. Fortunatamente Java utilizza il *binding*

dinamico; quindi, a fronte di un'istruzione di invocazione del metodo `centra` da parte di un oggetto della classe `Rettangolo`, l'invocazione di `visualizza` (che si trova nel metodo `centra`) non viene associata a una specifica definizione di `visualizza` finché l'invocazione non viene effettivamente eseguita. Nel momento in cui `visualizza` viene invocato da `centra`, l'ambiente d'esecuzione conosce che il chiamante è un'istanza di tipo `Rettangolo` e associa la chiamata alla definizione di `visualizza` fornita nella classe `Rettangolo` (anche se la chiamata di `visualizza` si trova nel metodo `centra` che è definito nella classe `Figura`). Quindi, il metodo `centra` si comporta in modo diverso per un oggetto di tipo `Rettangolo` piuttosto che per un oggetto di tipo `Cerchio` o `Figura`. Grazie al *binding* dinamico l'esecuzione procede nel modo ideale in maniera del tutto automatica. Il termine polimorfismo e *binding* dinamico identificano due concetti fortemente simili. Il termine *polimorfismo* indica la possibilità di assegnare più significati a uno stesso metodo sfruttando il *binding* dinamico.

Binding dinamico e polimorfismo

Il *binding* dinamico fa sì che l'invocazione di un metodo venga associata alla sua definizione solamente a run-time, quindi nel momento in cui l'invocazione viene effettivamente eseguita. Java utilizza il *binding* dinamico per tutti i metodi, tranne per quelli discussi nel Paragrafo 11.1.4.

Il termine polimorfismo fa riferimento alla capacità di assegnare più significati a uno stesso nome di metodo, sfruttando il meccanismo di *binding* dinamico. Quindi, polimorfismo e *binding* dinamico fanno riferimento al medesimo concetto.

ESEMPIO DI PROGRAMMAZIONE
ARCHIVIO VENDITE

Si supponga di voler progettare un programma per la gestione di un archivio delle vendite di componenti per auto. L'obiettivo è quello di realizzare un programma versatile, proprio perché è difficile considerare fin dall'inizio tutte le possibili tipologie di vendita. In principio il sistema supporterà solamente le vendite "normali" che avvengono direttamente nel negozio; in un secondo momento, si potrebbe voler aggiungere il supporto per le vendite di componenti in saldo oppure prevedere la vendita per corrispondenza e quindi includere nel prezzo anche il costo di spedizione. Per tutte queste tipologie di vendita, il componente venduto avrà un costo base che contribuirà alla determinazione del costo finale. Per le vendite che avvengono in negozio, il costo base del componente sarà anche il costo finale, ma se si aggiunge il supporto per la vendita di componenti in saldo, il costo finale dipenderà anche dalla percentuale di sconto. Si suppone che il programma debba anche calcolare il fatturato lordo giornaliero che, intuitivamente, è dato dalla somma di tutte le vendite effettuate nell'arco della giornata. Si potrebbe anche voler determinare la vendita di importo più alto o più basso del giorno, piuttosto che il ricavo medio delle vendite del giorno. Tutti questi dati possono essere derivati dal costo di ciascuna vendita, ma molti dei metodi di calcolo dei costi saranno aggiunti solamente quando saranno definiti tutti i tipi di vendita da supportare. Poiché Java utilizza il *binding* dinamico è possibile scrivere un programma che calcola il totale delle vendite,

anche se la modalità di calcolo del costo di ciascuna vendita sarà aggiunto in un secondo momento. Per semplicità, nell'esempio seguente si supporrà che ciascuna vendita riguardi un solo componente, anche se nel caso generale una vendita può includere più componenti anche diversi fra loro.

Il Listato 11.1 mostra la definizione della classe `Vendita`. Tutti i tipi di vendita saranno classi derivate dalla classe `Vendita`. La classe `Vendita` corrisponde a semplici vendite di singoli elementi prive di sconti e tariffe aggiuntive. Si noti che i metodi `minoreDi` e `uguaglianzaVendite` includono entrambi l'invocazione del metodo `totale`. È possibile definire in un secondo momento delle classi derivate da `Vendita` che includono la propria definizione del metodo `totale`. Le definizioni dei metodi `minoreDi` e `uguaglianzaVendite` date nella classe `Vendita` utilizzeranno la versione del metodo `totale` che corrisponde al particolare oggetto della classe derivata.

Per esempio, il Listato 11.2 mostra la classe derivata `VenditaScontata`. Si noti che la classe `VenditaScontata` necessita di una nuova definizione per il metodo `totale`. I metodi `minoreDi` e `uguaglianzaVendite`, che usano il metodo `totale`, sono ereditati dalla classe base `Vendita`. Quando i metodi `minoreDi` e `uguaglianzaVendite` saranno utilizzati con un oggetto di tipo `VenditaScontata` essi utilizzeranno la versione del metodo `totale` data nella classe `VenditaScontata`.

Si consideri che `d1` e `d2` siano oggetti di tipo `VenditaScontata` e che venga effettuata la seguente invocazione di metodo:

```
d1.minoreDi(d2)
```

Il metodo `minoreDi` è definito nella classe `Vendita` come di seguito:

```
public boolean minoreDi(Vendita altraVendita) {
    if (altraVendita == null) {
        System.out.println("Errore: oggetto Vendita è null.");
        System.exit(0);
    }
    //else
    return (totale() < altraVendita.totale());
}
```

Chiaramente l'invocazione è lecita da parte di `d1`, poiché la classe `VenditaScontata` eredita il metodo dalla sua classe base `Vendita`. Anche l'argomento passato `d2` è lecito, poiché `d2`, essendo di tipo `VenditaScontata` è anche di tipo `Vendita` grazie all'ereditarietà (una relazione *is-a* vista nel Capitolo 10).

Nel frammento di istruzione:

```
totale() < altraVendita.totale()
```

l'invocazione al metodo `totale`, non essendo specificato l'oggetto ricevente, è effettuata da `this` (come visto nel Capitolo 8). A seguito dell'esecuzione dell'istruzione `d1.minoreDi(d2)`, `this` è un sinonimo di `d1` all'interno del metodo `minoreDi`. Grazie al *binding* dinamico, il metodo `totale` invocato da `this` è il metodo definito nella classe `VenditaScontata` e non quello definito nella classe `Vendita`, poiché `this` è un oggetto di tipo `VenditaScontata`. Lo stesso discorso si applica per `altraVendita` che è un sinonimo di `d2`.

Il Listato 11.3 riporta un esempio che illustra il funzionamento del *binding* dinamico del metodo `totale` in un programma completo.

MyLab

LISTATO 11.1 La classe base Vendita.

```
/**
La classe rappresenta la vendita di un singolo elemento.
La classe ignora tasse, sconti e qualsiasi altro aggiustamento del prezzo.
Il prezzo assume valori non negativi; il nome è una stringa non vuota
*/
public class Vendita {
    private String nome;    //Una stringa non vuota
    private double prezzo;  //non negativo

    public Vendita() {
        nome = "Nessun nome";
        prezzo = 0;
    }

    /**
    Precondizione: ilNome è una stringa non vuota; ilPrezzo è non negativo.
    */
    public Vendita(String ilNome, double ilPrezzo) {
        setNome(ilNome);
        setPrezzo(ilPrezzo);
    }

    public Vendita(Vendita oggettoOriginale) {
        if (oggettoOriginale == null) {
            System.out.println("Errore: oggetto Vendita null.");
            System.exit(0);
        } //else
        nome = oggettoOriginale.nome;
        prezzo = oggettoOriginale.prezzo;
    }

    public static void annuncio() {
        System.out.println("Questa e' la classe Vendita.");
    }

    public double getPrezzo() {
        return prezzo;
    }

    /**
    Precondizione: nuovoPrezzo è non negativo.
    */
```

```
/**
Precondizione: nuovoNome è una stringa non vuota.
*/
public void setNome(String nuovoNome) {
    if (nuovoNome != null && nuovoNome != "")
        nome = nuovoNome;
    else {
        System.out.println("Errore: nome errato.");
        System.exit(0);
    }
}

public String toString() {
    return ("Componente = " + nome +
            ", Prezzo e costo totale = E" + prezzo);
}

public double totale() {
    return prezzo;
}

/**
Restituisce true se i nomi e i totali delle vendite sono gli stessi;
altrimenti restituisce false.
Il metodo restituisce false anche se altraVendita è null.
*/
public boolean uguaglianzaVendite(Vendita altraVendita) {
    if (altraVendita == null)
        return false;
    else
        return (nome.equals(altraVendita.nome) &&
                totale() == altraVendita.totale());
}

/**
Restituisce true se il totale dell'oggetto è minore del totale
dell'oggetto altraVendita; altrimenti restituisce false.
*/
public boolean minoreDi(Vendita altraVendita) {
    if (altraVendita == null) {
        System.out.println("Errore: oggetto Vendita è null.");
        System.exit(0);
    }
    //else
    return (totale() < altraVendita.totale());
```

Quando invocati, questi metodi utilizzeranno la definizione del metodo `totale()` più appropriata per ciascun oggetto.

```
    }

    public boolean equals(Object altroOggetto) {
        if (altroOggetto == null)
            return false;
        else if(!(altroOggetto instanceof Vendita))
            return false;
        else {
            Vendita altraVendita = (Vendita)altroOggetto;
            return (nome.equals(altraVendita.nome) &&
                (prezzo == altraVendita.prezzo));
        }
    }
}
```

MyLab

LISTATO 11.2 La classe derivata `VenditaScontata`.

```
/**
Classe per la vendita di un componente con lo sconto
espresso come una percentuale del prezzo, ma senza altri aggiustamenti.
Il prezzo e lo sconto assumono valori non negativi;
il nome è una stringa non vuota.
*/
public class VenditaScontata extends Vendita {

    private double sconto;     //Una percentuale del prezzo.
                               //Non può essere negativa.

    public VenditaScontata() {
        super();
        sconto = 0;
    }

    /**
    Precondizione: ilNome è una stringa non vuota; ilPrezzo è non negativo;
    loSconto è espresso come una percentuale del prezzo ed è non negativo.
    */
    public VenditaScontata(String ilNome, double ilPrezzo,
                           double loSconto) {
        super(ilNome, ilPrezzo);
        setSconto(loSconto);
    }

    public VenditaScontata(VenditaScontata oggettoOriginale) {
        super(oggettoOriginale);
        sconto = oggettoOriginale.sconto;
    }
```

Il risultato non cambia se questa riga fosse stata omessa.

```
    public static void annuncio() {
        System.out.println("Questa e' la classe VenditaScontata.");
    }

    public double totale() {
        double frazione = sconto / 100;
        return (1 - frazione) * getPrezzo();
    }

    public double getSconto() {
        return sconto;
    }

    /**
    Precondizione: nuovoSconto è non negativo.
    */
    public void setSconto(double nuovoSconto) {
        if (nuovoSconto >= 0)
            sconto = nuovoSconto;
        else {
            System.out.println("Errore: sconto negativo.");
            System.exit(0);
        }
    }

    public String toString() {
        return ("Componente = " + getNome() +
                ", Prezzo = E" + getPrezzo() +
                " Sconto = " + sconto + "%\n" +
                " Costo totale = E" + totale());
    }

    public boolean equals(Object altroOggetto) {
        if (altroOggetto == null)
            return false;
        else if (!(altroOggetto instanceof VenditaScontata))
            return false;
        else {
            VenditaScontata altraVenditaScontata =
                (VenditaScontata)altroOggetto;
            return (super.equals(altraVenditaScontata) &&
                sconto == altraVenditaScontata.sconto);
        }
    }
}
```

MyLab

LISTATO 11.3 Dimostrazione del *binding* dinamico.

```
/**
Esempio di binding dinamico.
*/
public class BindingDinamicoDemo {
    public static void main(String[] args) {

        Vendita semplice =
            new Vendita("tappetino auto", 10.00);     //un prodotto a E10.00.
        VenditaScontata scontato =
            new VenditaScontata("tappetino auto", 11.00, 10);
            //un prodotto a E11.00 con il 10% di sconto.

        System.out.println(semplice.toString());
        System.out.println(scontato.toString());

        if (scontato.minoreDi(semplice))
            System.out.println("Il prodotto scontato costa meno.");
        else
            System.out.println("Il prodotto scontato non costa meno.");

        Vendita prezzoNormale =
            new Vendita("porta bicchiere", 9.90); //un prodotto a E9.90.
        VenditaScontata prezzoSpeciale =
            new VenditaScontata("porta bicchiere", 11.00, 10);
            //un prodotto a E11.00 con il 10% di sconto.

        System.out.println(prezzoNormale.toString());
        System.out.println(prezzoSpeciale.toString());

        if (prezzoSpeciale.uguaglianzaVendite(prezzoNormale))
            System.out.println("Il costo totale e' lo stesso.");
        else
            System.out.println("Il costo totale e' diverso.");
    }
}
```

Il metodo `minoreDi` usa definizioni diverse per `semplice.totale()` e `scontato.totale()`.

Il metodo `uguaglianzaVendite` usa definizioni diverse per `prezzoSpeciale.totale()` e `prezzoNormale.totale()`.

Il metodo `uguaglianzaVendite` classifica le due vendite come uguali se hanno lo stesso nome e lo stesso totale. Non importa come viene calcolato il totale.

Esempio di output

```
Componente = tappetino auto, Prezzo e costo totale = E10.0
Componente = tappetino auto, Prezzo = E11.0 Sconto = 10.0%
Costo totale = E9.9
Il prodotto scontato costa meno.
Componente = porta bicchiere, Prezzo e costo totale = E9.9
Componente = porta bicchiere, Prezzo = E11.0 Sconto = 10.0%
Costo totale = E9.9
Il costo totale e' lo stesso.
```

La Figura 11.1 illustra il diagramma delle classi per l'esempio proposto. Si noti che il metodo `annuncia`, dal momento che è statico, è sottolineato.

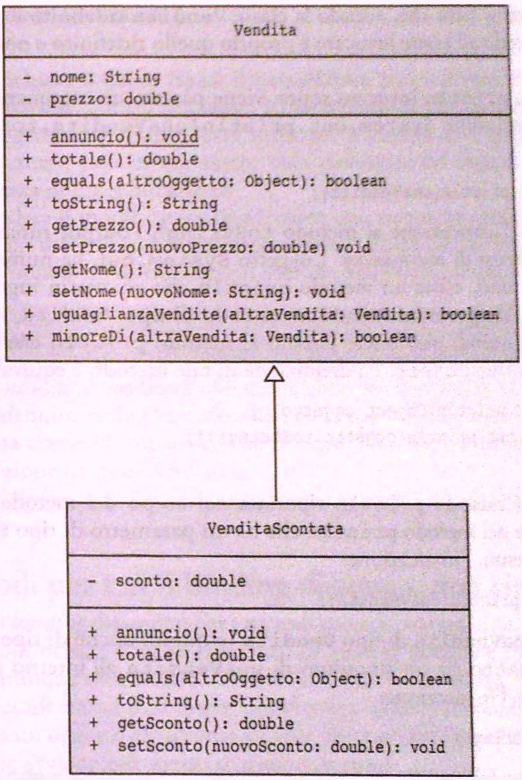


Figura 11.1 Diagramma delle classi di `Vendita` e `VenditaScontata`.

11.1.2 Binding dinamico con `toString`

Nel Capitolo 10 si è mostrato che la classe `Object` definisce, tra gli altri, anche il metodo `toString`. Tale metodo restituisce tutti i dati di un oggetto, impacchettati in una stringa. Tuttavia, la versione ereditata di `toString` è quasi sempre inutile, poiché non produce un'adeguata rappresentazione della stringa di dati. È necessario ridefinire `toString` in modo che produca una stringa appropriata per i dati degli oggetti della classe.

Le classi `Vendita` e `VenditaScontata`, quindi, ridefiniscono opportunamente il metodo `toString`. A seguito delle seguenti istruzioni:

```
Vendita unaVendita = new Vendita("pneumatico", 9.95);
System.out.println(unaVendita.toString());
```

si ottiene un output come quello che segue:

```
Componente = pneumatico, Prezzo e costo totale = E9.95
```

Questo è dovuto al fatto che, avendo la classe `Vendita` ridefinito il metodo `toString` ereditato, il metodo ad essere invocato è proprio quello ridefinito e non quello della classe `Object`.

Al metodo `println` invocato sopra, viene passato un argomento di tipo `String`. In realtà, se l'istruzione `System.out.println(unaVendita.toString())` venisse sostituita da:

```
System.out.println(unaVendita);
```

(si omette, cioè, l'invocazione al metodo `toString`), l'output rimane invariato. Infatti, `println` è frutto di *overloading*. L'oggetto `System.out` ha numerose definizioni di `println`. Fra i tanti, esiste un metodo `println` che accetta in ingresso un parametro di tipo `Object`. L'oggetto `unaVendita`, essendo di tipo `Vendita`, è anche un oggetto di tipo `Object`; quindi può essere passato al metodo `println` che accetta in ingresso un argomento di tipo `Object`. La definizione di tale metodo è equivalente alla seguente:

```
public void println(Object oggetto) {
    System.out.println(oggetto.toString());
}
```

L'invocazione del metodo `println` riportata nel corpo del metodo sopra specificato, esegue la versione del metodo `println` che ha un parametro di tipo `String`.

Di conseguenza, l'invocazione:

```
System.out.println(unaVendita);
```

con parametro `unaVendita` di tipo `Vendita` (e quindi anche di tipo `Object`) fa sì che il parametro `oggetto` sia un sinonimo di `unaVendita` all'interno del metodo. Per il *binding* dinamico, l'invocazione:

```
oggetto.toString()
```

causa l'invocazione del metodo `toString` definito in `Vendita` e non quello di `Object`, poiché è stato ridefinito.

Definire sempre `toString` nelle proprie classi

È sempre buona norma ridefinire il metodo `toString` ereditato dalla classe `Object` in modo che, se occorre visualizzare informazioni utili sullo stato di un oggetto, basta invocare il metodo `println` passando l'oggetto come argomento.

11.1.3 Il modificatore `final`

È possibile specificare che un metodo non può essere ridefinito (*overriding*) nelle classi derivate. Per fare questo è sufficiente aggiungere il modificatore `final` alla dichiarazione del metodo, come mostrato di seguito:

```
public final void unMetodo() {
    ...
```

Un'intera classe può essere dichiarata `final`. Le classi `final` non possono essere utilizzate come classi base di classi derivate. Ecco la sintassi per dichiarare una classe come `final`:

```
public final class unaClasse {
  ...
```

Se un metodo è dichiarato come `final` il compilatore può utilizzare il *binding* statico per migliorare l'efficienza dell'applicazione. L'incremento di efficienza non è comunque significativo. Si suggerisce, quindi, di non utilizzare il modificatore `final` solamente per migliorare le prestazioni; può invece essere utile dichiarare dei metodi come `final` per migliorare la sicurezza di un'applicazione.

È possibile vedere il modificatore `final` come uno strumento per disabilitare il *binding* dinamico per un metodo (o per un'intera classe). Chiaramente, il modificatore `final` fa qualcosa di più: impedisce di ridefinire il metodo nelle classi derivate.

Il modificatore `final`

Se si aggiunge il modificatore `final` alla definizione di un metodo, si impedisce che il metodo venga ridefinito nelle classi derivate. Se si aggiunge il modificatore `final` alla definizione di una classe, si impedisce che la classe possa essere utilizzata come classe base per la definizione di classi derivate.

11.1.4 Metodi per cui il binding dinamico non viene applicato

Java non utilizza il *binding* dinamico per i metodi privati, i metodi `final` e i metodi statici. Nel caso dei metodi privati e `final`, l'assenza di *binding* dinamico non rappresenta un limite, perché comunque non sarebbe di alcuna utilità. Al contrario, l'assenza di *binding* dinamico per i metodi statici può essere significativa quando il metodo statico viene invocato utilizzando un oggetto chiamante (invece che mediante il nome della classe che lo definisce), cosa che avviene più spesso di quanto si pensi.

Quando Java, o un qualsiasi altro linguaggio, non utilizza il *binding* dinamico, utilizza il *binding* statico. Nel caso del *binding* statico, la decisione su quale definizione di metodo debba essere eseguita viene presa durante la compilazione sulla base del tipo dell'oggetto chiamante.

Il Listato 11.4 mostra l'effetto del *binding* statico nel caso in cui venga invocato un metodo statico con un oggetto chiamante. Si noti che il metodo statico `annuncio()`, definito nella classe `Vendita`, è stato ridefinito nella classe `VenditaScontata`. Nonostante questo, quando si ha un riferimento a un oggetto di tipo `VenditaScontata` da una variabile di tipo `Vendita`, il metodo `annuncio()` che viene eseguito è quello definito nella classe `Vendita` e non quello definito nella classe `VenditaScontata`.

Per quale motivo Java si comporta in questo modo? Un metodo statico viene normalmente invocato utilizzando un nome di classe e non un oggetto chiamante. Purtroppo non è sempre così e alcune volte un metodo statico può avere un oggetto chiamante nascosto. Se si invoca un metodo statico dalla definizione di un metodo non statico senza utilizzare né un nome di classe, né un oggetto chiamante, l'oggetto chiamante implicitamente utilizzato è `this`.

Per esempio, si supponga di aggiungere il seguente metodo alla classe `Vendita`:

```
public void mostraAnnuncio() {
    annuncio();
```

```
        System.out.println(toString());
    }
```

Si supponga, inoltre, che il metodo `mostraAnnuncio()` non sia ridefinito nella classe `VenditaScontata` e che quindi `VenditaScontata` erediti direttamente questo metodo dalla classe `Vendita`.

Si consideri il seguente frammento di codice:

```
Vendita vendita = new Vendita("tappetino auto", 10.00);
VenditaScontata scontata = new VenditaScontata("tappetino auto", 11.00, 10);
vendita.mostraAnnuncio();
scontata.mostraAnnuncio();
```

Ci si potrebbe aspettare il seguente output:

```
Questa e' la classe Vendita.
Componente = tappetino auto, Prezzo e costo totale = E10.0
Questa e' la classe VenditaScontata.
Componente = tappetino auto, Prezzo = E11.0 Sconto = 10.0%
Costo totale = E9.9
```

Siccome la definizione associata all'invocazione del metodo statico `annuncio` che si trova nel metodo `mostraAnnuncio` è determinata al momento della compilazione (sulla base del tipo dell'oggetto chiamante), l'output realmente prodotto è il seguente (le differenze sono mostrate in blu):

```
Questa e' la classe Vendita.
Componente = tappetino auto, Prezzo e costo totale = E10.0
Questa e' la classe Vendita.
Componente = tappetino auto, Prezzo = E11.0 Sconto = 10.0%
Costo totale = E9.9
```

Java utilizza il *binding* dinamico per il metodo non statico `toString`, ma utilizza il *binding* statico per il metodo statico `annuncio`.

MyLab

LISTATO 11.4 Dimostrazione del binding statico.

```
/**
Esempio di binding statico dei metodi statici.
*/
public class BindingStaticoDemo {
    public static void main(String[] args) {

        Vendita.annuncio();
        VenditaScontata.annuncio();
        System.out.println("Questo esempio mostra che e' possibile " +
                           "fare l'overriding di un metodo statico.");

        Vendita vendita = new Vendita();
        VenditaScontata scontata = new VenditaScontata();
        vendita.annuncio();
        scontata.annuncio();
        System.out.println("Nessuna sorpresa finora, ma...");
```

Java utilizza il binding statico con i metodi statici, quindi la scelta del metodo da eseguire è determinata dal tipo dell'oggetto chiamante e non dall'oggetto.

```
        Vendita scontata2 = scontata;
        System.out.println("scontata2 e' di tipo VenditaScontata ma e'" +
                           "\nreferenziata da una variabile di tipo Vendita.");
        System.out.println("Quale definizione di annuncio() sara' usata?");
        scontata2.annuncio();
        System.out.println("Usa la versione di annuncio() definita " +
                           "in Vendita!");
    }
}
```

scontata e scontata2 fanno riferimento allo stesso oggetto, ma una variabile è di tipo Vendita e l'altra è di tipo VenditaScontata.

Esempio di output

```
Questa e' la classe Vendita.
Questa e' la classe VenditaScontata.
Questo esempio mostra che e' possibile fare l'overriding di un metodo statico.
Questa e' la classe Vendita.
Questa e' la classe VenditaScontata.
Nessuna sorpresa finora, ma...
scontata2 e' di tipo VenditaScontata ma e'
referenziata da una variabile di tipo Vendita.
Quale definizione di annuncio() sara' usata?
Questa e' la classe Vendita.
Usa la versione di annuncio() definita in Vendita!
```

Se Java avesse usato il binding dinamico con i metodi statici, questa riga sarebbe stata prodotta dal metodo annuncio della classe `VenditaScontata` anziché dal metodo annuncio della classe `Vendita`.

Metodi per cui il *binding* dinamico non viene applicato

Java non utilizza il *binding* dinamico per i metodi privati, i metodi `final` e i metodi statici. Nel caso dei metodi privati e `final`, l'assenza di *binding* dinamico non rappresenta un limite, perché comunque esso non sarebbe di alcuna utilità. Al contrario, l'assenza del *binding* dinamico per i metodi statici può essere significativa quando il metodo statico viene invocato utilizzando un oggetto chiamante (invece che mediante il nome della classe che lo definisce), cosa che avviene più spesso di quanto si pensi.

11.1.5 Downcast e upcast

Come anticipato nel Capitolo 10, il seguente frammento di codice è perfettamente legale (date le definizioni di classi nei Listati 11.1 e 11.2):

```
Vendita variabileVendita;
VenditaScontata variabileScontata =
    new VenditaScontata("verniciatura", 15, 10);
variabileVendita = variabileScontata;
System.out.println(variabileVendita.toString());
```

Un oggetto di una classe derivata, in questo caso `VenditaScontata`, è anche del tipo della classe base, in questo caso `Vendita` e quindi può essere assegnato a una variabile del

tipo della classe base. Per questo motivo, è possibile assegnare a `variabileVendita` (di tipo `Vendita`) l'oggetto `variabileScontata` di tipo `VenditaScontata`.

Si consideri ora l'invocazione del metodo `toString()` nell'ultima riga del frammento di codice riportato.

Dal momento che Java utilizza il *binding* dinamico, l'invocazione:

```
variabileVendita.toString()
```

esegue il metodo `toString` definito nella classe `VenditaScontata`. Quindi l'output prodotto dal frammento di codice è:

```
Componente = verniciatura, Prezzo = E15.0 Sconto = 10.0%
Costo totale = E13.5
```

A causa del *binding* dinamico il significato del metodo `toString` è determinato dall'oggetto e non dal tipo della variabile `variabileVendita`.

Si potrebbe erroneamente credere che assegnare un oggetto di tipo `VenditaScontata` a una variabile di tipo `Vendita` sia inutile[1]. Questi assegnamenti vengono fatti molto più di frequente di quanto si possa pensare, ma si tende a non notarli, perché spesso avvengono in modo trasparente "dietro le quinte". Si ricordi che un parametro è, a tutti gli effetti, una variabile locale. Quindi, ogni volta che un oggetto di tipo `VenditaScontata` viene utilizzato come argomento per un parametro di tipo `Vendita`, si sta assegnando un oggetto di tipo `VenditaScontata` a una variabile di tipo `Vendita`. Si consideri, per esempio, la seguente invocazione estratta dalla definizione dell'ultimo costruttore della classe `VenditaScontata` (chiamato anche costruttore per copia, dal momento che istanzia un nuovo oggetto "copiando" i valori delle variabili di istanza dell'oggetto passato come argomento) mostrata nel Listato 11.2:

```
super(oggettoOriginale);
```

In questa invocazione, `oggettoOriginale` è di tipo `VenditaScontata`, mentre `super` è il costruttore per copia della classe base `Vendita`. Il costruttore per copia accetta un parametro di tipo `Vendita` e, quindi, implica l'esistenza di una varabile locale dello stesso tipo. L'invocazione sopra riportata usa l'argomento `oggettoOriginale` di tipo `VenditaScontata` in corrispondenza di un parametro di tipo `Vendita`, assegnando quindi un oggetto di tipo `VenditaScontata` a una variabile di tipo `Vendita`.

Si noti che il tipo di una variabile che fa riferimento a un oggetto determina quali metodi possono essere invocati utilizzando l'oggetto come oggetto chiamante. Invece, l'oggetto chiamante è sempre fondamentale per determinare la definizione di metodo che deve essere associata all'invocazione, per effetto del *binding* dinamico.

Un oggetto conosce le definizioni dei propri metodi

Il tipo di una variabile determina quali metodi possono essere invocati usando la variabile, ma l'oggetto referenziato dalla variabile determina l'esatta definizione di metodo da eseguire. Il tipo di un parametro determina quali metodi possono essere invocati utilizzando il parametro, ma l'argomento determina l'esatta definizione del metodo da eseguire.

[1] In realtà sono i riferimenti agli oggetti che vengono assegnati e non gli oggetti stessi, ma queste sottili differenze non sono importanti nel contesto di questa discussione.

Assegnare un oggetto di una classe derivata a una variabile del tipo della classe base (o una qualsiasi superclasse) è spesso chiamato **upcast** (letteralmente "conversione verso l'alto") perché è come fare la conversione dal tipo derivato al tipo della classe base e, normalmente, i diagrammi che mostrano la relazioni di ereditarietà riportano le classi base sopra le classi derivate[2].

La conversione da una classe base a una classe derivata (o da una superclasse a una sottoclasse) viene chiamata **downcast** (letteralmente "conversione verso il basso").

L'*upcast* è semplice e non ci sono casi insidiosi di cui preoccuparsi. Java si comporta sempre come ci si aspetta. Il *downcast* è più complesso. Innanzitutto non ha sempre senso. Per esempio, il seguente *downcast*:

```
Vendita variabileVendita = new Vendita("verniciatura",15);
VenditaScontata variabileScontata;
variabileScontata = (VenditaScontata)variabileVendita;    //errore
```

non ha senso perché l'oggetto `variabileVendita` non ha nessuna variabile di istanza che si chiami `sconto` e quindi non può essere un oggetto di tipo `VenditaScontata`. Ogni `VenditaScontata` è una `Vendita`, ma non ogni `Vendita` è una `VenditaScontata`, come mostrato da questo esempio. È una responsabilità del programmatore utilizzare il *downcast* solamente nei casi in cui ha senso farlo.

È istruttivo far notare che:

```
variabileScontata = (VenditaScontata)variabileVendita;
```

produce un errore a run-time, ma sarà compilato senza problemi. Invece la seguente istruzione, sempre errata, produrrà un errore di compilazione:

```
variabileScontata = variabileVendita;
```

Java intercetta gli errori di *downcast* appena possibile. Alcuni errori di *downcast* possono essere rilevati durante la compilazione, mentre altri possono essere identificati solamente a run-time.

Benché il *downcast* possa essere pericoloso, talvolta è necessario. Per esempio, inevitabilmente si utilizza il *downcast* quando viene definito il metodo `equals` di una classe. Si osservi la seguente riga di codice estratta dalla definizione di `equals` della classe `Vendita` (si veda il Listato 11.1):

```
Vendita altraVendita = (Vendita)altroOggetto;
```

Si tratta di un *downcast* dal tipo `Object` al tipo `Vendita`. Senza questo *downcast* le variabili di istanza `nome` e `prezzo` sarebbero utilizzate illegalmente nell'istruzione di `return` riportata di seguito, perché la classe `Object` non ha tali variabili di istanza:

```
return (nome.equals(altraVendita.nome) &&
       (prezzo == altraVendita.prezzo));
```

[2] È preferibile pensare a un oggetto del tipo della classe derivata come se fosse anche del tipo della classe base. Quindi l'*upcast* non è letteralmente una conversione, ma non è errato pensare a questa operazione come a una conversione di tipo.

Downcast

È una responsabilità del programmatore quella di utilizzare il *downcast* solamente nei casi in cui ha senso farlo. Il compilatore non fa alcun controllo per verificare che il *downcast* sia sensato. In ogni caso, se si usa il *downcast* in situazioni dove non ha senso, solitamente sarà prodotto un messaggio d'errore a run-time.

Verificare se il *downcast* è legale

È già stato introdotto alla fine del Capitolo 10, ma si riprende qui l'operatore `instanceof` presentandolo nella sua completezza. Come si è detto, è possibile utilizzare l'operatore `instanceof` per verificare se un *downcast* è possibile. Il *downcast* verso un tipo specifico funziona se l'oggetto che deve essere convertito è di quel tipo. Questo tipo di controllo può essere realizzato con l'operatore `instanceof`.

L'operatore `instanceof` controlla se un oggetto è del tipo specificato come secondo argomento dell'operatore. La sintassi è:

oggetto `instanceof` *nome_di_una_classe*

L'espressione restituisce `true` se *oggetto* è di tipo *nome_di_una_classe*; altrimenti restituisce `false`. Quindi, la seguente istruzione restituirà `true` se `unOggetto` è di tipo `VenditaScontata`:

```
unOggetto instanceof VenditaScontata
```

Si noti che qualsiasi oggetto di un tipo derivato da `VenditaScontata` è anche di tipo `VenditaScontata`, quindi questa espressione restituirà `true` se `unOggetto` è una istanza di una qualsiasi sottoclasse di `VenditaScontata`.

Quindi, se si desidera convertire un oggetto al tipo `VenditaScontata` si può rendere più robusta la conversione nel seguente modo:

```
VenditaScontata vs = null;
if (unOggetto instanceof VenditaScontata) {
    vs = (VenditaScontata)unOggetto;
    System.out.println("vs ora referenzia " + unOggetto);
} else
    System.out.println("vs non e' stato modificato");
```

In questo modo il codice non produce errori anche nel caso in cui `unOggetto` sia di tipo `Vendita` o `Object`.

CASO DI STUDIO
FIGURE DI CARATTERI

La libreria standard di Java include già le classi e i metodi necessari per disegnare delle figure sullo schermo del computer. Ma si supponga che lo schermo del dispositivo su

cui si sta disegnando sia particolarmente economico e non abbia funzionalità grafiche: consente solo la visualizzazione di puro testo.

In questo caso di studio si progetteranno tre classi che possono aggirare questo problema disegnando sullo schermo semplici figure sfruttando solamente i comuni caratteri. L'esempio risolverà il problema sfruttando l'ereditarietà e il polimorfismo.

Si inizi scrivendo una classe che rappresenta una generica forma. Questa classe non rappresenterà una forma specifica, ma sarà utilizzata come classe base per le classi che rappresenteranno le forme vere e proprie. Il nome della classe base è `FormaGenerica` ed è mostrata nel Listato 11.5. I metodi implementati da questa classe rappresentano le operazioni e le proprietà che devono essere comuni a tutte le forme. Ogni sottoclasse di `FormaGenerica` ridefinirà questi metodi in base allo specifico tipo di forma. In questo caso di studio saranno realizzate due sottoclassi di `FormaGenerica`: la classe `Rettangolo` e la classe `Triangolo`.

Inizialmente si identificano le operazioni e le proprietà comuni a tutte le forme. Sicuramente tutte le forme possono essere disegnate. A tale scopo si definiscono due metodi nella classe `FormaGenerica`: il metodo `disegnaQui` che disegna la forma iniziando dalla posizione corrente e il metodo `disegnaDa` che disegna la forma a partire da un dato numero di linee più in basso della riga corrente. Inoltre, sarà necessario un metodo che permetta di visualizzare un certo quantitativo di spazi bianchi in modo da realizzare la figura desiderata. Tale metodo, `saltaSpazi`, sarà un metodo statico poiché accetterà in ingresso il numero di spazi bianchi da visualizzare e quindi non dipende da alcuna variabile di istanza. Sebbene sia un metodo di servizio, non lo si dichiarerà `private`, ma `protected`, in modo da renderlo accessibile alle sottoclassi.

La definizione della classe `FormaGenerica` è fornita nel Listato 11.5.

LISTATO 11.5 La classe base `FormaGenerica`.

MyLab

```
/**
Una classe per disegnare semplici forme sullo schermo utilizzando solo caratteri.
Questa classe disegnera' un asterisco sullo schermo come prova.
Non si intende creare una forma "reale", questa classe e' stata concepita
come classe base per altre classi di forme.
*/
public class FormaGenerica {
    private int scostamento;

    public FormaGenerica() {
        scostamento = 0;
    }

    public FormaGenerica(int scostamentoIniziale) {
        scostamento = scostamentoIniziale;
    }

    public void setScostamento(int nuovoScostamento) {
        scostamento = nuovoScostamento;
    }

    public int getScostamento() {
        return scostamento;
```

```
    }

    public void disegnaDa(int numeroLinee) {
        for (int conteggio = 0; conteggio < numeroLinee; conteggio++)
            System.out.println();
        disegnaQui();
    }

    public void disegnaQui() {
        for (int conteggio = 0; conteggio < scostamento; conteggio++)
            System.out.print(' ');
        System.out.println('*');
    }

    //Scrive il numero indicato di spazi.
    protected static void saltaSpazi(int numero) {
        for (int conteggio = 0; conteggio < numero; conteggio++)
            System.out.print(' ');
    }
}
```

Per chiarezza, il metodo `saltaSpazi` è stato reso statico, perché non dipende dall'oggetto.

Tutte le forme hanno alcune proprietà in comune. Per esempio, ognuna delle forme avrà uno `scostamento` che indica quanto bisogna rientrare dal lato sinistro dello schermo prima di disegnare la forma. La classe base implementa i metodi *set* e *get* per lo `scostamento`. Ogni forma avrà anche una dimensione, ma la dimensione di alcune forme è descritta da un singolo numero, mentre quella di altre da un insieme di numeri. Poiché la dimensione sarà specificata secondo il tipo di forma, questa non è una proprietà che tutte le forme hanno in comune e quindi non sarà inclusa nella classe base.

Si passa ora ad analizzare l'implementazione dei metodi `disegnaDa` e `disegnaQui` della classe base. Il metodo `disegnaDa` ha un solo parametro di tipo `int`, il cui valore indica il numero di linee vuote da inserire prima di disegnare la forma. La forma viene disegnata invocando il metodo `disegnaQui`.

Il metodo `disegnaQui` mostra sullo schermo un numero di spazi pari a `scostamento` e poi produce sullo schermo un asterisco. Questo semplice output è utilizzato per produrre sullo schermo un risultato verificabile, ma non si intende utilizzare questa versione di `disegnaQui` nelle sottoclassi o in altre applicazioni. Il metodo `disegnaQui` sarà, quindi, ridefinito dalle classi che rappresentano i rettangoli e i triangoli.

Si rivolge ora l'attenzione alla classe per disegnare un rettangolo. Si noti che la dimensione di un rettangolo è data dalla sua larghezza e dalla sua altezza, espresse in caratteri. Poiché i caratteri sono più alti che larghi, un rettangolo potrebbe sembrare più alto rispetto alle attese. Per esempio, un rettangolo 5 per 5 non sembrerà un quadrato sullo schermo, ma apparirà come mostrato in Figura 11.2.

La classe che rappresenta i rettangoli si chiama `Rettangolo` e specializza la classe `FormaGenerica`. La definizione di classe inizierà quindi con:

```
public class Rettangolo extends FormaGenerica
```

Ora occorre decidere se ci sono variabili di istanza da aggiungere a quelle già presenti nella classe `FormaGenerica` e se occorre ridefinire alcune definizioni di metodo in

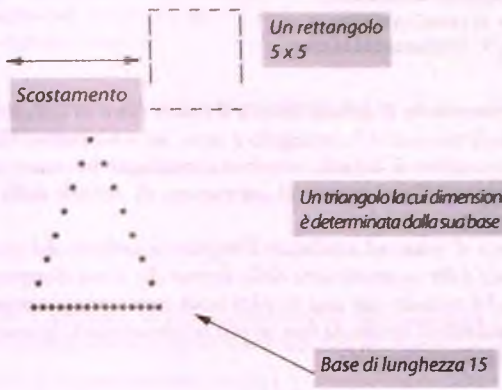


Figura 11.2 Un semplice rettangolo e un semplice triangolo.

`FormaGenerica`. La classe `Rettangolo` potrà usare la variabile di istanza `scostamento` della classe base, ma è necessario introdurre delle variabili di istanza per rappresentare l'altezza e la larghezza del rettangolo. La definizione della classe sarà dunque:

```
public class Rettangolo extends FormaGenerica {
    private int altezza;
    private int larghezza;

    < Definizioni di metodi >
}
```

La definizione della classe `Rettangolo` viene estesa introducendo i costruttori e il metodo `set` per `altezza` e `larghezza`. Si osservi che la variabile di istanza `scostamento` non è stata definita in `Rettangolo`. Infatti, `Rettangolo` eredita i metodi `setScostamento`, `getScostamento`, `disegnaDa` e `disegnaQui` dalla classe `FormaGenerica`. Il metodo `disegnaQui` va però ridefinito in modo che disegni un rettangolo. Bisogna anche ridefinire il metodo `disegnaDa`? Se si osserva il metodo `disegnaDa` del Listato 11.5, si può notare che se `disegnaQui` è correttamente definito, il metodo `disegnaDa` funzionerà per un rettangolo o per qualsiasi altra forma: il polimorfismo assicurerà che `disegnaDa` invochi la versione corretta di `disegnaQui`.

Si osservi il secondo esempio di costruttore, quello che imposta tutte le variabili di istanza ai valori dati dagli argomenti. La variabile di istanza `scostamento` è privata della classe base `FormaGenerica` e `Rettangolo` non può accedervi direttamente. Esistono tuttavia soluzioni alternative: si può invocare il metodo `setScostamento` oppure utilizzare `super` per invocare il costruttore della classe base. È sempre buona norma inizializzare le variabili di istanza in fase di istanziazione di un oggetto e, quindi, all'interno di uno dei costruttori. Per questo motivo, la soluzione scelta invoca il costruttore della classe base come segue:

```
public Rettangolo(int scostamentoIniziale, int altezzaIniziale,
                  int larghezzaIniziale) {
```

```
        super(scostamentoIniziale);
        altezza = altezzaIniziale;
        larghezza = larghezzaIniziale;
    }
```

Analogamente, il costruttore di default invoca il costruttore di default della classe base, impostando così le dimensioni del rettangolo a zero. Se si omettesse l'invocazione esplicita a `super` nel costruttore di default, verrebbe comunque invocato automaticamente. Nella soluzione proposta, l'invocazione al costruttore di default della classe base viene lasciata per trasparenza.

A questo punto si inizia ad analizzare l'implementazione del metodo `disegnaQui`, il quale dipende dalle caratteristiche della forma che si sta disegnando. Si realizzerà l'implementazione del metodo con una tecnica nota come **progettazione top-down**. In questa tecnica si suddivide il lavoro da fare in più sotto-compiti. In questo caso, si devono svolgere i seguenti sotto-compiti:

Algoritmo per disegnare un rettangolo

1. Disegnare la linea superiore.
2. Disegnare le linee laterali.
3. Disegnare la linea inferiore.

Si noti che l'ordine dei sotto-compiti non può essere variata. All'inizio si potrebbe essere tentati di disegnare le linee laterali del rettangolo in due sotto-compiti. Tuttavia, l'output deve essere prodotto una linea dopo l'altra e non è permesso ritornare indietro. Così si devono "tracciare" le due linee laterali insieme.

La definizione del metodo `disegnaQui` è semplice:

```
public void disegnaQui() {
    disegnaLineaOrizzontale();
    disegnaLineeVerticali();
    disegnaLineaOrizzontale();
}
```

Sebbene fin qui sia stato semplice, il grosso del lavoro deve essere ancora svolto. Bisogna ancora definire i metodi `disegnaLineaOrizzontale` e `disegnaLineeVerticali`. Poiché questi sono metodi di supporto, saranno metodi privati.

La logica per `disegnaLineaOrizzontale` non è complicata, come si può vedere dal seguente pseudocodice:

Algoritmo per `disegnaLineaOrizzontale`

1. Visualizza `scostamento` spazi vuoti.
2. Visualizza `larghezza` copie del carattere '-'.
3. `System.out.println()`.

Il compito di scrivere un numero specificato di spazi bianchi (pari a `scostamento`) è svolto dal metodo `saltaSpazi` definito nella classe `FormaGenerica` il quale contiene un semplice ciclo. Il codice finale per il metodo `disegnaLineaOrizzontale` è:

```
private void disegnaLineaOrizzontale() {
    saltaSpazi(getScostamento());
```

```
        for (int conteggio = 0; conteggio < larghezza; conteggio++)
            System.out.print('-');
        System.out.println();
    }
```

Si consideri ora il metodo `disegnaLineeVerticali`. Il suo compito è di disegnare una forma come la seguente:

```
|           |
|           |
|           |
|           |
|           |
```

Poiché ogni linea è identica, si può considerare la visualizzazione di una di queste linee come un sotto-compito. Così la definizione del metodo `disegnaLineeVerticali` è:

```
private void disegnaLineeVerticali() {
    for (int conteggio = 0; conteggio < (altezza - 2); conteggio++)
        disegnaUnaLineaDiLineeVerticali();
}
```

Si osservi che si visualizzano meno linee rispetto all'altezza. La linea orizzontale superiore e quella inferiore introdurranno le due unità aggiuntive di altezza mancanti.

L'unica cosa che è rimasta da fare è definire il metodo di supporto `disegnaUnaLineaDiLineeVerticali`. Poiché si ha già un metodo per il sotto-compito di scrivere gli spazi, lo pseudocodice per `disegnaUnaLineaDiLineeVerticali` è dato dal codice Java:

```
saltaSpazi(getScostamento());
System.out.print('|');
saltaSpazi(larghezza - 2);
System.out.println('|');
```

Di fatto questo conclude il lavoro. Il Listato 11.6 riporta la definizione completa della classe `Rettangolo`.

LISTATO 11.6 La classe `Rettangolo`.

MyLab

```
/**
Una classe per disegnare rettangoli sullo schermo utilizzando i caratteri
della tastiera. Ogni carattere e' piu' alto che largo, cosi' questi
rettangoli sembreranno piu' alti rispetto alle aspettative. La classe
eredita getScostamento, setScostamento e disegnaDa dalla classe FormaGenerica.
*/
public class Rettangolo extends FormaGenerica {
    private int altezza;
    private int larghezza;

    public Rettangolo() {
        super();
        altezza = 0;
        larghezza = 0;
    }
```

```
    public Rettangolo(int scostamentoIniziale, int altezzaIniziale,
                      int larghezzaIniziale) {
        super(scostamentoIniziale);
        altezza = altezzaIniziale;
        larghezza = larghezzaIniziale;
    }

    public void set(int nuovaAltezza, int nuovaLarghezza) {
        altezza = nuovaAltezza;
        larghezza = nuovaLarghezza;
    }

    public void disegnaQui() {
        disegnaLineaOrizzontale();
        disegnaLineeVerticali();
        disegnaLineaOrizzontale();
    }

    private void disegnaLineaOrizzontale() {
        saltaSpazi(getScostamento());
        for (int conteggio = 0; conteggio < larghezza; conteggio++)
            System.out.print('-');
        System.out.println();
    }

    private void disegnaLineeVerticali() {
        for (int conteggio = 0; conteggio < (altezza - 2); conteggio++)
            disegnaUnaLineaDiLineeVerticali();
    }

    private void disegnaUnaLineaDiLineeVerticali() {
        saltaSpazi(getScostamento());
        System.out.print('|');
        saltaSpazi(larghezza - 2);
        System.out.println('|');
    }
}
```

Si passa ad analizzare la classe `Triangolo`. Si supponga di disegnare la punta del triangolo sempre rivolta verso l'alto con la base in basso. Una volta fissata la lunghezza della base, per rendere uniformi i lati, le pendenze sono limitate dal risultato ottenibile rientrando di un carattere per riga. Così una volta scelta la base vengono automaticamente determinati anche i lati del triangolo. La Figura 11.2 mostra un esempio di triangolo.

Il Listato 11.7 contiene la definizione della classe `Triangolo`. Si può progettare questa classe utilizzando la stessa tecnica usata per la classe `Rettangolo`. Si tratteranno solo le parti specifiche del metodo `disegnaQui`. Il metodo `disegnaQui` suddivide il suo lavoro in due sotto-compiti: disegnare la `V` rovesciata per la punta del triangolo e disegnare la linea orizzontale per la base del triangolo.

Il metodo `disegnaPunta` disegna una forma come la seguente:

Si noti che l'intera figura è bilanciata. La rientranza della base dal margine sinistro è esattamente `scostamento`. Procedendo dal basso verso l'alto, ogni linea ha una rientranza maggiore. Alternativamente, percorrendo le linee dall'alto verso il basso (come fa il computer quando visualizza la figura) la rientranza si riduce di un carattere per linea. Così, se la rientranza della punta del triangolo è data dal valore della variabile intera `inizioDellaLinea`, la prima rientranza può essere prodotta con l'invocazione:

```
saltaSpazi(inizioDellaLinea);
```

Una volta prodotta la rientranza, si scrive il singolo asterisco.

Dopo aver scritto l'asterisco per la prima linea, occorre scrivere due asterischi per ogni linea successiva. Si scrive quindi un ciclo che a ogni iterazione diminuisce di una unità il valore di `inizioDellaLinea` per poi eseguire una invocazione di `saltaSpazi` esattamente analoga a quella sopra riportata. Si scrive quindi un asterisco e si saltano alcuni spazi prima di scrivere il secondo asterisco. La dimensione dello spazio tra i due asterischi su una linea aumenta di due unità ogni volta che si scende di una linea. Se si utilizza la variabile intera `larghezzaInterna` per rappresentare il vuoto tra i due asterischi l'implementazione del ciclo sarà:

```
for (int conteggio = 0; conteggio < conteggioLinee; conteggio++) {
    inizioDellaLinea--;
    saltaSpazi(inizioDellaLinea);
    System.out.print('*');
    saltaSpazi(larghezzaInterna);
    System.out.println('*');
    larghezzaInterna = larghezzaInterna + 2;
}
```

La completa definizione del metodo `disegnaPunta` è fornita nel Listato 11.7.

La base del triangolo è una linea di asterischi. Idealmente si dovrebbe utilizzare un numero dispari di asterischi, altrimenti il triangolo sembrerà leggermente sbilanciato. Tuttavia, per mantenere semplice la classe, si dichiara ciò come una precondizione del metodo `set` di `Triangolo` che tuttavia non viene imposta.

Sebbene non si descriverà il processo di verifica in questo caso di studio, tutti i metodi delle classi `FormaGenerica`, `Rettangolo` e `Triangolo` devono essere verificati. Si ricorda, infatti, che ogni metodo dovrebbe essere verificato per attestarne il corretto funzionamento.

Il programma applicativo di esempio nel Listato 11.8 mostra l'uso delle classi introdotte disegnando un triangolo e un rettangolo per formare un'immagine elementare di un abete.

MyLab

LISTATO 11.7 La classe Triangolo.

```
/**
Una classe per disegnare triangoli sullo schermo utilizzando i caratteri
della tastiera. Un triangolo punta verso l'alto. La sua dimensione e'
determinata dalla lunghezza della sua base, che deve essere un intero
dispari. La classe eredita getScostamento, setScostamento e disegnaDa dalla
classe FormaGenerica.
*/
public class Triangolo extends FormaGenerica {
    private int base;

    public Triangolo() {
        super();
        base = 0;
    }

    public Triangolo(int scostamentoIniziale, int baseIniziale) {
        super(scostamentoIniziale);
        base = baseIniziale;
    }

    /**
    Precondizione: nuovaBase e' un intero dispari.
    */
    public void set(int nuovaBase) {
        base = nuovaBase;
    }

    public void disegnaQui() {
        disegnaPunta();
        disegnaBase();
    }

    private void disegnaBase() {
        saltaSpazi(getScostamento());
        for (int conteggio = 0; conteggio < base; conteggio++)
            System.out.print('*');
        System.out.println();
    }

    private void disegnaPunta() {
        //inizioDellaLinea == numero di spazi prima
        //del primo '*' sulla linea. Inizialmente e' impostato
        //al numero di spazi prima del '*' piu' alto.
        int inizioDellaLinea = getScostamento() + base / 2;
        saltaSpazi(inizioDellaLinea);
        System.out.println('*');                    //punta '*'
        int conteggioLinee = base / 2 - 1;          //altezza sopra la base
```

```
        //larghezzaInterna == numero di spazi
        //tra due '*' su una linea.
        int larghezzaInterna = 1;
        for (int conteggio = 0; conteggio < conteggioLinee; conteggio++) {
            //Scendendo di una linea, il primo '*'
            //e' uno spazio in piu' verso sinistra.
            inizioDellaLinea--;
            saltaSpazi(inizioDellaLinea);
            System.out.print('*');
            saltaSpazi(larghezzaInterna);
            System.out.println('*');
            //Scendendo di una linea, l'interno
            //e' piu' largo di due spazi.
            larghezzaInterna = larghezzaInterna + 2;
        }
    }
}
```

LISTATO 11.8 Una dimostrazione di `Triangolo` e `Rettangolo`.

MyLab

```
/**
Un programma che disegna un abete composto da un triangolo e
un rettangolo, entrambi disegnati utilizzando i caratteri
della tastiera.
*/
public class AlberoDemo {
    public static final int RIENTRO = 5;
    public static final int LARGHEZZA_PUNTA_ALBERO = 21;// deve essere dispari
    public static final int LARGHEZZA_BASE_ALBERO = 4;
    public static final int ALTEZZA_BASE_ALBERO = 4;

    public static void main(String[] args) {
        disegnaAlbero(LARGHEZZA_PUNTA_ALBERO, LARGHEZZA_BASE_ALBERO,
                      ALTEZZA_BASE_ALBERO);
    }

    public static void disegnaAlbero(int larghezzaPunta, int larghezzaBase,
                                     int altezzaBase) {
        System.out.println("       Salva le Sequoie!");
        Triangolo puntaAlbero = new Triangolo(RIENTRO, larghezzaPunta);
        disegnaPunta(puntaAlbero);
        Rettangolo troncoAlbero = new Rettangolo(RIENTRO +
            (larghezzaPunta / 2) - (larghezzaBase / 2),
            altezzaBase, larghezzaBase);
        disegnaTronco(troncoAlbero);
    }
```

```
    private static void disegnaPunta(Triangolo puntaAlbero) {
        puntaAlbero.disegnaDa(1);
    }

    private static void disegnaTronco(Rettangolo troncoAlbero) {
        troncoAlbero.disegnaQui(); // o troncoAlbero.disegnaDa(0);
    }
}
```

Esempio di output

```
Salva le Sequoie!

          *
         * *
        *   *
       *     *
      *       *
     *         *
    *           *
   *             *
  *               *
 *                 *
**********************
        ----
        |  |
        |  |
        ----
```

Chiaramente questo esempio fa uso del *binding* dinamico, il cui funzionamento viene brevemente discusso in questo caso di studio. Se si riguarda la definizione del metodo `disegnaDa` nella classe `FormaGenerica` (Listato 11.5), si nota che compare un'invocazione al metodo `disegnaQui`. Se si fosse avuta solo la classe `FormaGenerica`, non ci sarebbe stato nulla di strano. Tuttavia, è stata derivata la classe `Rettangolo` dalla classe base `FormaGenerica`. La classe `Rettangolo` eredita il metodo `disegnaDa` immutato dalla classe `FormaGenerica` e `Rettangolo` ridefinisce la definizione del metodo `disegnaQui`.

Quindi, a seguito delle seguenti istruzioni:

```
Rettangolo r = new Rettangolo(1, 4, 4);
r.disegnaDa(2);
```

e grazie al *binding* dinamico, il metodo `disegnaDa` nella classe `FormaGenerica` non invoca il metodo `disegnaQui` definito in `FormaGenerica`, bensì quello ridefinito nella classe `Rettangolo`, poiché l'oggetto `r` fa riferimento a un oggetto di tipo `Rettangolo`. Se così non fosse, `r.disegnaDa(2)` disegnerebbe un solo asterisco invocando il metodo `disegnaQui` della classe `FormaGenerica`. Cosa che ovviamente non accade come mostrato nell'output di esempio del Listato 11.8.

11.2 Classi astratte

Una classe astratta è una classe che ha alcuni metodi non completamente definiti. Non è possibile creare nuovi oggetti utilizzando il costruttore di una classe astratta, ma è possibile utilizzare una classe astratta come classe base per la definizione di classi derivate.

11.2.1 Concetti di base

Quando è stata scritta la classe `FormaGenerica` nel Listato 11.5, non si aveva l'intenzione di creare oggetti della classe `FormaGenerica`. Questa classe è stata progettata come una classe base per altre classi, come la classe `Rettangolo` nel Listato 11.6.

Sebbene non esista alcuna necessità di istanziare oggetti di tipo `FormaGenerica`, la seguente istruzione è del tutto lecita:

```
FormaGenerica variabileForma = new FormaGenerica();
```

Il motivo per cui tale istruzione è lecita è il fatto che vi è stata la necessità di definire il metodo `disegnaQui` nella classe `FormaGenerica`.

In realtà la definizione che è stata scritta è solo una sorta di "segnaposto": tutto ciò che fa, è disegnare un singolo asterisco. Il metodo non verrà mai utilizzato, poiché non visualizza alcuna figura geometrica. Si sarebbe anche potuta rendere del tutto vuota la definizione del metodo come di seguito:

```
public void disegnaQui() {
}
```

Se ci si pensa bene, effettivamente non si sa dare una risposta alla domanda: "Qual è la forma geometrica di una forma generica?". La risposta più sensata a questa domanda è: "Dipende da che forma si tratta, se è un rettangolo, allora ...".

Tale metodo è stato definito all'interno della classe `FormaGenerica` esclusivamente per poter sfruttare il polimorfismo. In questo modo, né la classe `Rettangolo`, né la classe `Triangolo` hanno dovuto ridefinire il metodo `disegnaQua`, ma hanno fornito esclusivamente la loro versione del metodo `disegnaQui`.

Tuttavia, invece di fornire una definizione "inventata" di un metodo che si pensa di ridefinire in una classe derivata, si può dichiarare il **metodo astratto**:

```
public abstract void disegnaQui();
```

La sintassi per definire un metodo astratto prevede di far precedere all'intestazione del metodo la parola chiave `abstract`, di porre un punto e virgola alla fine dell'intestazione e di omettere il corpo del metodo.

Definire un metodo astratto significa posticipare la sua definizione al momento in cui si saprà effettivamente come definirla. Nel caso specifico, è come dire che: "ogni figura avrà un metodo `disegnaQui`, ma in questa classe non si sa come implementarlo".

Un metodo astratto deve essere ridefinito da ogni classe derivata dalla classe base astratta. Chiaramente, questo vale se la classe derivata sa come definirlo. Nell'ipotesi che la classe derivata sappia definirlo, includere un metodo astratto in una classe base è un modo per obbligare la classe derivata a definire un particolare metodo.

Java richiede che se una classe ha almeno un metodo astratto, la classe deve essere dichiarata astratta. Si fa ciò includendo la parola chiave `abstract` nell'intestazione della definizione della classe:

```
public abstract class FormaGenerica {
    ...
```

Una classe definita in questo modo è detta **classe astratta**. Se una classe è astratta, non si possono creare oggetti per quella classe: può essere usata solo come una classe base per derivare altre classi. Per questa ragione, una classe astratta è detta anche **classe base astratta**. All'opposto, una classe non astratta è chiamata **classe concreta**.

Nel caso specifico, la seguente istruzione non sarebbe più lecita:

```
FormaGenerica f = new FormaGenerica();     //ILLEGALE perché
                                           //FormaGenerica è astratta
```

Ma ciò non costituisce un problema, poiché non ci sarà mai l'esigenza di creare oggetti di tipo `FormaGenerica`, mentre esisteranno certamente oggetti di tipo `Rettangolo` e `Triangolo`.

Nel Listato 11.9, è stata rivisitata la classe `FormaGenerica` in modo da renderla astratta ed è stata chiamata `FormaBase`. Si noti che il precedente caso di studio continua a funzionare se le intestazioni delle classi `Rettangolo` e `Triangolo` si modificano in:

```
public class Rettangolo extends FormaBase
public class Triangolo extends FormaBase
```

anche se la classe `FormaBase` non può essere istanziata.

Sebbene la classe `FormaBase` sia astratta, non tutti i suoi metodi sono astratti. Tutte le definizioni di metodo, tranne `disegnaQui`, sono esattamente le stesse dell'originale `FormaGenerica`. Queste sono definizioni complete e non utilizzano la parola chiave `abstract`. Infatti, quando ha senso definire il corpo di un metodo di una classe astratta, questo dovrebbe essere fornito. In altre parole, quando esiste una specifica per la definizione di un metodo, questo va implementato. In tal modo, la classe base conterrà tutti quei dettagli che non dovranno poi essere ridefiniti nelle classi derivate.

Perché esistono le classi astratte? Perché semplificano le cose. Si è già spiegato nel caso di studio che la classe `FormaGenerica` evita di duplicare le definizioni di metodi come `disegnaDa` per tutti i tipi di forma. Però è stato necessario scrivere una definizione inutile per il metodo `disegnaQui`. Un metodo astratto facilita la definizione di una classe base, evitando al programmatore di scrivere metodi inutili. Se un metodo dovrà sempre essere ridefinito, basta renderlo astratto; di conseguenza, la classe sarà astratta. Si noti che il metodo `disegnaDa` del Listato 11.9 include un'invocazione del metodo `disegnaQui`. Se il metodo astratto `disegnaQui` fosse stato omesso dalla definizione di classe, questa invocazione di `disegnaQui` sarebbe stata illegale.

MyLab

LISTATO 11.9 La classe astratta `FormaBase`.

```
/**
Una classe base astratta per disegnare semplici forme
sullo schermo utilizzando i caratteri.
*/
public abstract class FormaBase {
    private int scostamento;

    public FormaBase() {
        scostamento = 0;
    }
```

```
    public FormaBase(int scostamentoIniziale) {
        scostamento = scostamentoIniziale;
    }

    public void setScostamento(int nuovoScostamento) {
        scostamento = nuovoScostamento;
    }

    public int getScostamento() {
        return scostamento;
    }

    public void disegnaDa(int numeroLinee) {
        for (int conteggio = 0; conteggio < numeroLinee; conteggio++)
            System.out.println();
        disegnaQui();
    }

    public abstract void disegnaQui();

    //Scrive il numero indicato di spazi.
    protected static void saltaSpazi(int numero) {
        for (int conteggio = 0; conteggio < numero; conteggio++)
            System.out.print(' ');
    }
}
```

Metodi astratti

Un metodo viene definito astratto se non si è in grado di fornire la definizione del suo corpo o se si vuole delegare la sua definizione alla classi derivate. È possible inserire invocazioni di un metodo astratto all'interno di altri metodi. Grazie al *binding* dinamico, verrà effettivamente eseguito il metodo concreto che ha ridefinito quello astratto. La sintassi per definire un metodo astratto prevede di far precedere all'intestazione del metodo la parola chiave `abstract`, di porre un punto e virgola alla fine dell'intestazione e di omettere il corpo del metodo.

Sintassi

```
public abstract tipo_di_ritorno nome(parametri);
```

Esempi

```
public abstract void disegnaQui();
public abstract int calcolaArea();
```

Classi astratte

Se una classe è definita astratta, non è possibile creare oggetti utilizzando uno dei suoi costruttori, ma è possibile utilizzare una classe astratta come classe base per la definizione di classi derivate. Se una classe definisce uno o più metodi astratti deve essere definita astratta. Se la classe definisce tutti i metodi, ma si vuole impedire la creazione di oggetti di quella classe, basta definire la classe come astratta.

```
public abstract class nome_classe
```

Esempi

```
public abstract class FormaBase
public abstract class Prodotto
```

11.2.2 La classe astratta è un tipo

Sebbene non sia possibile creare un oggetto da una classe astratta, ha perfettamente senso avere una variabile dichiarata del tipo di una classe astratta, per esempio `FormaBase`. A tale variabile può essere assegnato un oggetto di una qualsiasi classe discendente da `FormaBase`.

Istruzioni come quelle che seguono sono quindi legali:

```
Rettangolo r = new Rettangolo(1, 4, 4);
FormaBase s = r;
s.disegnaQui();
```

oppure:

```
FormaBase s = new Rettangolo(1, 4, 4);
s.disegnaQui();
```

Questo perché un `Rettangolo` è anche una `FormaBase` grazie all'ereditarietà.

È importante notare che la decisione riguardo a quale definizione di metodo utilizzare dipende dalla posizione dell'oggetto nella catena di ereditarietà e non dal tipo della variabile dell'oggetto.

Infatti, l'oggetto è come se ricordasse di essere stato creato come un `Rettangolo`. Nei casi sopra presentati, `s.disegnaQui()` userà la definizione di `disegnaQui` fornita in `Rettangolo`, non la definizione di `disegnaQui` fornita in `FormaBase`. Per determinare quale definizione di `disegnaQui` usare, Java controlla quale classe è stata usata quando l'oggetto è stato creato utilizzando `new`.

Nel caso in cui non trovasse la definizione del metodo invocato nella classe con cui è stato istanziato l'oggetto, Java risale la catena di ereditarietà a partire dalla classe con cui è stato istanziato l'oggetto invocante il metodo, finché non trova in qualche classe antenata il metodo invocato. Il primo che trova è quello che sarà eseguito.

Una classe astratta è un tipo

Non è possibile creare un oggetto da una classe astratta, ma ha perfettamente senso avere una variabile del tipo di una classe astratta, per esempio `FormaBase`. A quella variabile può essere assegnato un oggetto di una qualsiasi delle classi discendenti da `FormaBase`.

MyLab

Video 11.2 Definire una classe astratta

Gli oggetti sanno come agire

Quando viene invocato un metodo ridefinito, il metodo che verrà eseguito è quello definito nella classe usata per creare l'oggetto usando l'operatore new (o in una classe antenata) e non è determinato dal tipo della variabile che fa riferimento all'oggetto. Una variabile di una qualsiasi classe antenata può far riferimento a un oggetto di una classe discendente, ma l'oggetto agirà secondo le definizioni dei metodi fornite nella classe con cui è stato istanziato o, se assenti, nella prima classe antenata in cui sono definiti. Il tipo di variabile non ha importanza. Quello che importa è il nome della classe utilizzata per creare l'oggetto. Questo è il motivo per cui Java utilizza il *binding* dinamico.

11.2.3 Ulteriori dettagli

Una classe astratta può avere un numero qualsiasi di metodi astratti oltre a eventuali metodi non astratti. Se una classe derivata non è in grado di definire uno o più metodi della classe base astratta, anche lei sarà una classe astratta e dovrà includere nella propria definizione la parola chiave `abstract`.

Il concetto verrà illustrato per mezzo di un esempio un po' irrealistico, ma molto efficace. Si immagini di definire una classe `Animale` che rappresenta un qualsiasi tipo di animale: da un gatto a un elefante. Ogni animale, oltre ad avere un nome, è caratterizzato dal fatto che dorme e si esprime con un qualche verso. Ogni animale, però, "parla" e dorme a modo proprio e quindi i metodi `parla` e `dormi` sono dichiarati come astratti, poiché in `Animale` non si sarebbe in grado di fornire una definizione. Si supponga che tutti i felini dormano nella stessa maniera. Si definisce quindi una classe derivata `Felino` che definisce il metodo `dormi`. La classe però non è in grado di definire il comportamento del metodo `parla`, perché ogni felino lo fa a modo proprio. Si può immaginare, a titolo esemplificativo, che un gatto dica 'Miao' e un leone 'Roar'. Ne consegue che la classe `Felino` deve essere dichiarata anch'essa astratta, perché non definisce il comportamento del metodo astratto `parla` ereditato da `Animale`. Infine, le classi `Gatto` e `Leone`, derivate dalla classe astratta `Felino`, definiscono il comportamento del metodo ereditato `parla` e quindi non sono astratte. La classe `Gatto` definisce poi un ulteriore metodo che si chiama `faiLeFusa`.

La Figura 11.3 illustra il diagramma delle classi appena descritte. Si noti che, essendo `Animale` e `Felino` classi astratte, sono riportate in corsivo. La stessa convenzione stilistica è applicata ai metodi astratti.

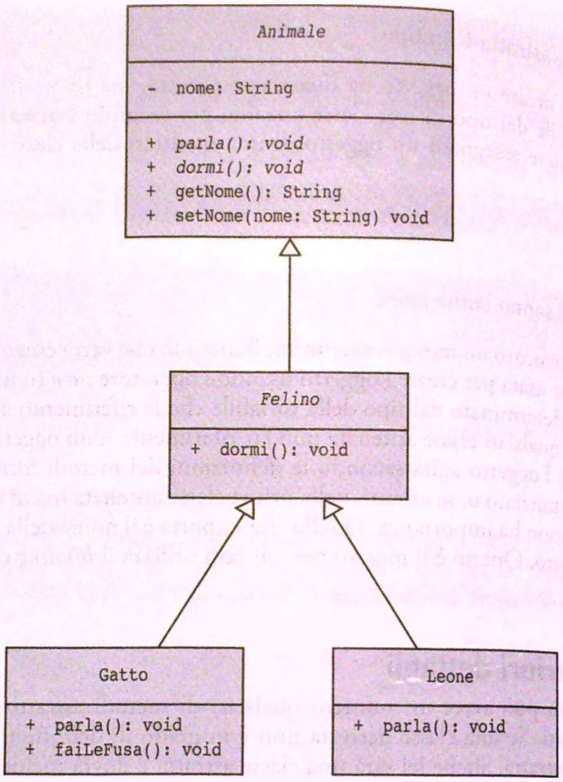


Figura 11.3 Il diagramma delle classi per gli animali.

Le classi `Animale`, `Felino` e `Gatto` sono riportate rispettivamente nei Listati 11.10, 11.11 e 11.12. La classe `Leone` non viene presentata, poiché è sostanzialmente simile alla classe `Gatto`.

MyLab

FIGURA 11.10 La classe astratta `Animale`.

```
/**
Una generica classe Animale. Un animale ha un nome.
I comportamenti che un qualsiasi animale ha sono: dormi e parla
*/
public abstract class Animale {
    private String nome;

    public Animale() {
        this("Nessun nome");
    }

    public Animale(String nome) {
        this.nome = nome;
    }
```

```
    public abstract void parla();

    public abstract void dormi();

    public String getNome() {
        return nome;
    }

    public void setNome(String nuovoNome) {
        nome = nuovoNome;
    }
}
```

FIGURA 11.11 La classe astratta `Felino`.

```
/**
Un felino è un particolare tipo di animale.
La classe definisce il metodo dormi.
*/
public abstract class Felino extends Animale {

    public Felino() {
        super();
    }

    public Felino(String nome) {
        super(nome);
    }

    public void dormi() {
        System.out.println("Ronf...");
    }
}
```

La classe è astratta perché non definisce il corpo del metodo ereditato `parla`.

FIGURA 11.12 La classe concreta `Gatto`.

```
/**
Un gatto è un particolare tipo di felino. La classe definisce
il metodo parla e un ulteriore metodo faiLeFusa.
*/
public class Gatto extends Felino {

    public Gatto() {
        super();
    }
```

```
    }

    public void parla() {
        System.out.println("Miao");
    }

    public void faiLeFusa() {
        System.out.println("Prrrr");
    }
}
```

Si immagini ora di aver definito le seguenti istruzioni in un programma:

```
Animale gatto1 = new Gatto("Miguel");
gatto1.parla();
```

Entrambe le istruzioni sono del tutto lecite. La prima è lecita perché `Gatto` è un `Animale` grazie all'ereditarietà, a prescindere dal fatto che `Animale` sia astratta. La seconda è lecita perché il metodo `parla` è stato definito in `Animale`. In fase di compilazione, infatti, i metodi vengono cercati nella classe con cui è stata dichiarata la variabile. Se non vengono trovati, vengono cercarti nelle classi antenate risalendo la gerarchia di ereditarietà. In fase di esecuzione, il metodo `parla` che viene eseguito è quello definito nella classe `Gatto`. Tutto ciò grazie al polimorfismo.

Per le motivazioni riportate sopra, anche le seguenti istruzioni risultano lecite:

```
Felino gatto2 = new Gatto("Speedy");
gatto2.parla();
```

Infine, la seguente istruzione:

```
gatto1.faiLeFusa();     //ILLEGALE!!
```

non è lecita. Infatti, la classe `Animale` non definisce tale metodo. Come precisato sopra, in fase di compilazione, l'esistenza di un metodo viene verificata a partire dalla classe con cui è stata dichiarata la variabile. La stessa cosa vale anche per:

```
gatto2.faiLeFusa();     //ILLEGALE!!
```

perchè neanche la classe `Felino` definisce il metodo `faiLeFusa`.

Quando una classe è astratta, non è possibile creare oggetti di quella classe. Se una classe definisce uno o più metodi astratti, si è costretti a definire la classe come astratta. Esistono, però, situazioni in cui non ha senso creare oggetti di una certa classe anche se la classe è totalmente definita (cioè non dovrebbe essere dichiarata astratta). Per esempio, si immagini di realizzare un sistema di gestione delle vendite. La classe base è `Prodotto` che ha variabili di istanza per rappresentare una descrizione del prodotto in formato stringa e un prezzo. Si hanno poi delle classi specifiche, tipo: `Monitor`, `Tastiera`, `Schermo` e così via che specializzano la classe base `Prodotto`. Sebbene la classe `Prodotto` sia totalmente definita, in realtà non ha senso avere degli oggetti di quel tipo. Ha senso parlare di un monitor, di una tastiera, ma non di un prodotto generico. In questi casi, la classe viene definita astratta in modo tale da evitare che venga istanziata perché, se istanziata, non avrebbe alcun significato.

Compilazione senza errori

Il compilatore verifica l'esistenza di una definizione di metodo a partire dalla classe con cui è stata definita la variabile che fa riferimento all'oggetto su cui poi viene invocato il metodo. Se non viene trovata in quella classe, il compilatore la cerca nella classe base; se anche lì non viene trovata, la cerca nella classe antenata e così via fino a cercarla nella classe `Object`. Nel caso in cui non venga trovata, il compilatore produce un errore di compilazione nonostante il fatto che quel metodo sia stato definito nella classe con cui è stato creato l'oggetto che ha invocato il metodo.

11.3 Interfacce

Il Capitolo 8 ha definito l'interfaccia di una classe come la parte di una classe che indica a un programmatore come utilizzarla. In particolare, l'interfaccia di una classe è costituita dalle intestazioni dei metodi pubblici e dalle costanti pubbliche della classe, insieme con alcuni commenti esplicativi. Conoscendo solo l'interfaccia di una classe, un programmatore può scrivere un programma che utilizza la classe senza sapere nulla dell'implementazione della classe stessa. Finora, l'interfaccia di una classe è stata integrata nella sua definizione. Tuttavia, Java consente di separare l'interfaccia di una classe dalla sua implementazione, realizzando due file sorgenti distinti. Il concetto di interfaccia di classe si traduce in una interfaccia (*interface*) Java.

11.3.1 Interfacce di classi

Nel Capitolo 1 si è immaginato che una persona chiamasse i suoi animali per cena fischiando. Ogni animale ha risposto a suo modo: alcuni hanno corso, alcuni hanno volato e alcuni hanno nuotato. Si supponga, per esempio, che gli animali siano capaci di:

- essere chiamati;
- mangiare;
- rispondere a un comando.

Si potrebbero specificare le intestazioni dei seguenti metodi per questi comportamenti:

```
/** Imposta il nome di un animale a nomeAnimale. */
public void setNome(String nomeAnimale)

/** Restituisce vero se un animale mangia il cibo dato. */
public boolean mangia(String cibo)

/** Restituisce una descrizione della risposta di un animale al comando dato. */
public String rispondi(String comando)
```

Queste intestazioni di metodo possono costituire un'interfaccia di una classe.

Ora si immagini che ognuna delle tre classi `Cane`, `Uccello` e `Pesce` implementi tutti i suddetti metodi. Gli oggetti di queste classi pertanto hanno gli stessi comportamenti, cioè ogni oggetto può essere chiamato, può mangiare e può reagire. La natura di questi comportamenti, tuttavia, può essere differente da oggetto a oggetto. Per esempio,

cani, uccelli e pesci possono anche rispondere a un comando, ma il modo con cui lo fanno è differente.

Si immagini un'istruzione Java come questa:

```
String risposta = mioAnimale.rispondi("Vieni!");
```

Questa istruzione è legittima indipendentemente dal fatto che `mioAnimale` faccia riferimento a un oggetto di tipo `Cane`, `Uccello` o `Pesce`. Il valore della stringa di risposta, tuttavia, differisce a seconda del tipo di oggetto a cui fa riferimento `mioAnimale`. Si può sostituire un tipo di oggetto con l'altro senza alcun problema, finché ognuna delle tre classi implementa il metodo `rispondi` a prescindere dalla sua implementazione. Come si può essere sicuri che una classe implementi determinati metodi? Per la risposta occorre proseguire con la lettura.

11.3.2 Interfacce Java

Un'**interfaccia Java** (*Java interface*) è un componente di un programma che contiene le intestazioni di un certo numero di metodi. Un'interfaccia può anche definire delle costanti pubbliche. Inoltre, un'interfaccia potrebbe includere i commenti che descrivono i metodi, cosicché un programmatore avrà tutte le informazioni necessarie per implementarli.

L'interfaccia di classe scritta nel paragrafo precedente è quasi completa per poter essere implementata in un'interfaccia Java. Un'interfaccia Java inizia come una definizione di classe, tranne per il fatto che utilizza la parola riservata `interface` al posto di `class`. Una definizione di interfaccia inizia quindi con:

```
public interface nome_interfaccia
```

al posto di:

```
public class nome_classe
```

L'interfaccia può contenere un numero qualsiasi di intestazioni di metodi pubblici, ognuna seguita da un punto e virgola. Per esempio, il Listato 11.13 contiene un'interfaccia Java per oggetti i cui metodi restituiscono i loro perimetri e aree.

MyLab

FIGURA 11.13 Un'interfaccia Java.

```
/**
Un'interfaccia che contiene metodi che calcolano
e restituiscono il perimetro e l'area di un oggetto
*/
public interface Misurabile {
    /**
    Restituisce il perimetro.
    */
    public double getPerimetro();

    /**
    Restituisce l'area.
    */
    public double getArea();
}
```

Non dimenticarsi del punto e virgola alla fine di ogni definizione di metodo.

Per convenzione, il nome di un'interfaccia inizia con una lettera maiuscola, proprio come il nome di una classe. Si può memorizzare un'interfaccia in un file distinto, utilizzando un nome che inizia con il nome dell'interfaccia, seguito da `.java`. Per esempio, l'interfaccia presentata nel Listato 11.13 è salvata nel file `Misurabile.java`.

Un'interfaccia non dichiara alcun costruttore. I metodi di un'interfaccia devono essere pubblici. Un'interfaccia può anche definire un numero qualsiasi di costanti pubbliche. Essa non contiene, tuttavia, variabili di istanza o definizioni complete di metodo; in pratica, i metodi non possono avere un corpo.

Le interfacce Java

Sintassi

```
public interface nome_interfaccia {
    definizioni_di_costanti_pubbliche
    . . .
    intestazione_del_metodo_pubblico_1;
    . . .
    intestazione_del_metodo_pubblico_n;
}
```

Esempio

```
/**
    Un'interfaccia di metodi statici per convertire misurazioni
    da piedi a pollici e viceversa
*/
public interface Convertibile {
    public static final int POLLICI_PER_PIEDE = 12;
    public static double convertiInPollici(double piedi);
    public static double convertiInPiedi(double pollici);
}
```

11.3.3 Implementare un'interfaccia

Quando si scrive una classe che definisce i metodi dichiarati in un'interfaccia, si dice che la classe *implementa* l'interfaccia. Una classe che implementa un'interfaccia deve definire un corpo per ogni metodo specificato nell'interfaccia. Se non definisce il corpo di tutti i metodi, deve essere dichiarata astratta. La classe potrebbe anche definire metodi non dichiarati nell'interfaccia. Cioè, un'interfaccia non deve necessariamente dichiarare ogni metodo definito in una classe. Inoltre, una classe può implementare più di un'interfaccia.

Per implementare un'interfaccia, si devono fare due cose.

1. Includere l'espressione:

   `implements` *nome_interfaccia*

   all'inizio della definizione di classe. Per implementare più di un'interfaccia, è sufficiente elencare il nome di tutte le interfacce, separati da una virgola, come segue:

   ```
   implements MiaInterfaccia, TuaInterfaccia
   ```

2. Definire ogni metodo dichiarato nell'interfaccia (o nelle interfacce) per creare una classe concreta. In caso contrario, sarà una classe astratta.

In questo modo, un programmatore può indicare ad altri programmatori che una classe definisce determinati metodi.
Per esempio, per implementare l'interfaccia `Misurabile` mostrata nel Listato 11.13, una classe `Quadrato` deve iniziare come segue:

```
public class Quadrato implements Misurabile
```

La classe, per essere concreta, deve anche implementare i due metodi `getPerimetro` e `getArea`. La definizione completa della classe `Quadrato` è presentata nel Listato 11.14.

Altre classi, come la classe `Cerchio` mostrata nel Listato 11.15, possono implementare l'interfaccia `Misurabile`. Si noti che `Cerchio` definisce il metodo `getCirconferenza` in aggiunta ai metodi dichiarati nell'interfaccia. Non è insolito per una classe definire due metodi che eseguono lo stesso compito. Questo garantisce maggiore praticità ai programmatori che usano la classe, ma che preferiscono usare un nome più familiare per un determinato metodo. Si noti, tuttavia, che `getCirconferenza` invoca `getPerimetro` invece di eseguire un suo calcolo. Così facendo, si facilita la gestione della classe. Per esempio, se si scoprisse un problema con il codice di `getPerimetro`, sistemandolo si correggerebbe il problema anche in `getCirconferenza`.

MyLab

LISTATO 11.14 Un'implementazione dell'interfaccia `Misurabile`.

```
/**
Una classe che rappresenta quadrati
*/
public class Quadrato implements Misurabile {
    private double lato;

    public Quadrato(double ilLato) {
        super();
        lato = ilLato;
    }

    public double getArea() {
        return lato * lato;
    }

    public double getPerimetro() {
        return lato * 4;
    }
}
```

LISTATO 11.15 Un'altra implementazione dell'interfaccia `Misurabile`.

```
/**
Una classe che rappresenta cerchi
*/
public class Cerchio implements Misurabile {
   private double raggio;

   public Cerchio(double ilRaggio) {
      raggio = ilRaggio;
   }

   public double getPerimetro() {
      return 2 * Math.PI * raggio;
   }

   public double getCirconferenza() {
      return getPerimetro();
   }

   public double getArea() {
      return Math.PI * raggio * raggio;
   }
}
```

Questo metodo non è dichiarato nell'interfaccia.

Chiama un altro metodo invece di riscrivere il suo corpo.

Le interfacce aiutano i progettisti e i programmatori

Scrivere un'interfaccia è un modo con cui il progettista di una classe specifica i metodi a un'altro programmatore. Implementare un'interfaccia è un modo per un programmatore per garantire che una classe definisca determinati metodi.

Classi differenti possono implementare la stessa interfaccia

Classi differenti possono implementare la stessa interfaccia, in genere in modi differenti. Per esempio, più classi possono implementare l'interfaccia `Misurabile` e fornire la propria versione dei metodi `getPerimetro` e `getArea`.

Implementare un'interfaccia

Una classe può implementare una o più interfacce.

Sintassi

```
public class nome_classe implements interfaccia_1, interfaccia_2, ..., interfaccia_n
```

Esempi

```
public class Quadrato implements Misurabile
public class MiaClasse implements Interfaccia1, Interfaccia2
```

Quando una classe implementa una o più interfacce deve implementare tutti i metodi in esse definiti. In caso contrario, dovrà essere dichiarata astratta.

11.3.4 Un'interfaccia come un tipo

Un'interfaccia è un tipo riferimento. Così, si può scrivere un metodo che ha un parametro di un tipo d'interfaccia, per esempio un parametro di tipo `Misurabile`. Si supponga che un programma definisca i seguenti metodi:

```
public static void visualizza(Misurabile figura) {
    double perimetro = figura.getPerimetro();
    double area = figura.getArea();
    System.out.println("Perimetro = " + perimetro + ", area = " + area);
}
```

Il programma può invocare questo metodo passandogli un oggetto di una qualsiasi classe che implementi l'interfaccia `Misurabile`.

Per esempio, un programma potrebbe contenere le seguenti istruzioni:

```
Misurabile scatola = new Quadrato(5.0);
Misurabile disco = new Cerchio(5.0);
```

Anche se il tipo di entrambe le variabili è `Misurabile`, gli oggetti che si riferiscono a `scatola` e `disco` hanno i metodi `getPerimetro` e `getArea` definiti diversamente. La variabile `scatola` si riferisce a un oggetto di tipo `Rettangolo`; mentre `disco` si riferisce a un oggetto di tipo `Cerchio`. Così l'invocazione:

```
visualizza(scatola);
```

visualizza:

```
Perimetro = 20.0, area = 25.0
```

mentre l'invocazione:

```
visualizza(disco);
```

visualizza:

```
Perimetro = 31.4, area = 78.5
```

Le classi `Rettangolo` e `Cerchio` implementano la stessa interfaccia, così si è in grado di sostituire un'istanza di una con un'istanza dell'altra quando si invoca il metodo `visualizza`.

Una variabile di un tipo d'interfaccia può far riferimento a un oggetto di una classe che implementa l'interfaccia. Come si è detto in precedenza nel capitolo parlando delle classi, anche per quanto riguarda le interfacce, occorre prestare attenzione ai metodi che si invocano: se non sono presenti nell'interfaccia il compilatore genererà un errore.

Così, nell'esempio, si può utilizzare la variabile `disco` per invocare solo metodi definiti nell'interfaccia `Misurabile`. L'invocazione di `getCirconferenza` in:

```
System.out.println(disco.getCirconferenza());     //ILLEGALE!
```

risulta illegale, perché `getCirconferenza` non è il nome di un metodo dell'interfaccia `Misurabile`. In questa invocazione, la variabile `disco` è di tipo `Misurabile`, ma l'oggetto cui fa riferimento `disco` è ancora un oggetto di tipo `Cerchio`. Così, sebbene l'oggetto abbia il metodo `getCirconferenza`, il compilatore non lo sa! Per rendere valida l'invocazione, si ha bisogno di una conversione di tipo, come la seguente:

```
Cerchio c = (Cerchio)disco;
System.out.println(c.getCirconferenza());     //Legale
```

Cosa è legale e cosa succede

Il tipo di una variabile determina i nomi dei metodi che possono essere utilizzati, ma è il tipo dell'oggetto referenziato dalla variabile che determina quale definizioni dei metodi saranno utilizzate.

MyLab

Video 1
interfacc

Il binding dinamico e il polimorfismo si applicano anche alle interfacce

Il binding dinamico si applica alle interfacce esattamente come alle classi. Ciò consente l'intercambiabilità di oggetti di classi diverse, purché implementino la stessa interfaccia. Questa funzionalità – denominata polimorfismo – permette a oggetti diversi di utilizzare definizioni di metodi diverse a parità di nome dei metodi.

11.3.5 Estendere un'interfaccia

È possibile definire una nuova interfaccia che **estende** (*extends*) un'interfaccia già esistente utilizzando una sorta di ereditarietà. In questo modo, si può creare un'interfaccia formata da un insieme di metodi: quelli da lei definiti e quelli "ereditati".

Per esempio, si considerino le classi di animali presentate all'inizio di questo paragrafo e la seguente interfaccia:

```
public interface Nominabile {
    public void setNome(String nomeAnimale);
    public String getNome();
}
```

Si può estendere `Nominabile` per creare l'interfaccia `Chiamabile`:

```
public interface Chiamabile extends Nominabile {
    public void vieni(String nomeAnimale);
}
```

Una classe concreta che implementa `Chiamabile` deve implementare i metodi `vieni`, `setNome` e `getNome`.

Si possono anche combinare più interfacce a formare una nuova interfaccia. Per esempio, si supponga che, in aggiunta alle precedenti due interfacce, si definiscano le seguenti interfacce:

```
public interface Capace {
    public void ascolta();
    public String rispondi();
}

public interface Ammaestrabile extends Chiamabile, Capace {
    public void siedi();
    public String parla();
    public void sdraiati();
}
```

Una classe concreta che implementa `Ammaestrabile` deve implementare i metodi `setNome`, `getNome`, `vieni`, `ascolta` e `rispondi` e anche i metodi `siedi`, `parla` e `sdraiati`.

CASO DI STUDIO
L'INTERFACCIA `Comparable`

Java offre molte interfacce predefinite, utilizzate da varie classi. Una di queste è l'interfaccia `Comparable`, utilizzata per imporre un ordinamento sugli oggetti che la implementano. L'interfaccia `Comparable` contiene l'intestazione del solo metodo `compareTo`, che deve essere quindi definito per ogni classe che implementi l'interfaccia:

```
public int compareTo(Object altro)
```

L'interfaccia consente di specificare come un oggetto vada confrontato con un altro definendo quando uno dei due "viene prima", "viene dopo" o "è uguale" all'altro. È responsabilità del programmatore mantenere la consistenza del confronto. Per esempio, se A viene prima di B e B prima di C, non deve poter accadere che C venga prima di A.

Il metodo `compareTo` dovrà restituire:

- un numero negativo, se l'oggetto sul quale è chiamato "viene prima" del parametro `altro`;
- zero, se l'oggetto sul quale è chiamato "è uguale" al parametro `altro`;
- un numero positivo, se l'oggetto sul quale è chiamato "viene dopo" il parametro `altro`.

Come esempio, si consideri il modo di dire secondo il quale non si possono confrontare mele e arance. Si vedrà che questa regola non si applica alle classi Java, dato che è possibile definire il metodo `compareTo` come si vuole. Un primo tentativo di definire una classe `Frutto` per rappresentare sia mele che arance potrebbe essere quello riportato nel Listato 11.16. Questa semplice classe utilizza una stringa per immagazzinare il nome del frutto e offre i metodi *get* e *set* corrispondenti. Il costruttore richiede come parametro il nome del frutto. Questo primo tentativo non utilizza le interfacce.

LISTATO 11.16 Primo tentativo di definizione della classe `Frutto`.

```
public class Frutto {
     private String nomeFrutto;

     public Frutto() {
         this("");
     }

     public Frutto(String nome) {
         nomeFrutto = nome;
     }

     public void setNome(String nome) {
         nomeFrutto = nome;
     }

     public String getNome() {
         return nomeFrutto;
     }
}
```

Il Listato 11.17 mostra un breve programma di esempio che utilizza la classe `Frutto`. In questo esempio si crea un array che contiene quattro oggetti `Frutto`, dopodiché si prova a ordinare l'array utilizzando il metodo `Arrays.sort` descritto nel Capitolo 6.

LISTATO 11.17 Programma per l'ordinamento di un array di oggetti `Frutto`.

```
import java.util.Arrays;

public class DemoFrutto {
     public static void main(String[] args) {
         Frutto[] frutto = new Frutto[4];
         frutto[0] = new Frutto("Arancia");
         frutto[1] = new Frutto("Mela");
         frutto[2] = new Frutto("Banana");
         frutto[3] = new Frutto("Pera");

         Arrays.sort(frutto);
         // Stampa l'array ordinato
         for (Frutto f : frutto) {
             System.out.println(f.getNome());
         }
     }
}
```

Il programma del Listato 11.17 verrà compilato correttamente, ma produrrà il seguente errore in fase di esecuzione:

```
Exception in thread "main" java.lang.ClassCastException: Frutto cannot be
cast to java.lang.Comparable
```

Ciò accade perché Java non sa come confrontare due istanze della classe `Frutto` per vedere quale "viene prima" dell'altra. Più precisamente, il metodo `Arrays.sort` è stato implementato nell'ipotesi che gli oggetti contenuti nell'array da ordinare offrano il metodo `compareTo`, in accordo con le specifiche dell'interfaccia `Comparable`. Il metodo prova a invocare `compareTo` sugli elementi dell'array (per verificare, ad esempio, se `frutto[0]` deve essere posto prima di `frutto[1]`) per ordinarli, ma dato che nell'esempio il metodo non esiste si verifica un errore.

La soluzione a questo problema è garantire che la classe `Frutto` implementi l'interfaccia `Comparable` con il metodo `compareTo`. Un possibile criterio per confrontare due frutti è quello di utilizzare l'ordinamento lessicografico dei rispettivi nomi. L'ordine lessicografico coincide con quello alfabetico quando i caratteri di entrambe le stringhe sono tutti minuscoli o tutti maiuscoli. Ad esempio, le arance verrebbero prima delle mele, perché la parola "arancia" compare, nell'ordinamento lessicografico, prima della parola "mela". Per ottenere questo comportamento si può sfruttare il metodo `compareTo` definito nella classe `String`. Quindi, date due stringhe `str1` e `str2`, l'istruzione

```
str1.compareTo(str2)
```

restituirà un numero negativo, uno zero o un numero positivo a seconda che, rispettivamente, `str1` compaia prima, sia uguale o compaia dopo `str2` secondo l'ordine lessicografico. Il metodo `compareTo` per la classe `Frutto` può quindi restituire il risultato della chiamata al metodo `compareTo` sui nomi dei due frutti da confrontare. Il Listato 11.18 contiene una nuova versione della classe `Frutto`, nella quale sono state evidenziate queste modifiche.

MyLab

LISTATO 11.18 Una versione della classe `Frutto` che implementa `Comparable`.

```
public class Frutto implements Comparable {

    private String nomeFrutto;

    public Frutto() {
        this("");
    }

    public Frutto(String nome) {
        nomeFrutto = nome;
    }

    public void setNome(String nome) {
        nomeFrutto = nome;
    }
```

```
    public String getNome() {
        return nomeFrutto;
    }

    public int compareTo(Object o) {
        if ((o != null) && (o instanceof Frutto)) {
            Frutto altroFrutto = (Frutto) o;
            return (nomeFrutto.compareTo(altroFrutto.nomeFrutto));
        }
        return -1; // Default nel caso l'oggetto non sia un Frutto
    }
}
```

Ora il programma del Listato 11.17 funzionerà e produrrà il risultato seguente:

```
Arancia
Banana
Mela
Pera
```

Questa volta il metodo `Arrays.sort` funziona correttamente, perché può utilizzare il metodo `compareTo` per confrontare gli elementi dell'array e riordinarli. Per mostrare in modo ancora più chiaro che il metodo `compareTo` viene chiamato all'interno del metodo `Arrays.sort`, lo si può ridefinire utilizzando un criterio di confronto diverso. Invece di utilizzare l'ordinamento lessicografico, si potrebbe utilizzare come metro di paragone la lunghezza del nome del frutto: i frutti con un nome breve verranno prima di quelli con il nome più lungo. Di seguito è riportata la definizione alternativa del metodo `compareTo`:

```
public int compareTo(Object o) {
    if ((o != null) && (o instanceof Frutto)) {
        Frutto altroFrutto = (Frutto) o;
        if (nomeFrutto.length() > altroFrutto.nomeFrutto.length())
            return 1;
        else if (nomeFrutto.length() < altroFrutto.nomeFrutto.length())
            return -1;
        else
            return 0;
    }
    return -1; // Default nel caso l'oggetto non sia un Frutto
}
```

Con questa definizione il programma del Listato 11.17 produrrà il seguente risultato:

```
Mela
Pera
Banana
Arancia
```

I frutti sono quindi ordinati a partire da quello con il nome più corto.

11.4 Riepilogo

- Con *binding* dinamico, o *late binding*, si intende il fatto che la decisione di quale versione di un metodo è appropriata viene presa a run-time. Java usa il *binding* dinamico.
- Polimorfismo significa utilizzare il *binding* dinamico per fare in modo che gli oggetti possano eseguire azioni differenti con lo stesso nome di metodo.
- È possibile assegnare un oggetto di un tipo derivato a una variabile del tipo base (o antenato), ma non è possibile fare il contrario.
- Un metodo astratto funge da "segnaposto" per un metodo che sarà poi definito in una classe derivata.
- Una classe astratta è una classe che non può essere istanziata; non si possono, cioè, creare oggetti di un tipo classe astratto. Una classe che contiene metodi astratti deve essere dichiarata astratta.
- Una classe astratta serve da classe base per derivare altre classi.
- Una classe astratta è un tipo. È possibile definire variabili e parametri di metodi i cui tipi sono un tipo classe astratto.
- Un'interfaccia Java contiene le intestazioni dei metodi pubblici e le definizioni delle costanti pubbliche. Non dichiara costruttori, variabili di istanza o metodi definiti.
- Una classe che implementa un'interfaccia deve definire un corpo per ogni metodo specificato dell'interfaccia stessa. In caso contrario, deve essere definita astratta. La classe potrebbe definire ulteriori metodi che non sono dichiarati nell'interfaccia. Una classe può implementare più di un'interfaccia.
- Un progettista di classi utilizza le interfacce per specificare i metodi a un programmatore.
- Un'interfaccia è un tipo riferimento: è possibile dichiarare variabili e parametri di metodi che hanno un tipo interfaccia.
- Si può estendere un'interfaccia per creare un'interfaccia che comprende i metodi presenti nell'interfaccia esistente più alcuni nuovi metodi.

11.5 Esercizi

1. Si supponga di voler implementare un programma di disegno che crei varie forme utilizzando i caratteri della tastiera. Si implementi una classe base astratta `FormaDisegnabile` che definisce il centro (due valori interi) e il colore (una stringa) dell'oggetto. Si forniscano appropriati metodi *set* per gli attributi. Si dovrebbe definire, inoltre, un metodo *set* che sposti l'oggetto di una data quantità.
2. Si crei una classe `Quadrato` derivata da `FormaDisegnabile`, come descritta nel precedente esercizio. Un oggetto `Quadrato` ha una variabile di istanza che rappresenta la lunghezza del lato. La classe dovrebbe avere un metodo *get* e un metodo *set*

per la lunghezza. Dovrebbe anche avere i metodi per calcolare l'area e il perimetro del quadrato. Sebbene i caratteri siano più alti che larghi, non occorre preoccuparsi di questo dettaglio quando si disegna il quadrato.

3. Si crei una classe astratta `PoliticaSconto`. Essa dovrebbe avere un solo metodo astratto `calcolaSconto` che restituirà lo sconto per l'acquisto di un certo numero di articoli tutti dello stesso tipo. Il metodo ha due parametri, `numeroArticoli` e `prezzoArticolo`.

4. Si derivi una classe `ScontoQuantita` da `PoliticaSconto`, come descritta nel precedente esercizio. Essa dovrebbe avere un costruttore con due parametri, `minimo` e `percentuale`. Si dovrebbe ridefinire il metodo `calcolaSconto` in modo che se la quantità di un articolo acquistato è maggiore del minimo, lo sconto è di `percentuale` sul totale.

5. Si derivi una classe `CompraNArticoliPrendiUnoGratis` da `PoliticaSconto`, come descritta nell'Esercizio 3. La classe dovrebbe avere un costruttore che ha un singolo parametro `n`. In più, la classe dovrebbe ridefinire il metodo `calcolaSconto` così che ogni `n`-esimo articolo sia gratis. Per esempio, la seguente tabella fornisce lo sconto per l'acquisto di varie quantità di un articolo che costa 10 Euro, quando `n` è 3:

Quantità	1	2	3	4	5	6	7
Sconto	0	0	10	10	10	20	20

6. Si derivi una classe `ScontoCombinato` da `PoliticaSconto`, come descritta nell'Esercizio 3. Questa dovrebbe avere un costruttore con due parametri di tipo `PoliticaSconto`. Si dovrebbe ridefinire il metodo `calcolaSconto` per restituire il valore massimo restituito da `calcolaSconto` per ognuna delle sue politiche di sconto private. Le due politiche di sconto sono descritte negli Esercizi 4 e 5.

7. Si definisca `PoliticaSconto` come un'interfaccia invece che come la classe astratta descritta nell'Esercizio 3.

8. Si crei un'interfaccia `CodificatoreMessaggio` che ha un solo metodo `codifica(testoInChiaro)`, dove `testoInChiaro` sarà il messaggio da codificare. Il metodo restituirà il messaggio codificato.

9. Si crei una classe `CifrarioAScorrimento` che implementa l'interfaccia `CodificatoreMessaggio`, come descritta nel precedente esercizio. Il costruttore dovrebbe avere un parametro intero chiamato `chiave`. Si definisca il metodo `codifica` così che ogni lettera sia spostata del valore contenuto in `chiave`. Per esempio, se `chiave` è uguale a 3, la lettera `a` sarà sostituita da `d`, la lettera `b` sarà sostituita da `e`, la lettera `c` sarà sostituita da `f` e così via. *Suggerimento*: si potrebbe definire un metodo privato che sposta un singolo carattere.

10. Si crei una classe `CifrarioACombinazione` che implementa l'interfaccia `CodificatoreMessaggio`, come descritta nell'Esercizio 8. Il costruttore dovrebbe avere un parametro intero chiamato `n`. Si definisca il metodo `codifica` così che il messaggio sia combinato `n` volte. Per eseguire una singola combinazione, si divide il mes-

saggio a metà e poi si prendono i caratteri da ognuna delle metà in modo alternato. Per esempio, se il messaggio è `abcdefghi`, le metà sono `abcde` e `fghi`. Il messaggio combinato è `afbgchdie`. *Suggerimento*: si potrebbe definire un metodo privato che esegue una combinazione.

11.6 Progetti

1. Si definisca una classe `Rombo` derivata dalla classe `FormaGenerica` (Listato 11.5) o dalla classe astratta `FormaBase` (Listato 11.9). Un rombo ha la stessa rappresentazione per la sua parte superiore di un oggetto di `Triangolo` e la sua parte inferiore è una versione rovesciata della sua parte superiore. Si definiscano i metodi opportuni che disegnino le linee orizzontali, le linee della grande V e le linee della grande V rovesciata.
2. Si definiscano due classi derivate dalla classe astratta `FormaBase` nel Listato 11.9. Le due classi saranno chiamate `FrecciaDestra` e `FrecciaSinistra`. Queste classi saranno come le classi `Rettangolo` e `Triangolo`, ma disegneranno frecce che puntano, rispettivamente, a destra e a sinistra. Per esempio, la seguente freccia punta a destra:

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

   La dimensione della freccia è determinata da due numeri, uno per la lunghezza della coda e uno per la larghezza della punta della freccia. La larghezza è la lunghezza della base verticale. La freccia mostrata ha una lunghezza di 16 e una larghezza di 7. La larghezza della punta della freccia non può essere un numero pari; pertanto i costruttori e i metodi *set* dovrebbero verificare che questa sia sempre dispari. Si scriva un programma di prova per ogni classe che verifichi tutti i metodi nella classe. Si può supporre che la larghezza della base della punta della freccia sia almeno 3.
3. Si creino le classi `TriangoloRettangolo` e `Rettangolo`, ognuna delle quali sia derivata dalla classe astratta `FormaBase` del Listato 11.9. Si derivi poi una classe `Quadrato` dalla classe `Rettangolo`. Ognuna di queste tre classi derivate avrà due metodi aggiuntivi per calcolare l'area e il perimetro, oltre ai metodi ereditati. Non si dimentichi di ridefinire il metodo `disegnaQui`. Si dia alle classi un ragionevole insieme di costruttori e di metodi *get*. La classe `Quadrato` dovrebbe includere solo una dimensione (il lato) e dovrebbe automaticamente impostare l'altezza e la larghezza alla lunghezza del lato. Si possono usare le dimensioni in termini di larghezza del carattere e di interlinea anche se essi sono senza dubbio diversi, così un quadrato non sembrerà un quadrato (proprio come un oggetto di `Rettangolo`, come discusso in questo capitolo, non soddisferà le aspettative). Si scriva un programma *driver* che verifichi tutti i metodi.

4. Si crei un'interfaccia `DecodificatoreMessaggio` che abbia un solo metodo `decodifica(testoCodificato)`, dove `testoCodificato` sarà il messaggio da decodificare. Il metodo restituirà il messaggio decodificato. Si modifichino le classi `CifrarioAScorrimento` e `CifrarioACombinazione`, come descritte negli Esercizi 9 e 10, in modo che implementino `DecodificatoreMessaggio` oltre all'interfaccia `CodificatoreMessaggio` descritta nell'Esercizio 8. Infine, si scriva un programma che permetta a un utente di codificare e decodificare i messaggi inseriti da tastiera.

5. Nel Progetto 8 del Capitolo 10 è stato chiesto di ridefinire la classe `Alieno` in modo che sfruttasse l'ereditarietà. La nuova classe dovrebbe ora essere resa astratta, poiché non esiste alcuna esigenza di istanziare alieni, ma solo specifici tipi di alieni. Si renda astratto anche il metodo `getDanno`. Si verifichi la classe con il metodo `main` già definito.

6. Si definisca una classe astratta `Film` che rappresenta un film preso a noleggio da una videoteca. Nella classe `Film` si deve definire un codice identificativo e un titolo. Si definiscano per questi attributi i metodi *get* e *set*. Si definisca anche un metodo `equals` che sovrascrive quello ereditato da `Object` e che restituisce `true` se due film hanno il loro codice identificativo uguale. Si creino, inoltre, tre classi derivate dalla classe `Film` chiamate `Azione`, `Commedia` e `Dramma`. In ultimo, si crei un metodo ridefinito chiamato `calcolaPenaleRitardo` che prende in ingresso il numero di giorni di ritardo per un film e restituisce la penale per quel film. La penale predefinita è di Euro 2 al giorno. I film di azione hanno una penale pari a Euro 3 al giorno, le commedie Euro 2.50 al giorno e i film drammatici Euro 2 al giorno. Si verifichino le classi in un metodo `main`.

7. Si estenda il progetto precedente realizzando una classe `Noleggio`. Questa classe dovrebbe memorizzare il `Film` che è stato noleggiato, un numero intero che rappresenta il documento d'identificazione del cliente che ha affittato il film e un numero intero che rappresenta il numero di giorni di ritardo del film. Si aggiunga un metodo che calcola le penali per il noleggio. Si crei un'altra classe in cui si definisce il metodo `main`. Nel metodo `main` si crei un array di tipo base `Noleggio` e lo si riempia con i dati per tutti i tipi di film. Si crei, quindi, un metodo `calcolaPenaliRitardo` che itera attraverso l'array e restituisce l'ammontare totale di penali che devono essere incassate.

8. Modificare la classe `Studente` del Listato 10.2 in modo che implementi l'interfaccia `Comparable`. Si definisca il metodo `compareTo` per oggetti di tipo `Studente` basandosi sul numero di matricola. In un metodo `main`, si crei un array di almeno cinque studenti, lo si ordini utilizzando il metodo `Arrays.sort` e se ne stampino gli elementi, che dovrebbero comparire in ordine di numero di matricola crescente. Successivamente, si modifichi il metodo `compareTo` in modo che ordini gli studenti secondo l'ordinamento lessicografico dei rispettivi nomi. Senza che siano necessarie modifiche al metodo `main`, il programma dovrebbe ora elencare gli studenti ordinati per nome.

9. L'obiettivo di questo progetto è quello di creare una semplice simulazione di un sistema predatore-preda in due dimensioni. Questi animali vivono in un mondo costituito da una griglia di celle 20 x 20. Ad ogni istante, ogni cella può essere occu-

Capitolo 12

ArrayList e generici

OBIETTIVI

- Definire e utilizzare un'istanza di `ArrayList`.
- Definire e utilizzare classi che hanno tipi generici.

Come si è detto nel Capitolo 8, un tipo di dato astratto o ADT (*Abstract Data Type*), specifica un insieme di dati e le operazioni consentite su di esso. Pertanto descrive le operazioni da effettuare, ma non come implementarle o come memorizzare i dati. Fondamentalmente, un ADT specifica solo una particolare organizzazione dei dati. Le specifiche di un ADT possono essere espresse attraverso un'interfaccia (*interface*) Java, come descritto nel Capitolo 11. Una classe, come noto, può implementare un'interfaccia in diversi modi. Questo significa che un ADT può essere implementato definendo una classe Java. Così facendo, si utilizzano differenti strutture dati. Una **struttura dati** è un costrutto, per esempio una classe o un array, all'interno di un linguaggio di programmazione.

Una struttura dati le cui dimensioni aumentano o diminuiscono durante l'esecuzione del programma viene chiamata **dinamica**. Un tipo di struttura dinamica è basata sugli array. A titolo esemplificativo, si presenta la classe `ArrayList` definita nella Java Class Library.

A partire dalla versione 5.0, Java consente di attribuire alle definizioni di classe dei parametri per i tipi di dato utilizzati. Questi parametri sono noti come **tipi generici** (*generic types*). Nel Paragrafo 12.1 si mostra come utilizzare una di queste definizioni di classe (`ArrayList`) presente nella Java Class Library. Il Paragrafo 12.2 insegna a scrivere definizioni di classi che contengono tipi di dato generici.

Prerequisiti

Il Paragrafo 12.1 deve essere letto prima del Paragrafo 12.2. I Capitoli da 1 a 6 e i Capitoli 8 e 9 sono necessari al fine di comprendere a pieno questo capitolo. Un po' di familiarità con le basi dell'ereditarietà sarà, comunque, utile per la comprensione degli argomenti trattati in questo capitolo. I dettagli sono i riportati nel successivo prospetto.

Paragrafo	Prerequisiti
12.1 Strutture di dati basate su array	Capitoli da 1 a 6, Capitoli 8 e 9
12.2 Generici	Paragrafo 12.1

12.1 Strutture di dati basate su array

In Java, la lunghezza di un array può essere letta anche durante l'esecuzione del programma, ma una volta che il programma crea un array di una certa lunghezza, questa non può essere cambiata. Per esempio, si supponga di scrivere un programma che registra gli ordini dei clienti per un'azienda di vendita per corrispondenza e di memorizzare tutti gli articoli ordinati da un cliente in un array `ordine` di oggetti di una classe chiamata `ArticoloOrdine`. Si potrebbe chiedere all'utente il numero di articoli che compongono l'ordine, memorizzare tale numero in una variabile chiamata `numeroDiArticoli` e poi creare l'array utilizzando la seguente istruzione:

```
ArticoloOrdine[] ordine = new ArticoloOrdine[numeroDiArticoli];
```

Ma cosa accade se il cliente inserisce `numeroDiArticoli` ma poi decide di ordinare un ulteriore articolo? Non esiste alcuna possibilità di incrementare le dimensioni dell'array `ordine`. Tuttavia, si può simulare tale incremento: basta creare un nuovo array più grande, copiare gli elementi dall'array originale al nuovo array e poi rinominare il nuovo array come `ordine`. Per esempio, le seguenti istruzioni raddoppiano efficacemente le dimensioni dell'array:

```
ArticoloOrdine[] nuovoArray = new ArticoloOrdine[2 * numeroDiArticoli];
for (int indice = 0; indice < numeroDiArticoli; indice++)
   nuovoArray[indice] = ordine[indice];
ordine = nuovoArray;
```

12.1.1 La classe `ArrayList`

Al posto di cambiare le dimensioni dell'array `ordine`, si può utilizzare un'istanza della classe `ArrayList`, che si trova nel package `java.util` della Java Class Library. Tale istanza può offrire gli stessi servizi offerti da un array, tranne per il fatto che è in grado di cambiare la propria lunghezza durante l'esecuzione del programma. Un oggetto `ArrayList` potrebbe pertanto gestire senza alcun problema l'aumento di un articolo nell'ordine.

Ma se basta utilizzare l'`ArrayList` per superare il limite principale nell'utilizzo degli array, perché è necessario studiare gli array? Perché non utilizzare sempre `ArrayList`? È evidente che ogni medaglia ha un rovescio. L'`ArrayList` presenta due grandi svantaggi.

- Un'istanza di `ArrayList` è meno efficiente di un array.
- Un'istanza di `ArrayList` può memorizzare solo oggetti; non può contenere valori di un tipo primitivo, come `int`, `double` o `char`.

L'implementazione di `ArrayList` si basa comunque su array. Di fatto, per estendere la capacità del suo array, `ArrayList` utilizza la tecnica utilizzata precedentemente per estendere l'array `ordine`. In un programma, l'utilizzo di `ArrayList` al posto di un array richiederà un tempo di computazione maggiore, che in alcuni casi potrebbe avere un impatto sulla velocità del programma. Di conseguenza, bisognerebbe analizzare caso per caso prima di prendere una decisione in merito. Il secondo svantaggio può essere risolto come segue: invece di memorizzare valori di tipo `int`, si potrebbe memorizzare valori di tipo `Integer`, dove `Integer` è una classe *wrapper* i cui oggetti simulano valori di tipo `int`. Il *boxing* e l'*unboxing* automatico (discussi nel Capitolo 9) rendono conveniente l'utilizzo di una classe *wrapper*. Tuttavia l'utilizzo di tale classe aggiunge al programma un ulteriore *overhead* (letteralmente "sovraccarico") computazionale.

Si noti che `ArrayList` è un'implementazione di un ADT chiamata **lista** (*list*). L'ADT lista organizza i dati nello stesso modo con cui si scrive una lista nella vita di tutti i giorni: liste di compiti, di indirizzi, di regali e della spesa. In ogni caso, in un oggetto di tipo `ArrayList` si possono aggiungere voci alla lista (all'inizio, alla fine o tra elementi) o eliminare, leggere e contare le voci.

12.1.2 Creare un'istanza di `ArrayList`

Utilizzare un'istanza di `ArrayList` è come utilizzare un array, con alcune importanti differenze. In primo luogo, la definizione della classe `ArrayList` non viene fornita automaticamente. La definizione è nel package `java.util` e qualsiasi codice che utilizza la classe `ArrayList` deve contenere la seguente istruzione all'inizio del file:

```
import java.util.ArrayList;
```

Si crea e si nomina un'istanza di `ArrayList` nello stesso modo con cui si crea e si nomina un oggetto di una qualsiasi classe, ad eccezione del fatto che occorre specificare il tipo base. Per esempio:

```
ArrayList<String> lista = new ArrayList<String>(20);
```

Questa istruzione imposta `lista` come il nome di un oggetto che memorizza istanze della classe `String`. Il tipo `String` è il tipo base. Un oggetto della classe `ArrayList` memorizza gli oggetti del suo tipo base, esattamente come un array memorizza gli elementi del suo tipo base. La differenza è che un tipo base di `ArrayList` deve essere una classe; non si può utilizzare come tipo base un tipo primitivo, come `int` o `double`.

L'oggetto `lista` ha una **capacità iniziale** di 20 elementi. Quando si dice che un oggetto `ArrayList` ha una capacità iniziale, si intende che è stata allocata memoria sufficiente per questo numero di elementi. Se si ha la necessità di ospitare più elementi, il sistema allocherà automaticamente più memoria. Scegliendo attentamente la capacità iniziale, si può migliorare l'efficienza del codice. Se si sceglie una capacità iniziale abbastanza grande, il sistema non avrà bisogno di riallocare la memoria troppo spesso, e, come risultato, il programma sarà più veloce. D'altro canto, se si imposta una capacità iniziale eccessiva, si andrà incontro a uno spreco di memoria. In ogni caso, qualsiasi sia la capacità scelta, ciò non ha nessun effetto sul numero di elementi che possono essere inseriti in un oggetto `ArrayList`. Se poi si omette la capacità iniziale, verrà invocato il costruttore di default di `ArrayList`, il quale adotta una capacità pari a 10.

Creare e nominare un'istanza di `ArrayList`

Si crea e si nomina un oggetto della classe `ArrayList` allo stesso modo di qualsiasi altro oggetto, ad eccezione del fatto che deve essere specificato un tipo base.

Sintassi

```
ArrayList<tipo_base> variabile = new ArrayList<tipo_base>();
ArrayList<tipo_base> variabile = new ArrayList<tipo_base>(capacità);
```

Il *tipo_base* deve essere una classe; non può essere un tipo primitivo come `int` o `double`. Quando al costruttore viene passato come argomento *capacità*, essa determina la capacità iniziale della lista. Omettere tale argomento imposta una capacità iniziale pari a 10.

Esempi

```
ArrayList<String> listaA = new ArrayList<String>();
ArrayList<Double> listaB = new ArrayList<Double>(30);
```

12.1.3 Utilizzare i metodi di `ArrayList`

Un oggetto della classe `ArrayList` può essere utilizzato come un array, ma occorre utilizzare i suoi metodi al posto della notazione a parentesi quadre, tipica degli array. Di seguito vengono definiti un array e un oggetto `ArrayList` e viene assegnata loro la stessa capacità:

```
String[] unArray = new String[20];
ArrayList<String> unaLista = new ArrayList<String>(20);
```

Gli oggetti di `ArrayList` sono indicizzati allo stesso modo di un array: l'indice del primo elemento è 0. Così, se si utilizzasse:

```
unArray[indice] = "Ciao Mamma!";
```

per l'array `unArray`, l'istruzione analoga per l'oggetto `unaLista` sarebbe:

```
unaLista.set(indice, "Ciao Mamma!");
```

Se si volesse utilizzare:

```
String temp = unArray[indice];
```

per estrarre un elemento dall'array `unArray`, l'analoga istruzione per `unaLista` sarebbe:

```
String temp = unaLista.get(indice);
```

I due metodi `set` e `get` forniscono agli oggetti `ArrayList` approssimativamente le stesse funzionalità che le parentesi quadre forniscono agli array.

Comunque, bisogna essere consapevoli di un punto importante: l'invocazione del metodo:

```
unaLista.set(indice, "Ciao Mamma!");
```

non è sempre completamente analoga a:

```
unArray[indice] = "Ciao Mamma!";
```

Il metodo `set` può sostituire un qualsiasi elemento *esistente*, ma non può essere utilizzato per inserire un elemento in una posizione che non contiene ancora un elemento, come invece è possibile fare con un array. Il metodo `set` viene utilizzato per cambiare il valore degli elementi esistenti, non per impostarli la prima volta.

Per aggiungere un elemento per la prima volta, si utilizza il metodo `add`. Questo metodo aggiunge incrementalmente gli elementi alle posizioni indicizzate 0, 1, 2 e così via. Un oggetto `ArrayList` deve sempre essere riempito esattamente in questo ordine crescente. Il metodo `add` ha due definizioni. Quando si fornisce un argomento, `add` aggiunge tale elemento immediatamente dopo l'ultima posizione occupata in quel momento. Fornendo, invece, due argomenti, il metodo aggiunge l'elemento alla posizione indicata, sempre che la posizione precedente sia occupata. Per esempio, se `unaLista` contiene cinque elementi, sia:

```
unaLista.add("Cambusa");
```

sia:

```
unaLista.add(5, "Cambusa");
```

aggiungono `"Cambusa"` come sesto e ultimo elemento.

Al contrario, per una lista di cinque elementi, l'istruzione:

```
unaLista.add(6, "Cambusa");
```

scatena un errore di indice fuori dai limiti (*index out of bounds*), che causa un messaggio di errore. Infatti, poiché la lista contiene cinque elementi, l'indice dell'ultimo elemento è 4. Il tentativo di aggiungere un elemento con indice 6 prima di inserirne un altro con indice 5 è illegale.

Dopo aver aggiunto elementi a `unaLista`, si può anche utilizzare `add` per inserire un elemento prima o tra due elementi esistenti. L'istruzione:

```
unaLista.add(0, "Motore");
```

aggiunge (inserisce) una stringa prima di tutti gli altri elementi di `unaLista`. Gli elementi esistenti vengono spostati per fare spazio al nuovo elemento. Così, la stringa che originariamente si trovava alla posizione con indice 0 non viene sostituita, ma spostata alla posizione con indice 1. Allo stesso modo:

```
unaLista.add(4, "Vagone Merci");
```

inserisce l'elemento `"Vagone Merci"` all'indice 4. Gli elementi agli indici precedenti rimangono al proprio posto, mentre quelli successivi vengono spostati di una posizione. Quando si utilizza il metodo `add` per inserire un nuovo elemento in una posizione indicizzata, tutti gli elementi che si trovavano da quella posizione in poi vedranno il proprio indice incrementato di una unità, in modo da creare spazio per inserire il nuovo elemento senza perdere nessun elemento preesistente. Diversamente dalla procedura di inserimento in un array, questo processo avviene automaticamente e non è necessario scrivere alcun codice aggiuntivo per lo spostamento degli elementi. Chiaramente, utilizzare `ArrayList` è molto più semplice che utilizzare un array.

Per ottenere il numero di elementi presenti in `unaLista` si utilizza il metodo `size`. L'espressione `unaLista.size()` restituisce quindi il numero di elementi memorizzati in `unaLista`. Gli indici di questi elementi sono compresi tra 0 e `unaLista.size() - 1`. La Figura 12.1 descrive una selezione dei metodi della classe `ArrayList`.

`public ArrayList<tipo_base>()`
Si comporta come il costruttore precedente, ma la capacità iniziale è pari a 10.

`public boolean add(tipo_base nuovoElemento)`
Inserisce `nuovoElemento` alla fine di questa lista e incrementa la dimensione della lista di 1 unità. Se necessario incrementa la capacità della lista. Restituisce vero se l'inserimento avviene con successo.

`public void add(int indice, tipo_base nuovoElemento)`
Inserisce `nuovoElemento` nella posizione `indice` di questa lista. Per fare spazio al nuovo elemento, sposta gli elementi successivi incrementando il loro indice di 1 unità. La dimensione della lista viene incrementata di 1. Se necessario incrementa la capacità della lista. Se `indice < 0` o se `indice ≥ size()` scatena un errore di indice fuori dai limiti (*index out of bounds*).

`public tipo_base get(int indice)`
Restituisce l'elemento alla posizione `indice` di questa lista. Se `indice < 0` o se `indice ≥ size()` scatena un errore di indice fuori dai limiti (*index out of bounds*).

`public tipo_base set(int indice, tipo_base elemento)`
Sostituisce l'elemento alla posizione `indice` di questa lista con `elemento`. Restituisce l'elemento sostituito. Se `indice < 0` o se `indice ≥ size()` scatena un errore di indice fuori dai limiti (*index out of bounds*).

`public tipo_base remove(int indice)`
Rimuove e restituisce l'elemento alla posizione `indice` di questa lista. Sposta gli elementi che sono nelle posizioni successive decrementando il loro indice di 1 unità. Decrementa la dimensione della lista di 1 unità. Se `indice < 0` o se `indice ≥ size()` scatena un errore di indice fuori dai limiti (*index out of bounds*).

`public boolean remove(Object elemento)`
Rimuove la prima occorrenza di `elemento` in questa lista e sposta gli elementi successivi decrementando il loro indice di 1 unità. Decrementa la dimensione della lista di 1 unità. Restituisce vero se `elemento` è stato rimosso; altrimenti restituisce falso e non altera la lista.

`public void clear()`
Rimuove tutti gli elementi da questa lista.

`public int size()`
Restituisce il numero di elementi di questa lista.

`public boolean contains(Object elemento)`
Restituisce vero se `elemento` è in questa lista; altrimenti restituisce falso.

`public int indexOf(Object elemento)`
Restituisce l'indice della prima occorrenza di `elemento` in questa lista. Restituisce -1 se l'elemento non è nella lista.

`public boolean isEmpty()`
Restituisce vero se questa lista è vuota; altrimenti restituisce falso.

Figura 12.1 Alcuni metodi della classe `ArrayList`.

FAQ Perché alcuni parametri sono di tipo base e altri di tipo `Object`?

Si guardi la tabella dei metodi nella Figura 12.1. In alcuni casi, quando un parametro è ovviamente un oggetto del tipo base, il tipo di parametro è il tipo base, ma in altri casi è di tipo `Object`. Per esempio, il metodo `add` ha un parametro del tipo base, ma il metodo `contains` ha un parametro di tipo `Object`. Perché c'è differenza nei tipi di parametro? Dal punto di vista dell'`ArrayList`, ha senso solo aggiungere un elemento del tipo base, così `add` non gestisce oggetti di altro tipo. Tuttavia, `contains` è più generico. Poiché il suo tipo di parametro è `Object`, può verificare se un oggetto di qualunque tipo è nella lista. Un oggetto il cui tipo non è il tipo base non può assolutamente essere nella lista, così se passato a `contains`, il metodo restituisce `false`.

Array o oggetti della classe `ArrayList`

Negli array, le parentesi quadre e la variabile di istanza `length` sono gli unici strumenti nelle mani di un programmatore. Se occorre utilizzare l'array per altre operazioni, bisogna scrivere tutto il codice necessario. `ArrayList`, al contrario, offre un'ampia selezione di metodi, che possono svolgere automaticamente molte delle operazioni che un programmatore sarebbe costretto a scrivere per utilizzare un array. Per esempio, la classe `ArrayList` ha un metodo, `add`, che inserisce un nuovo elemento tra due elementi esistenti.

ESEMPIO DI PROGRAMMAZIONE
UNA LISTA DELLE COSE DA FARE

Il Listato 12.1 contiene un esempio di come utilizzare l'`ArrayList` per memorizzare una lista di attività quotidiane. L'utente può inserire tante attività quante ne desidera e, successivamente, il programma mostrerà la lista.

LISTATO 12.1 Utilizzare `ArrayList` per memorizzare una lista.

```
import java.util.ArrayList;
import java.util.Scanner;

public class ArrayListDemo {

    public static void main(String[] args) {
        ArrayList<String> listaDelleCoseDaFare = new ArrayList<String>();

        System.out.println("Inserisci gli elementi per" +
                           " la lista quando richiesto.");
        boolean fatto = false;
        Scanner tastiera = new Scanner(System.in);
```

```
        while (!fatto) {
            System.out.println("Inserisci un elemento:");
            String elemento = tastiera.nextLine();
            listaDelleCoseDaFare.add(elemento);
            System.out.print("Altri elementi per la lista? ");
            String risposta = tastiera.nextLine();

            if (!risposta.equalsIgnoreCase("si"))
                fatto = true;
        }

        System.out.println("La lista contiene:");
        int dimensioneLista = listaDelleCoseDaFare.size();

        for (int posizione = 0; posizione < dimensioneLista; posizione++)
            System.out.println(listaDelleCoseDaFare.get(posizione));
    }
}
```

Esempio di output

```
Inserisci gli elementi per la lista quando richiesto.
Inserisci un elemento:
Comprare il latte
Altri elementi per la lista? si
Inserisci un elemento:
Lavare l'auto
Altri elementi per la lista? si
Inserisci un elemento:
Fare il compito
Altri elementi per la lista? no
La lista contiene:
Comprare il latte
Lavare l'auto
Fare il compito
```

Usare un ciclo `for-each` per accedere a tutti gli elementi presenti in un'istanza di `ArrayList`

L'ultimo ciclo del Listato 12.1, che mostra tutti gli elementi in un'istanza di `ArrayList`, può essere sostituito dal seguente ciclo `for-each`:

```
for (String elemento : listaDelleCoseDaFare)
    System.out.println(elemento);
```

Questo ciclo, presentato nel Paragrafo 6.1.7, è molto più semplice da scrivere rispetto a quello originale quando occorre accedere a tutti gli elementi di una collezione, per esempio di un oggetto `ArrayList`.

Usare `trimToSize` per risparmiare memoria

Gli oggetti `ArrayList` raddoppiano automaticamente la propria capacità quando il programma richiede loro di crescere. Tuttavia, la nuova capacità potrebbe essere molto maggiore di quella effettivamente richiesta dal programma. In tali casi, la capacità non si restringe automaticamente. Se la capacità della lista espansa è maggiore del necessario, si può risparmiare memoria utilizzando il metodo `trimToSize`. Per esempio, se `unaLista` è un'istanza di `ArrayList`, l'istruzione `unaLista.trimToSize()` restringerà la capacità di `unaLista` fino alle sue dimensioni reali, lasciandola senza capacità inutilizzata. Normalmente, si dovrebbe usare `trimToSize` solo quando si è certi che, a breve, non sarà necessario un incremento di capacità.

Usare un'istruzione di assegnamento per copiare una lista

Così come non è possibile utilizzare un'istruzione di assegnamento per copiare un'istanza di una classe o un array, non si può copiare un'istanza di `ArrayList` utilizzando un'istruzione di assegnamento. Per esempio, si consideri il seguente codice:

```
ArrayList<String> unaLista = new ArrayList<String>();
<codice per riempire unaLista>
ArrayList<String> altroNome = unaLista;     //Definisce un alias
```

Questo codice semplicemente rende `altroNome` un sinonimo di `unaLista`: due nomi diversi, ma una sola lista. Per creare una copia identica di `unaLista`, in modo da avere due copie distinte, si usa il metodo `clone`. Di conseguenza, al posto della precedente istruzione di assegnamento si scriverebbe:

```
ArrayList<String> listaDuplicata = (ArrayList<String>)unaLista.clone();
```

Per liste di oggetti diversi dalle stringhe, il metodo `clone` risulta essere più complicato e può condurre ad alcune difficoltà.

12.1.4 Classi parametriche e tipi di dato generico

La classe `ArrayList` è una **classe parametrica** (*parameterized class*). Ciò vuol dire che ha un parametro, che è stato definito nei paragrafi precedenti *tipo_base*, che può essere sostituito con un qualunque tipo classe per ottenere una classe che memorizza oggetti di tipo *tipo_base*. *tipo_base* è anche detto **tipo di dato generico** (*generic data type*). Dal momento che è stato illustrato l'utilizzo della classe `ArrayList`, si è già in grado di utilizzare classi parametriche. Il Paragrafo 12.2 spiegherà, invece, come definire tali classi.

Collezioni

Java ha un gruppo di classi, chiamato **Java Collections Framework**, che implementano l'interfaccia `Collection`. Anche la classe `ArrayList` fa parte di questo insieme.

12.2 Generici

Come si è detto precedentemente, a partire dalla versione 5.0, Java permette di scrivere definizioni di classe che includono parametri per i tipi di dato. Questi parametri sono chiamati **generici** (*generics*). Questo paragrafo offre una breve introduzione a questo argomento. Programmare con i generici può essere insidioso e richiede attenzione. Per programmare seriamente con i generici si dovrebbe consultare un testo più avanzato.

12.2.1 Fondamenti

Le classi e i metodi possono utilizzare un **tipo parametrico** (*type parameter*) al posto di uno specifico tipo di dato. Quando un programmatore usa una tale classe o un tale metodo, specifica il tipo classe per il tipo parametrico, in modo da produrre un tipo classe o un metodo specifico per le sue esigenze. Per esempio, il Listato 12.2 mostra una definizione di classe molto semplice che utilizza un tipo parametrico `T`. Si noti la presenza delle parentesi angolari intorno a `T` nell'intestazione della classe. Si può utilizzare un qualsiasi identificativo per il tipo parametrico tranne le parole chiave del linguaggio: non è necessario utilizzare esattamente `T`. Comunque, per convenzione, i tipi parametrici iniziano con una lettera maiuscola e ormai si è sviluppata una sorta di convenzione nell'uso di una singola lettera. Iniziare con una lettera maiuscola ha senso, poiché solo un tipo classe può essere collegato al tipo parametrico. La convenzione di usare una singola lettera non è obbligatoria.

Quando si scrive una classe o un metodo che utilizza un tipo parametrico, si può utilizzare tale parametro in (quasi) qualsiasi punto in cui si può utilizzare un tipo classe. Per esempio, la classe `Esempio` del Listato 12.2 utilizza `T` come tipo di dato per la variabile di istanza `dati`, per il parametro `nuovoValore` del metodo `setDati` e per il tipo restituito dal metodo `getDati`. Non si può, invece, utilizzare un tipo parametrico per creare un nuovo oggetto. Così anche se `dati` è di tipo `T` nella classe `Esempio`, non si può scrivere

```
dati = new T();      //Illegale
```

MyLab

LISTATO 12.2 Una definizione di classe che utilizza un tipo parametrico.

```
public class Esempio<T> {
    private T dati;

    public void setDati(T nuovoValore) {
        dati = nuovoValore;
    }

    public T getDati() {
        return dati;
    }
}
```

Inoltre, non si può utilizzare un tipo parametrico quando si alloca memoria per un array. Sebbene si possa dichiarare un array scrivendo un'istruzione come questa

```
T[] unArray;                    //Dichiarazione valida di un array
```

non si può scrivere

```
unArray = new T[20];            //Illegale
```

Non è possibile utilizzare un tipo parametrico in tutti i punti dove è possibile usare un nome di tipo

All'interno della definizione di una classe parametrica, ci sono punti in cui si può utilizzare un nome di tipo, ma non un tipo parametrico. Non si può utilizzare un tipo parametrico in espressioni semplici che usano new per creare un nuovo oggetto o per allocare memoria per un array. Per esempio, le seguenti espressioni sono illegali nella definizione di una classe parametrica il cui tipo parametrico è `T`:

```
T oggetto = new T();            //Il primo T è legale
                                //il secondo è illegale.
T[] a = new T[10];              //Il primo T è legale
                                //il secondo è illegale.
```

Questa restrizione non è arbitraria come potrebbe sembrare a prima vista. Nel primo caso, `T` non è utilizzato come tipo parametrico, ma come nome di un costruttore. Nel secondo caso, `T` è usato in modo simile a un costruttore, anche se non lo è a tutti gli effetti.

Una definizione di classe che utilizza un tipo parametrico viene memorizzata in un file e compilata come qualsiasi altra classe. Per esempio, la classe parametrica mostrata nel Listato 12.2 dovrebbe essere memorizzata nel file `Esempio.java`.

Una volta compilata, la classe parametrica può essere utilizzata come qualsiasi altra classe, tranne per il fatto che occorre specificare il tipo classe che sostituisce il tipo parametrico. Per esempio, la classe `Esempio` tratta dal Listato 12.2 potrebbe essere usata come segue:

```
Esempio<String> esempio1 = new Esempio<String>();
esempio1.setDati("Ciao");
Esempio<Specie> esempio2 = new Esempio<Specie>();
Specie creatura = new Specie();
... <Codice per impostare i dati per l'oggetto creatura>
esempio2.setDati(creatura);
```

Si noti la presenza delle parentesi angolari che contengono il tipo classe attuale che sostituisce il tipo parametrico. La classe `Specie` potrebbe essere quella definita nel Capitolo 8, ma i dettagli non sono importanti: potrebbe trattarsi di una qualsiasi classe definita dal programmatore.

A questo punto dovrebbe essere chiaro che la classe `ArrayList`, presentata all'inizio del capitolo, è una classe parametrica. È stata creata un'istanza di `ArrayList` scrivendo, per esempio:

```
ArrayList<String> lista = new ArrayList<String>(20);
```

In questo caso, `String` sostituisce il tipo parametrico nella definizione della classe.

Inferenza di tipo in Java 7

A partire dalla versione 7, Java offre una funzionalità denominata **inferenza di tipo**. Si tratta della capacità di inferire il tipo da sostituire a un tipo parametrico in una chiamata a un costruttore a partire dal tipo utilizzato nella dichiarazione della variabile. La seguente istruzione

```
NomeClasse<TipoBase> nomeOggetto = new NomeClasse<TipoBase>();
```

può quindi essere scritta, in Java 7, come

```
NomeClasse<TipoBase> nomeOggetto = new NomeClasse<>();
```

Il nuovo formato rende un po' più corte le istruzioni ed è anche un po' più chiaro da leggere. Tuttavia, dato che il formato precedente è già stato utilizzato dai programmatori per diversi anni, è probabile che lo si incontri nel codice esistente. Per una maggiore compatibilità, gli esempi di questo libro non utilizzano il nuovo formato.

Esempi

```
ArrayList<String> lista = new ArrayList<>();
ArrayList<Double> lista2 = new ArrayList<>(30);
```


Definizioni di classi che hanno un tipo parametrico

Si possono definire classi che usano un parametro al posto di un tipo classe. Si specifica un tipo parametrico all'interno delle parentesi angolari proprio dopo il nome della classe nell'intestazione della classe stessa. Si può utilizzare come identificatore del tipo parametrico qualsiasi parola (che non sia una parola chiave del linguaggio), ma, per convenzione, il tipo parametrico inizia con una lettera maiuscola.
Si utilizza il tipo parametrico all'interno della definizione di classe nello stesso modo in cui si usa un tipo classe, a parte il fatto che non si può utilizzare in congiunzione con `new`.

Esempio

Si veda il Listato 12.2.

Usare una classe la cui definizione ha un tipo parametrico

Si può creare un oggetto di una classe parametrica nello stesso modo in cui si crea un oggetto di qualsiasi altra classe, tranne per il fatto che si deve specificare un tipo classe attuale (al posto del tipo parametrico) all'interno delle parentesi angolari che seguono il nome della classe.

Esempio

```
Esempio<String> unOggetto = new Esempio<String>();
unOggetto.setDati("Ciao");
```

Compilare con l'opzione `-Xlint`

L'utilizzo dei tipi parametrici espone a molte insidie. Se si compila con l'opzione `-Xlint`, si riceveranno molte informazioni diagnostiche su qualsiasi sorta di problema reale o potenziale. Per esempio, la classe `Esempio` nel Listato 12.2 dovrebbe essere compilata come segue:

```
javac -Xlint Esempio.java
```

Se, per compilare un programma, si utilizza un ambiente di sviluppo integrato (IDE), è opportuno controllare la documentazione per vedere come impostare l'opzione di compilazione.

Quando si compila con l'opzione `-Xlint` si avranno molti più *warning* (letteralmente "avvertimento") di quanti se ne otterrebbero altrimenti. Un *warning* non è un errore e se il compilatore produce solo *warning* e nessun messaggio di errore, la classe è stata compilata e può essere utilizzata. Tuttavia, in molti casi occorre assicurarsi di comprendere i *warning* e che questi non indichino un problema reale. In tal caso, occorre modificare il codice per eliminare i *warning*.

ESEMPIO DI PROGRAMMAZIONE
UNA CLASSE GENERICA PER COPPIE ORDINATE

Nel Listato 12.3 è presentata una classe parametrica per la rappresentazione di coppie ordinate di valori. Si noti che la dichiarazione del costruttore non include il tipo parametrico `T`. Ciò è poco intuitivo, ma corretto. Si può utilizzare un tipo parametrico, come `T`, come tipo per un parametro di un costruttore, ma la dichiarazione del costruttore non deve comprendere il tipo parametrico tra parentesi angolari come `<T>`.

Utilizzando questa classe parametrica con il tipo `String` sostituito al tipo parametrizzato `T`, come mostrato più avanti, si ottiene una classe le cui istanze rappresentano coppie di oggetti `String`:

```
Coppia<String> coppiaSegreta =
    new Coppia<String>("Buona", "Giornata");
```

Sostituendo il tipo `T` con il tipo `Integer`, si ottiene invece una classe le cui istanze sono coppie di oggetti `Integer`:

```
Coppia<Integer> lanciDado =
    new Coppia<Integer>(new Integer(2), new Integer(3));
```

Se `Animale` è una qualche classe già definita, si può usare `Animale` al posto di `T`, come mostrato di seguito, per ottenere una classe i cui oggetti sono coppie di oggetti di tipo `Animale`:

```
Animale maschio = new Animale();
Animale femmina = new Animale();
<Codice per impostare le proprietà di maschio e femmina>
Coppia<Animale> coppiaDaAllevamento =
    new Coppia<Animale>(maschio, femmina);
```

Il Listato 12.4 contiene un semplice esempio dell'uso della classe generica `Coppia`.

MyLab

LISTATO 12.3 Una classe generica per coppie ordinate.

```
public class Coppia<T> {
  private T primo;
  private T secondo;

  public Coppia() {
      primo = null;
      secondo = null;
  }

  public Coppia(T primoElemento, T secondoElemento) {
      primo = primoElemento;
      secondo = secondoElemento;
  }

  public void setPrimo(T nuovoPrimo) {
      primo = nuovoPrimo;
  }

  public void setSecondo(T nuovoSecondo) {
      secondo = nuovoSecondo;
  }

  public T getPrimo() {
      return primo;
  }
```

Le dichiarazioni dei costruttori non includono il tipo parametrico tra parentesi angolari.

```
    public T getSecondo() {
        return secondo;
    }

    public String toString() {
        return ("primo: " + primo.toString() + "\n"
                           + "secondo: " + secondo.toString());
    }

    public boolean equals(Object altroOggetto) {
        if (altroOggetto == null)
            return false;
        else if (getClass() != altroOggetto.getClass())
            return false;
        else {
            Coppia<T> altraCoppia = (Coppia<T>)altroOggetto;
            return (primo.equals(altraCoppia.primo)
                           && secondo.equals(altraCoppia.secondo));
        }
    }
}
```

LISTATO 12.4 Uso della classe Coppia.

```
import java.util.Scanner;

public class CoppiaGenericaDemo {
    public static void main(String args[]) {
        Coppia<String> coppiaSegreta =
            new Coppia<String>("Buona", "Giornata");

        Scanner tastiera = new Scanner(System.in);
        System.out.println("Inserire due parole:");
        String parola1 = tastiera.next();
        String parola2 = tastiera.next();
        Coppia<String> coppiaDiInput =
            new Coppia<String>(parola1, parola2);

        if (coppiaDiInput.equals(coppiaSegreta)) {
            System.out.println("Hai indovinato le parole segrete");
            System.out.println("nell'ordine giusto!");
        } else {
            System.out.println("Hai sbagliato.");
            System.out.println("Hai provato con");
            System.out.println(coppiaDiInput);
            System.out.println("Le parole segrete sono");
            System.out.println(coppiaSegreta);
        }
    }
}
```

MyLab

Video 12.1 Definire classi che usano i generici

Esempio di output

```
Inserire due parole:
due parole
Hai sbagliato.
Hai provato con
primo: due
secondo: parole
Le parole segrete sono
primo: Buona
secondo: Giornata
```

Una definizione di classe può avere più di un tipo parametrico

La definizione di una classe generica può avere un qualunque numero di tipi parametrici. I tipi parametrici sono elencati tra parentesi angolari come nel caso di un singolo tipo parametrico, separati da virgole. Per esempio, nel Listato 12.5 la classe `Coppia` è stata riscritta in modo che il primo e il secondo elemento della coppia possano essere di tipi diversi. Nel Listato 12.6 è presentato un semplice esempio dell'uso di questa classe generica con due tipi parametrici.

MyLab

LISTATO 12.5 *Tipi parametrici multipli.*

```
public class CoppiaADueTipi<T1, T2> {
    private T1 primo;
    private T2 secondo;

    public CoppiaADueTipi() {
        primo = null;
        secondo = null;
    }

    public CoppiaADueTipi(T1 primoElemento, T2 secondoElemento) {
        primo = primoElemento;
        secondo = secondoElemento;
    }

    public void setPrimo(T1 nuovoPrimo) {
        primo = nuovoPrimo;
    }

    public void setSecondo(T2 nuovoSecondo) {
        secondo = nuovoSecondo;
    }

    public T1 getPrimo() {
        return primo;
    }
```

```
    public T2 getSecondo() {
        return secondo;
    }

    public String toString() {
        return ("primo: " + primo.toString() + "\n"
                          + "secondo: " + secondo.toString());
    }

    public boolean equals(Object altroOggetto) {
        if (altroOggetto == null)
            return false;
        else if (getClass() != altroOggetto.getClass())
            return false;
        else {
            CoppiaADueTipi<T1, T2> altraCoppia =
                        (CoppiaADueTipi<T1, T2>)altroOggetto;
            return (primo.equals(altraCoppia.primo)
                && secondo.equals(altraCoppia.secondo));
        }
    }
}
```

Il primo equals è quello del tipo T1, il secondo quello del tipo T2.

LISTATO 12.6 Uso di una classe generica con due tipi parametrici.

```
import java.util.Scanner;

public class CoppiaADueTipiDemo {
    public static void main(String[] args) {
        CoppiaADueTipi<String, Integer> giudizio =
            new CoppiaADueTipi<String, Integer>("The Car Guys", 8);
        Scanner tastiera = new Scanner(System.in);
        System.out.println(
            "Il nostro voto attuale per " + giudizio.getPrimo());
        System.out.println(" e' " + giudizio.getSecondo());
        System.out.println("Tu che voto daresti?");
        int punteggio = tastiera.nextInt();
        giudizio.setSecondo(punteggio);
        System.out.println(
            "Il nostro nuovo voto per " + giudizio.getPrimo());
        System.out.println(" e' " + giudizio.getSecondo());
    }
}
```

Esempio di output

```
Il nostro voto attuale per The Car Guys
e' 8
Tu che voto daresti?
10
Il nostro nuovo voto per The Car Guys
e' 10
```

Un nome di costruttore in una classe generica non ha tipo parametrico

Il nome della classe nella definizione di una classe parametrica ha specificato un tipo parametrico, come, per esempio, nel caso di `Coppia<T>` nel Listato 12.3. Questo può indurre a pensare che sia necessario usare il tipo parametrico nell'intestazione della definizione di costruttore, ma non è così. Per esempio, si utilizza

```
public Coppia()
```

al posto di

```
public Coppia<T>()      //Illegale
```

Un tipo parametrico, come `T`, può essere utilizzato come parametro per un costruttore, come nell'esempio seguente, ma l'intestazione del costruttore non include il tipo parametrico tra parentesi angolari:

```
public Coppia(T primoElemento, T secondoElemento)
```

Per l'esempio completo si veda il Listato 12.3.

A prima vista, questo comportamento può provocare confusione. Come è stato appena sottolineato, nella definizione di un costruttore di una classe parametrica non compare mai il tipo parametrico tra parentesi angolari. Pertanto, nel Listato 12.3 compare l'intestazione

```
public Coppia(T primoElemento, T secondoElemento)
```

Tuttavia, come mostrato nel Listato 12.4, quando si istanzia una classe parametrica specificando un tipo da sostituire a quello parametrico, tale tipo deve essere indicato tra parentesi angolari dopo il nome del costruttore, come nel seguente esempio tratto dal Listato 12.4:

```
Coppia<String> coppiaSegreta = new Coppia<String>("Buona", "Giornata");
```

Questo secondo aspetto non è però difficile da ricordare. Se non si specificasse il tipo `<String>`, Java non saprebbe quale versione della classe `Coppia` si vuole utilizzare: il compilatore non potrebbe stabilire se si tratta di `Coppia<String>`, `Coppia<Double>` o qualunque altra versione della classe `Coppia`.

Non si può sostituire un tipo primitivo a un tipo parametrico

Quando si crea un oggetto di una classe che ha un tipo parametrico, non si può sostituire il tipo parametrico con un tipo primitivo, come `int`, `double` o `char`. Per esempio, la seguente istruzione causerà un errore di compilazione:

```
ArrayList<int> unaLista = new ArrayList<int>(20);     //Illegale!
```

Tuttavia, ora che Java offre la funzionalità di conversione automatica, questa non è una grossa restrizione in pratica. Per esempio, non si può avere un oggetto di tipo `Coppia<int>`, ma se ne può avere uno di tipo `Coppia<Integer>`, che grazie alla conversione automatica (*boxing*) può essere utilizzato con valori di tipo `int`. Un esempio è riportato nel Listato 12.7.

In generale, il tipo sostituito al tipo parametrico deve essere un riferimento. Il caso più tipico è quello nel quale si utilizza un tipo classe, ma si può usare qualunque tipo di riferimento. In particolare, si può sostituire ad un tipo parametrico un tipo array.

LISTATO 12.7 Uso della classe `Coppia` con la conversione automatica.

```
import java.util.Scanner;

public class CoppiaGenericaDemo2 {
      public static void main(String[] args) {
          Coppia<Integer> coppiaSegreta = new Coppia<Integer>(42, 24);

          Scanner tastiera = new Scanner(System.in);
          System.out.println("Inserire due numeri:");
          int n1 = tastiera.nextInt();
          int n2 = tastiera.nextInt();

          Coppia<Integer> coppiaUtente = new Coppia<Integer>(n1, n2);

          if (coppiaUtente .equals(coppiaSegreta)) {
              System.out.println("Hai indovinato i numeri segreti");
              System.out.println("nell'ordine corretto!");
          } else {
              System.out.println("Hai sbagliato.");
              System.out.println("Hai provato");
              System.out.println(coppiaUtente);
              System.out.println("I numeri segreti sono");
              System.out.println(coppiaSegreta);
          }
      }
}
```

La conversione automatica consente di utilizzare un valore di tipo `int` al posto di un parametro di tipo `Integer`.

Esempio di output

```
Inserire due numeri:
42 24
Hai indovinato i numeri segreti
nell'ordine corretto!
```

Un'istanza di una classe parametrica non può essere il tipo base di un array

Le definizioni di array come la seguente non sono ammesse:

```
Coppia<String>[] a = new Coppia<String>[10];     //Illegale!
```

Questa sembrerebbe una cosa ragionevole da voler fare, ma non è consentita a causa dei dettagli tecnici relativi a come Java implementa le classi parametriche. La spiegazione completa di questa restrizione andrebbe oltre lo scopo di questo libro.

12.2.2 Vincoli sui tipi parametrici

A volte non ha senso sostituire un tipo di riferimento qualunque a un tipo parametrico nella definizione di una classe parametrica. Per esempio, si consideri la classe parametrica `Coppia` definita nel Listato 12.3. Si supponga di voler aggiungere un metodo che restituisca il massimo tra i due valori nella coppia ordinata. Si potrebbe aggiungere alla classe `Coppia` del Listato 12.3 la definizione del seguente metodo:

```
public T massimo() {
    if (primo.compareTo(secondo) <= 0)
        return primo;
    else
        return secondo;
}
```

Come illustrato nel Capitolo 11, il metodo `compareTo` è richiesto in ogni classe che implementi l'interfaccia `Comparable`. Questa è un'interfaccia della libreria standard di Java e richiede la presenza del solo metodo seguente:

```
public int compareTo(Object altro)
```

È utile ricordare qui che quando si definisce una classe che implementa l'interfaccia `Comparable`, è necessario definire il metodo `compareTo` in modo che restituisca:

- un numero negativo se l'oggetto sul quale è stato chiamato il metodo "viene prima" del parametro `altro`;
- zero se l'oggetto sul quale è stato chiamato il metodo "è uguale" al parametro `altro`;
- un numero positivo se l'oggetto sul quale è chiamato il metodo "viene dopo" il parametro `altro`.

C'è però un problema. Tutto questo funziona solo se il tipo sostituito al parametro `T` rispetta l'interfaccia `Comparable`, ma Java consente di inserire qualunque tipo al posto del tipo parametrico `T`.

In Java è possibile esprimere restrizioni sui possibili tipi sostituibili a un tipo parametrico. Per garantire che si possano sostituire a `T` solo classi che implementino l'interfaccia `Comparable`, si inizia la definizione della classe come segue:

```
public class Coppia<T extends Comparable>
```

La parte `extends Comparable` è chiamata vincolo sul tipo parametrico `T`. Se si cerca di sostituire a `T` un tipo che non implementa l'interfaccia `Comparable`, si otterrà un errore in fase di compilazione. Si noti che è necessario utilizzare la parola chiave `extends` e non `implements` come ci si potrebbe aspettare.

Il vincolo `extends Comparable` non è una comodità opzionale. Se lo si omette, si causerà un errore del compilatore perché il metodo `compareTo` è ignoto.

La nuova versione della classe generica `Coppia` con il metodo massimo è riportata nel Listato 12.8.

LISTATO 12.8 Un tipo parametrico vincolato.

```
public class Coppia<T extends Comparable> {
    private T primo;
    private T secondo;

    public T massimo() {
        if (primo.compareTo(secondo) <= 0)
            return primo;
        else
            return secondo;
    }

    <Tutti i costruttori e metodi presentati nel Listato 12.3
    sono inclusi anche nella definizione di questa classe generica>

}
```

Un vincolo su un tipo può essere anche espresso in termini di una classe (anziché di un'interfaccia), nel qual caso solo le classi che discendono da quella specificata possono essere sostituite al tipo parametrico. Per esempio, l'istruzione seguente specifica che solo le classi che discendono dalla classe `Impiegato` possono essere sostituite al tipo `T`:

```
public class UnaClasseGenerica<T extends Impiegato>
```

Ciò spiega perché nei vincoli si utilizza la parola chiave `extends` al posto di `implements`.

In un vincolo si possono specificare più interfacce ed eventualmente una classe. È sufficiente separare i vari elementi con un *ampersend* `&`, come nell'esempio seguente:

```
public class AltraClasseGenerica<T extends Impiegato & Comparable>
```

Se sono presenti più tipi parametrici, la sintassi è quella dell'esempio che segue:

```
public class AltraClasseGenerica<T1 extends Impiegato & Comparable,
    T2 extends Comparable>
```

In un vincolo si può elencare un numero qualunque di interfacce, ma al più una classe. Inoltre, se compaiono sia una classe che una o più interfacce, la classe va specificata per prima.

Vincoli ai tipi parametrici

È possibile imporre che la classe da sostituire a un tipo parametrico discenda da una classe specificata, implementi una o più interfacce specificate, o entrambe le cose.

Sintassi (per l'intestazione della definizione di una classe)

```
public class NomeClasse <Tipo extends ClasseBase & Interfaccia1
    & Interfaccia2 & ... & UltimaInterfaccia>
```

Se ci sono più tipi parametrici, vanno separati da virgole. Per ogni tipo parametrico può esserci un numero qualunque di interfacce, ma solo una classe base.

Esempi

```
public class Coppia<T extends Comparable>
public class MiaClasse<T extends Impiegato & Comparable>
public class TuaClasse<T1 extends Impiegato & Comparable & Cloneable,
    T2 extends Comparable>
```

Nell'esempio, `Impiegato` è una classe, mentre `Comparable` e `Cloneable` sono interfacce.

Interfacce generiche

Anche un'interfaccia può avere uno o più tipi parametrici. I dettagli e la notazione da utilizzare sono gli stessi visti per le classi con tipi parametrici.

12.2.3 Metodi generici

Quando si definisce una classe generica, si possono utilizzare i tipi parametrici nelle definizioni dei suoi metodi. Si possono anche definire metodi generici che abbiano i propri tipi parametrici, diversi da quelli della classe. Un metodo generico può essere un membro di una classe ordinaria (cioè non generica) o di una classe generica rispetto a qualche altro tipi parametrico. Per esempio, si consideri la seguente classe:

```
public class MetodiUtili {
    ...

    public static <T> T getPuntoMedio(T[] a) {
        return a[a.length / 2];
    }

    public static <T> T getPrimo(T[] a) {
        return a[0];
    }
```

In questo caso, la classe `MetodiUtili` non ha tipi parametrici, ma i metodi `getPuntoMedio` e `getPrimo` hanno ognuno un tipo parametrico. Si noti che il tipo parametrico tra parentesi angolari, `<T>`, è posto dopo tutti i modificatori (in questo caso, `public static`) e prima del tipo di ritorno.

Quando si invoca uno di questi metodi generici, è necessario anteporre al nome del metodo, tra parentesi angolari, il tipo da sostituire, come negli esempi seguenti:

```
String puntoMedio = MetodiUtili.<String>getPuntoMedio(b);
double primoNumero = MetodiUtili.<Double>getPrimo(c);
```

Si noti che il punto è prima della specifica del tipo tra parentesi angolari: il tipo fa parte del nome del metodo, non di quello della classe. Si noti anche che i metodi `getPuntoMedio` e `getPrimo` utilizzano tipi diversi in sostituzione dei rispettivi tipi parametrici. Il tipo parametrico è locale al metodo, non alla classe (l'argomento `b` è un array di `String`, mentre `c` è un array di `Double`).

È anche possibile definire metodi generici all'interno di classi generiche, come nell'esempio che segue:

```
public class Esempio<T> {
    private T dati;

    public Esempio(T nuoviDati) {
        dati = nuoviDati;
    }

    public <TipoVisualizzatore> void mostraA(TipoVisualizzatore
                                                visualizzatore) {
        System.out.println("Ciao " + visualizzatore);
        System.out.println("I dati sono " + dati);
    }
    ...
}
```

Si noti che `T` e `TipoVisualizzatore` sono tipi parametrici diversi. `T` è un tipo parametrico per l'intera classe, mentre `TipoVisualizzatore` è un tipo parametrico solo per il metodo `mostraA`. Di seguito è riportato un esempio di utilizzo di questi metodi generici:

```
Esempio<Integer> oggetto = new Esempio<Integer>(42);
oggetto.<String>mostraA("Amico");
```

Il risultato prodotto è

```
Ciao Amico
I dati sono 42
```

12.2.4 Ereditarietà con classi generiche

Una classe generica può estendere una classe ordinaria o un'altra classe generica. Il Listato 12.9 contiene la definizione di una classe generica denominata `CoppiaNonOrdinata`, che estende la classe generica `Coppia` (presentata nel Listato 12.3). La classe `CoppiaNonOrdinata` sovrascrive la definizione del metodo `equals` ereditato da `Coppia`. Per chi la utilizza, la classe `CoppiaNonOrdinata` è simile alla classe `Coppia`, con un'eccezione: in `CoppiaNonOrdinata` le due componenti non devono essere necessariamente nello

stesso ordine affinché due coppie siano considerate uguali. In termini meno formali, utilizzando `Coppia<String>`, la coppia "birra" e "noccioline" è diversa dalla coppia "noccioline" e "birra". Usando `CoppiaNonOrdinata<String>` le due coppie di stringhe sono uguali. Ciò è illustrato nel Listato 12.10.

MyLab

LISTATO 12.9 Una classe generica derivata.

```
public class CoppiaNonOrdinata<T> extends Coppia<T> {
    public CoppiaNonOrdinata () {
        setPrimo(null);
        setSecondo(null);
    }

    public CoppiaNonOrdinata(T primoElemento, T secondoElemento) {
        setPrimo(primoElemento);
        setSecondo(secondoElemento);
    }

    public boolean equals(Object altroOggetto) {
        if (altroOggetto == null)
            return false ;
        else if (getClass() != altroOggetto.getClass())
            return false ;
        else {
            CoppiaNonOrdinata<T> altraCoppia =
                    (CoppiaNonOrdinata <T>)altroOggetto;
            return (getPrimo().equals(altraCoppia.getPrimo())
                && getSecondo().equals(altraCoppia.getSecondo()))
                ||
                (getPrimo ().equals(altraCoppia.getSecondo()) &&
                getSecondo().equals(altraCoppia.getPrimo()));
        }
    }
}
```

MyLab

LISTATO 12.10 Uso della classe `CoppiaNonOrdinata`.

```
public class CoppiaNonOrdinataDemo {
    public static void main(String[] args) {
        CoppiaNonOrdinata<String> p1 =
            new CoppiaNonOrdinata<String>("noccioline", "birra");
        CoppiaNonOrdinata<String> p2 =
            new CoppiaNonOrdinata<String>("birra", "noccioline");

        if (p1.equals(p2)) {
            System.out.println(p1.getPrimo() + " e " +
                p1.getSecondo() + " è lo stesso di");
```

```
            System.out.println(p2.getPrimo() + " e "
                + p2.getSecondo());
        }
    }
}
```

Esempio di output

```
noccioline e birra è lo stesso di
birra e noccioline
```

Esattamente come ci si aspetta, un oggetto di tipo `CoppiaNonOrdinata<String>` è anche di tipo `Coppia<String>`. Come si è visto finora, l'ereditarietà con le classi generiche è semplice nella maggior parte dei casi. Tuttavia, ci sono situazioni che presentano delle insidie difficili da individuare. Ne verrà ora illustrato un esempio.

Si supponga di avere una classe `ImpiegatoAdOre`, derivata dalla classe `Impiegato`. Si potrebbe pensare che un oggetto di tipo `Coppia<ImpiegatoAdOre>` sia anche di tipo `Coppia<Impiegato>`. Ciò tuttavia non è corretto. Se `G` è una classe generica, non c'è alcuna relazione tra `G<A>` e `G<B>`, indipendentemente da qualunque relazione esista eventualmente tra `A` e `B`.

12.3 Riepilogo

- `ArrayList` è una classe definita nella Java Class Library. Le istanze di `ArrayList` possono essere considerate come array che possono aumentare di dimensione. Si possono creare e nominare tali istanze nello stesso modo utilizzato con ogni altro oggetto, tranne per il fatto che occorre specificare il loro tipo di base.
- `ArrayList` ha metodi potenti, che possono fare molte più cose rispetto a un comune array.
- Si può definire una classe che utilizza un parametro invece di un tipo classe.
- Si deve specificare il tipo parametrico fra parentesi angolari subito dopo il nome della classe, nell'intestazione della classe stessa.
- Il codice che crea un oggetto di una classe parametrica sostituisce il tipo parametrico con un tipo classe attuale, mantenendo le parentesi angolari.
- La classe standard `ArrayList` è una classe parametrica.

12.4 Esercizi

1. Si ripeta l'Esercizio 2 del Capitolo 6, ma usando un'istanza di `ArrayList` invece di un array. Non si legga il numero dei valori, ma si continui a leggere i valori finché l'utente non inserisce un valore negativo
2. Si ripeta l'Esercizio 13 del Capitolo 9, ma usando un'istanza di `ArrayList` invece di un array. Non si legga il numero di famiglie, ma si leggano i dati per le famiglie finché l'utente non inserisce la parola `fatto`.
3. Si ripeta l'Esercizio 4 del Capitolo 6, ma usando un'istanza di `ArrayList` invece di un array.

4. Si ripetano gli Esercizi 14 e 15 del Capitolo 9, ma usando un'istanza di `ArrayList` invece di un array. Non ci sarà più bisogno di conoscere il numero massimo di vendite, così i metodi dovranno cambiare per far fronte a ciò.
5. Si scriva un metodo statico `rimuoviDuplicati(ArrayList<Character> dati)` che rimuova ogni carattere duplicato nell'oggetto `dati`. Si tenga sempre la prima copia del carattere rimuovendo solo le successive.
6. Si scriva un metodo statico:

   ```
   getStringheComuni(ArrayList<String> lista1, ArrayList<String> lista2)
   ```

   che restituisca una nuova istanza di `ArrayList` contenente tutte le stringhe comuni a `lista1` e a `lista2`.
7. Si ripeta l'Esercizio 16 del Capitolo 9, ma usando un'istanza di `ArrayList` invece di un array. Si facciano i seguenti piccoli cambiamenti ai metodi per consentire a un oggetto `ArrayList` di aumentare la sua dimensione.
   - Si cambi il parametro del costruttore dal grado massimo al grado desiderato.
   - Il metodo `setCostante` potrebbe aver bisogno di aggiungere coefficienti pari a zero prima di a_i. Per esempio, se $a_0 = 3$, $a_1 = 5$, $a_2 = 0$, $a_3 = 2$, $a_4 = 0$ e $a_5 = 0$, il polinomio dovrebbe essere di grado 3, poiché l'ultima costante diversa da zero è a_3. L'invocazione di `setCostante(8, 15)` dovrebbe aver bisogno di impostare a_6 e a_7 a 0 e a_8 a 15.

12.5 Progetti

1. Si riveda il metodo `selectionSort` definito nella classe `OrdinaArray` come mostrato nel Listato 6.8, così da ordinare le stringhe di un'istanza della classe `ArrayList<String>` in ordine lessicografico invece di ordinare gli interi in un array in senso crescente. Per le parole, l'ordine lessicografico riconduce all'ordine alfabetico se tutte le parole sono in lettere minuscole o maiuscole. Si possono comparare due stringhe per vedere qual è la prima in senso lessicografico utilizzando il metodo `compareTo` della classe `String` come descritto nella Figura 2.5 del Capitolo 2.
2. Si ripeta il precedente progetto, ma scrivendo un metodo `bubbleSort` che realizzi un *bubble sort*, come descritto nel Progetto 4 del Capitolo 6.
3. Si crei una nuova versione del Progetto 1, ma scrivendo un metodo `insertionSort` che realizzi un *insertion sort*, come descritto nel Progetto 5 del Capitolo 6.
4. Si scriva un programma che crei un oggetto `Animale` dai dati letti da tastiera. Si memorizzino questi oggetti in un'istanza di `ArrayList`. Si sistemino quindi gli oggetti `Animale` in ordine alfabetico rispetto al nome dell'animale e infine si mostrino sullo schermo i dati degli oggetti `Animale` ordinati. La classe `Animale` è fornita nel Capitolo 9, Listato 9.1.
5. Si ripeta il precedente progetto, ma ordinando gli oggetti `Animale` rispetto al peso dell'animale invece che rispetto al nome. Dopo aver visualizzato i dati sullo schermo, si scriva il numero e la percentuale degli animali il cui peso è, rispettivamente, sotto i 5 kg, compreso tra i 5 e i 10 kg e oltre i 10 kg.

Capitolo 13

Eccezioni

OBIETTIVI

- Descrivere la gestione delle eccezioni.
- Reagire correttamente al presentarsi di certe eccezioni.
- Utilizzare in modo efficace le strutture di gestione delle eccezioni di Java all'interno di classi e programmi.

Un modo per scrivere programmi consiste nel presupporre che durante l'esecuzione di un programma non si presentino situazioni anomale (o eccezionali). Per esempio, se il programma deve recuperare un elemento da una lista, si presuppone che la lista non sia vuota. Una volta che il programma funziona per il caso normale (cioè quando le cose vanno come previsto), si aggiunge il codice che gestisce i casi eccezionali. Java fornisce gli strumenti necessari per supportare questa pratica operativa. In breve, si scrive il codice nell'ipotesi che non accada nulla di anomalo e, solo successivamente, si aggiunge il codice necessario per gestire le situazioni eccezionali, utilizzando costrutti appositi. Lo scopo di questo capitolo è di introdurre queste strutture.

Prerequisiti

Il Paragrafo 13.1 richiede la lettura dei Capitoli da 1 a 5 e dei Capitoli 8 e 9. La parte rimanente del capitolo richiede, in più, parte del materiale riguardante l'ereditarietà presentato nel Capitolo 10. Sono inoltre menzionati gli array, la cui comprensione richiede la lettura del Capitolo 6.

13.1 Concetti di base sulla gestione delle eccezioni

Java fornisce gli strumenti necessari per gestire alcuni tipi di anomalie che si possono verificare durante l'esecuzione dei programmi. Questi strumenti permettono di dividere un programma o un metodo in sezioni distinte: quella che gestisce il normale funzionamento e quella che gestisce il caso eccezionale. In questo modo è possibile scomporre l'attività programmativa in due sotto-attività più piccole e quindi più semplici da trattare.

Un'**eccezione** (*exception*) è un oggetto che segnala l'accadere di un evento anomalo (o eccezionale) durante l'esecuzione di un programma. Il processo di creazione di questo oggetto (cioè di generazione di un'eccezione) è chiamato **lancio** (o sollevamento) di **un'eccezione** (*throwing an exception*). In un'altra parte del programma (magari in un'altra classe o in un altro metodo) si inserisce il codice che si occupa dell'evento eccezionale. Il codice che rileva e che si occupa dell'eccezione si dice che **gestisce l'eccezione** (*handle the exception*).

L'utilizzo delle eccezioni risulta particolarmente utile nel caso in cui la situazione anomala che viene a presentarsi durante l'esecuzione di un metodo necessiti di un trattamento diverso a seconda del programma che usa quel metodo. Come si vedrà, tale metodo, in caso di anomalia, sarà in grado di lanciare un'eccezione. Questo permette di gestire in maniera appropriata l'anomalia al di fuori del metodo. Per esempio, se un metodo si trova nella condizione di dover effettuare una divisione per zero, in alcuni casi il programma deve terminare, mentre in altri deve proseguire con altre istruzioni.

13.1.1 Eccezioni in Java

Per introdurre l'argomento, si utilizzerà un semplice programma giocattolo. Il programma verrà inizialmente proposto in una versione che utilizza l'istruzione di selezione e, successivamente, verrà riproposto in una nuova forma che sfrutta le strutture per la gestione delle eccezioni.

Il programma consiste nello stabilire quante ciambelle si possono mangiare in relazione al numero di bicchieri di latte disponibili. Nel programma si presuppone che il latte sia un elemento talmente importante della nostra società da far pensare che le persone non ne rimarrebbero mai sprovviste. Nonostante ciò, si desidera comunque che il programma sia in grado di gestire la remota possibilità che una persona possa rimanere senza latte. Se si presuppone che una persona non possa mai rimanere sprovvista di latte, il codice di base del programma potrebbe essere il seguente:

```
System.out.println("Inserire il numero di ciambelle:");
int conteggioCiambelle = tastiera.nextInt();
System.out.println("Inserire il numero di bicchieri di latte:");
int conteggioLatte = tastiera.nextInt();
double ciambellePerBicchiere =
      conteggioCiambelle / (double)conteggioLatte;
System.out.println(conteggioCiambelle + " ciambelle.");
System.out.println(conteggioLatte + " bicchieri di latte.");
System.out.println("Hai " + ciambellePerBicchiere +
                   " ciambelle per ogni bicchiere di latte.");
```

Se il numero immesso per i bicchieri di latte è 0 (ovvero, non vi è più latte) questo codice si trova a dover compiere una divisione per zero. Per prevenire questo tipo di operazione illegale, è possibile aggiungere un'istruzione che accerti il verificarsi di questa situazione anomala. Il programma completo è presentato nel Listato 13.1.

Di seguito, si rivedrà il programma utilizzando le strutture per la gestione delle eccezioni di Java. Una divisione per zero produce un'eccezione. Di conseguenza, invece di evitare l'esecuzione di tale divisione, la si effettua e si reagisce opportunamente all'eccezione risultante, come si può vedere nel Listato 13.2.

Il programma riveduto è certamente più complesso rispetto a quello originale nel Listato 13.1. Ciononostante, il codice tra le parole `try` e `catch` è più pulito rispetto a quello utilizzato nel programma originale e suggerisce, quindi, che l'utilizzo della gestione delle eccezioni sia vantaggioso.

LISTATO 13.1 Un modo di gestire una situazione anomala.

MyLab

```
import java.util.Scanner;

public class PrendiLatte {

    public static void main(String[] args) {

        Scanner tastiera = new Scanner(System.in);

        System.out.println("Inserire il numero di ciambelle:");
        int conteggioCiambelle = tastiera.nextInt();

        System.out.println("Inserire il numero di bicchieri di latte:");
        int conteggioLatte = tastiera.nextInt();

        //Gestione degli eventi eccezionali senza utilizzare le strutture
        //di gestione delle eccezioni di Java
        if (conteggioLatte < 1) {
            System.out.println("Niente latte!");
            System.out.println("Vai a comprare del latte.");

        } else {
            double ciambellePerBicchiere =
                    conteggioCiambelle / (double)conteggioLatte;
            System.out.println(conteggioCiambelle + " ciambelle.");
            System.out.println(conteggioLatte + " bicchieri di latte.");
            System.out.println("Hai " + ciambellePerBicchiere +
                               " ciambelle per ogni bicchiere di latte.");

        }
        System.out.println("Fine programma.");
    }
}
```

Esempio di output

```
Inserire il numero di ciambelle:
2
Inserire il numero di bicchieri di latte:
0
Niente latte!
Vai a comprare del latte.
Fine programma.
```

MyLab

LISTATO 13.2 Un esempio di gestione delle eccezioni.

```
import java.util.Scanner;

public class PrendiLatteConEccezioni {

    public static void main(String[] args) {

        Scanner tastiera = new Scanner(System.in);

        try {
            System.out.println("Inserire il numero di ciambelle:");
            int conteggioCiambelle = tastiera.nextInt();

            System.out.println("Inserire il numero di bicchieri " +
                               "di latte:");
            int conteggioLatte = tastiera.nextInt();

            if (conteggioLatte < 1)
                throw new Exception("Eccezione: Niente latte!");

            double ciambellePerBicchiere =
                   conteggioCiambelle / (double)conteggioLatte;
            System.out.println(conteggioCiambelle + " ciambelle.");
            System.out.println(conteggioLatte + " bicchieri di latte.");
            System.out.println("Hai " + ciambellePerBicchiere +
                               " ciambelle per ogni bicchiere di latte.");
        } catch(Exception e) {
            System.out.println(e.getMessage());
            System.out.println("Vai a comprare del latte.");
        }

        System.out.println("Fine programma.");
    }
}
```

Blocco try

Blocco catch

Questo programma è solo un semplice esempio della sintassi di base per la gestione delle eccezioni.

Esempio di output 1

```
Inserire il numero di ciambelle:
3
Inserire il numero di bicchieri di latte:
2
3 ciambelle.
2 bicchieri di latte.
Hai 1.5 ciambelle per ogni bicchiere di latte.
Fine programma.
```

Esempio di output 2

```
Inserire il numero di ciambelle:
2
Inserire il numero di bicchieri di latte:
0
Eccezione: Niente latte!
Vai a comprare del latte.
Fine programma.
```

Il codice nel Listato 13.2 è praticamente uguale a quello presentato nel Listato 13.1, a eccezione del fatto che l'istruzione `if-else` è stata sostituita dalla seguente istruzione `if`:

```
if (conteggioLatte < 1)
    throw new Exception("Eccezione: Niente latte!");
```

L'istruzione `if` richiede che, qualora non vi sia latte, il programma compia qualcosa di eccezionale, specificato dopo la parola `catch`. L'idea è che le situazioni normali siano gestite dal codice che segue la parola `try`, mentre il codice che segue la parola `catch` venga utilizzato solamente in circostanze eccezionali.

La gestione di base delle eccezioni in Java avviene mediante l'utilizzo del gruppo di tre istruzioni `try-throw-catch`. Un **blocco try** (*try block*) ha la seguente sintassi:

```
try {
    codice_da_provare
}
```

Un blocco `try` contiene il codice che viene eseguito nelle situazioni normali. Viene chiamato blocco `try` (letteralmente "prova") poiché non vi è la sicurezza assoluta che ogni cosa vada come ci si aspetta, ma si vuole ugualmente fare un tentativo.

Nel caso qualcosa andasse storto, si vuole generare (o lanciare, dalla parola inglese *throw*) un'eccezione, che è un modo per indicare che sono sorti dei problemi. Aggiungendo un'**istruzione throw** (*throw statement*), la sintassi precedente diventa:

```
try {
    codice_da_provare
    eventuale_generazione_di_eccezione
    altro_codice
}
```

Il blocco `try` contenuto nel Listato 13.2 si presenta nella forma sopra descritta, in quanto contiene la seguente istruzione `throw`:

```
throw new Exception("Eccezione: Niente latte!");
```

Se viene eseguita, l'istruzione `throw` crea un nuovo oggetto della classe predefinita `Exception` mediante l'espressione:

```
new Exception("Eccezione: Niente latte!");
```

e **lancia** (*throws*) l'oggetto creato. La stringa `"Eccezione: Niente latte!"` costituisce l'argomento per il costruttore della classe `Exception`. L'oggetto di tipo `Exception` così creato, memorizza questa stringa in una sua variabile di istanza, in modo che possa essere successivamente recuperata.

Quando viene lanciata un'eccezione, l'esecuzione del codice all'interno del blocco `try` viene arrestata e viene eseguita un'altra porzione di codice, chiamata **blocco catch** (*catch block*). Eseguire il blocco `catch` è detto **catturare l'eccezione** (*catching the exception*). Quando viene lanciata un'eccezione, questa dovrebbe essere catturata da un qualche blocco `catch`. Nel Listato 13.2, il blocco `catch` segue immediatamente il blocco `try`.

Il blocco `catch` somiglia alla definizione di un metodo dotato di un parametro. Nonostante il blocco `catch` non sia un metodo, si comporta, per certi versi, come se lo fosse. È una porzione di codice distinta che viene eseguita se il programma esegue un'istruzione `throw` all'interno del blocco `try` precedente. Questa istruzione `throw` è simile a un'invocazione di metodo; ma, invece di invocare un metodo, chiama il blocco `catch`, provocando l'esecuzione del codice definito nel blocco stesso.

Si esamini ora la prima riga del blocco `catch` presentato nel Listato 13.2:

```
catch(Exception e)
```

L'identificativo `e` somiglia a un parametro e si comporta in modo molto simile a un parametro. Per questo motivo, nonostante il blocco `catch` non sia un metodo, l'identificativo `e` viene chiamato **parametro del blocco catch** (*catch-block parameter*). Il parametro del blocco `catch` fornisce un nome all'eccezione catturata. Ciò consente di inserire all'interno del blocco `catch` delle istruzioni in grado di manipolare l'oggetto eccezione. Il nome più comune per un parametro del blocco `catch` è `e`, ma è possibile utilizzare un qualsiasi identificativo valido. Quando viene lanciato un oggetto eccezione, questo viene assegnato all'identificativo `e` del blocco `catch` e poi viene eseguito il codice presente nel blocco `catch`. In questo caso, è possibile considerare `e` come il nome dell'oggetto eccezione lanciato. Ogni oggetto eccezione ha un metodo chiamato `getMessage`, e, a meno che non lo si ridefinisca (cioè non si effettui un *overriding*), questo metodo restituisce la stringa fornita al costruttore come argomento in fase di creazione dell'oggetto eccezione (cioè quando l'eccezione è stata lanciata). Nell'esempio presentato, l'invocazione al metodo `e.getMessage()` restituisce `"Eccezione: Niente latte!"`. Di conseguenza, quando viene eseguito il blocco `catch` del Listato 13.2, esso produce il seguente messaggio:

```
Eccezione: Niente latte!
Vai a comprare del latte.
```

Il nome di classe che precede il parametro del blocco `catch` specifica quale tipo di eccezione può essere catturata dal blocco `catch`. La classe `Exception` nell'esempio presentato indica che questo blocco `catch` può catturare un'eccezione di tipo `Exception`. Quindi, nell'esempio proposto nel Listato 13.2, un oggetto eccezione che viene lanciato deve essere di tipo `Exception` affinché il blocco `catch` possa venire eseguito. Come si vedrà in seguito, sono possibili altri tipi di eccezioni e un blocco `try` può essere seguito anche da più blocchi `catch`, uno per ogni tipo di eccezione da gestire. Anche se sono presenti più blocchi `catch`, può essere eseguito solamente uno di questi: quello corrispondente al tipo di eccezione generata. Gli altri vengono ignorati durante la gestione di quella particolare eccezione. Poiché tutte le eccezioni sono di tipo `Exception` (sono, cioè, o di tipo `Exception` o sue discendenti), il blocco `catch` dell'esempio presentato è in grado di catturare qualsiasi tipo di eccezione. Anche se questa può sembrare una buona idea, in generale non lo è. È preferibile prevedere più blocchi `catch` specifici piuttosto che uno generale.

Nel programma proposto nel Listato 13.2, quando l'utente inserisce un valore positivo per il numero di bicchieri di latte, non viene lanciata alcuna eccezione. Nel Listato 13.3 viene presentato il flusso di controllo relativo a questo caso. Nel Listato 13.4 è invece

presentato il flusso di controllo quando viene lanciata un'eccezione in risposta all'inserimento di un valore minore o uguale a zero per il numero di bicchieri di latte.

Ricapitolando, un blocco `try` contiene un frammento di codice che può lanciare un'eccezione. Il blocco può lanciare un'eccezione in quanto:

- presenta al proprio interno un'istruzione `throw` o
- invoca un altro metodo che contiene un'istruzione `throw`, come si vedrà più avanti.

LISTATO 13.3 Flusso di controllo quando non viene generata alcuna eccezione.

```
import java.util.Scanner;

public class PrendiLatteConEccezioni {

    public static void main(String[] args) {

        Scanner tastiera = new Scanner(System.in);

        try {
            System.out.println("Inserire il numero di ciambelle:");
            int conteggioCiambelle = tastiera.nextInt();

            System.out.println("Inserire il numero di bicchieri " +
                               "di latte:");
            int conteggioLatte = tastiera.nextInt();

            if (conteggioLatte < 1)
                throw new Exception("Eccezione: Niente latte!");

            double ciambellePerBicchiere =
                    conteggioCiambelle / (double)conteggioLatte;
            System.out.println(conteggioCiambelle + " ciambelle.");
            System.out.println(conteggioLatte + " bicchieri di latte.");
            System.out.println("Hai " + ciambellePerBicchiere +
                               " ciambelle per ogni bicchiere di latte.");

        } catch(Exception e) {
            System.out.println(e.getMessage());
            System.out.println("Vai a comprare del latte.");
        }

        System.out.println("Fine programma.");
    }
}
```

Si presuppone che l'utente inserisca un valore positivo come numero di bicchieri di latte.

`conteggioLatte` è positivo, quindi NON viene lanciata alcuna eccezione.

Questo codice NON viene eseguito

L'istruzione `throw` viene eseguita solamente in circostanze eccezionali, ma quando viene eseguita, lancia un'eccezione di una qualche classe (finora, si è parlato solo della classe `Exception`, ma più avanti verranno presentate altre classi). Quando viene lanciata un'eccezione, l'esecuzione del blocco `try` termina. Tutto il codice rimanente definito nel

blocco `try` viene ignorato e il controllo passa a un eventuale blocco `catch` appropriato. Un blocco `catch` è legato al solo blocco `try` immediatamente precedente. Se un'eccezione viene lanciata e catturata, l'oggetto eccezione viene assegnato all'identificativo del parametro del blocco `catch` e poi vengono eseguite le istruzioni contenute nel blocco `catch`. Al termine dell'esecuzione del codice contenuto nel blocco `catch`, il programma prosegue con il codice posto all'esterno dell'ultimo blocco `catch`. In altre parole, il controllo non ritorna al blocco `try`. Quindi non viene eseguita nessuna delle istruzioni del blocco `try` poste dopo l'istruzione che ha lanciato l'eccezione, come si può vedere dal Listato 13.4. Più avanti si vedrà anche cosa accade qualora non sia presente un blocco `catch` appropriato.

LISTATO 13.4 Flusso di controllo quando viene generata un'eccezione.

```
import java.util.Scanner;

public class PrendiLatteConEccezioni {

    public static void main(String[] args) {

        Scanner tastiera = new Scanner(System.in);

        try {
            System.out.println("Inserire il numero di ciambelle:");
            int conteggioCiambelle = tastiera.nextInt();

            System.out.println("Inserire il numero di bicchieri " +
                               "di latte:");
            int conteggioLatte = tastiera.nextInt();

            if (conteggioLatte < 1)
                throw new Exception("Eccezione: Niente latte!");

            double ciambellePerBicchiere =
                   conteggioCiambelle / (double)conteggioLatte;
            System.out.println(conteggioCiambelle + " ciambelle.");
            System.out.println(conteggioLatte + " bicchieri di latte.");
            System.out.println("Hai " + ciambellePerBicchiere +
                               " ciambelle per ogni bicchiere di latte.");

        } catch(Exception e) {
            System.out.println(e.getMessage());
            System.out.println("Vai a comprare del latte.");
        }

        System.out.println("Fine programma.");
    }
}
```

Si presuppone che l'utente inserisca 0 come numero di bicchieri di latte e, di conseguenza, che venga lanciata un'eccezione.

`conteggioLatte` è 0, quindi qui VIENE lanciata un'eccezione.

Questo codice NON viene eseguito

Quando il blocco `try` viene eseguito normalmente fino al completamento, senza quindi generare alcuna eccezione, l'esecuzione del programma prosegue con il codice che si trova dopo l'ultimo blocco `catch`. In altre parole, se non viene generata alcuna eccezione, tutti i relativi blocchi `catch` vengono ignorati, come si può osservare nel Listato 13.3.

La precedente spiegazione potrebbe portare a pensare che la sequenza `try-throw-catch` sia equivalente a un'istruzione `if-else`. Tali strutture sono quasi equivalenti, tranne per il messaggio incapsulato nell'eccezione lanciata. Un'istruzione `if-else` non può inviare un messaggio a uno dei suoi rami. Questa potrebbe non sembrare una differenza significativa, ma come si vedrà, la possibilità di inviare un messaggio dona al meccanismo di gestione delle eccezioni una maggiore versatilità rispetto a un'istruzione `if-else`.

Un'eccezione è un oggetto

Un'istruzione `throw` come la seguente:

```
throw new Exception("Carattere non valido.");
```

non specifica solamente un'azione che viene eseguita e subito dimenticata. Crea un oggetto che contiene un messaggio. Nell'esempio il messaggio è `"Carattere non valido."`.

Per comprendere quanto affermato, si può notare come l'istruzione `throw` precedente sia equivalente alla seguente:

```
Exception oggettoEccezione = new Exception("Carattere non valido.");
throw oggettoEccezione;
```

La prima di queste istruzioni invoca un costruttore della classe `Exception`, creando un oggetto di tipo `Exception`. La seconda istruzione lancia questo oggetto eccezione appena creato. Tale oggetto (e il messaggio che contiene) saranno quindi disponibili in un blocco `catch`. Di conseguenza, l'effetto del lancio di un'eccezione è qualcosa di più che un semplice trasferimento di controllo alla prima istruzione di un blocco `catch`.

Lanciare eccezioni

L'istruzione `throw` lancia un'eccezione.

Sintassi

```
throw new nome_della_classe_eccezione(possibili_argomenti);
```

Un'istruzione `throw` è solitamente inserita in un'istruzione `if` oppure `if-else`. Un blocco `try` può contenere un numero qualsiasi di istruzioni `throw` esplicite o invocazioni di metodi che possono lanciare eccezioni.

Esempio

```
throw new Exception("Terminazione dell'input inattesa.");
```

Gestire le eccezioni

Le istruzioni `try` e `catch`, utilizzate insieme, sono il meccanismo di base per gestire le eccezioni.

Sintassi

```
try {
    codice_da_provare
    eventuale_generazione_di_eccezione
    altro_codice
} catch (nome_della_classe_ eccezione parametro_del_blocco_catch) {
    gestione_della_eccezione_di_tipo_nome_della_classe_ eccezione
}
possibile_presenza_di_altri_blocchi_catch
```

Se all'interno del blocco `try` viene generata un'eccezione, la parte rimanente del blocco `try` viene ignorata e l'esecuzione prosegue con il primo blocco `catch` che corrisponde al tipo dell'eccezione generata. Dopo l'esecuzione del blocco `catch`, il programma prosegue con il codice che segue l'ultimo blocco `catch` definito.

Se nel blocco `try` non viene generata alcuna eccezione, dopo che ha completato l'esecuzione delle istruzioni del blocco, l'esecuzione del programma prosegue con il codice che segue l'ultimo blocco `catch` definito. In altre parole, se non viene generata alcuna eccezione, i blocchi `catch` vengono ignorati.

Per ogni blocco `try` è possibile definire più blocchi `catch`, ma ogni blocco `catch` può gestire un solo tipo di eccezione. L'identificativo *parametro_del_blocco_catch* serve come segnaposto per un'eccezione che potrebbe essere lanciata. Quando nel blocco `try` precedente viene generata un'eccezione della classe *nome_della_classe_eccezione*, questa eccezione viene assegnata a *parametro_del_blocco_catch*. Il codice nel blocco `catch` può utilizzare *parametro_del_blocco_catch* per gestire l'eccezione. Una scelta ampiamente diffusa per l'identificativo di *parametro_del_blocco_catch* è `e`; ciononostante, è possibile utilizzare ogni altro identificativo valido.

Il metodo `getMessage`

Ogni oggetto eccezione ha una variabile di istanza di tipo `String` che contiene un messaggio. Tale messaggio solitamente identifica la ragione per cui è stata generata l'eccezione. Per esempio, se l'eccezione è lanciata dalla seguente istruzione:

```
throw new Exception(argomento_di_tipo_stringa);
```

il valore della variabile di istanza di tipo `String` è *argomento_di_tipo_string*. Se l'oggetto eccezione viene chiamato `e`, l'invocazione di `e.getMessage()` restituisce questa stringa.

13.1.2 Classi di eccezioni predefinite

Quando si inizia a utilizzare i metodi delle classi predefinite, ci si accorge che a volte questi possono lanciare certi tipi di eccezioni. Tali eccezioni appartengono a classi predefinite all'interno della Java Class Library. Se si utilizza uno di questi metodi, è possibile inserire la sua invocazione in un blocco `try` e utilizzare il blocco `catch` per catturare la sua eventuale eccezione. Il nome delle eccezioni predefinite è stato scelto in modo da chiarirne il significato; ecco alcuni esempi di classi di eccezioni predefinite:

`BadStringOperationException` (operazione non consentita sulla stringa)
`ClassNotFoundException` (classe non trovata)
`IOException` (errore in input o in output)
`NoSuchMethodException` (metodo inesistente)

Quando si cattura un'eccezione di una di queste classi predefinite, la stringa restituita dal metodo `getMessage` fornisce solitamente informazioni sufficienti per identificare la causa dell'eccezione. Di conseguenza, supponendo di avere una classe chiamata `ClasseEsempio` e che questa classe abbia un metodo chiamato `faiQualcosa` che genera un'eccezione di tipo `IOException`, è possibile utilizzare il seguente codice:

```
ClasseEsempio oggetto = new ClasseEsempio();
try {
    <Possibile presenza di altro codice>
    oggetto.faiQualcosa(); //Potrebbe lanciare IOException
    <Possibile presenza di altro codice>
} catch(IOException e) {
    <Codice per gestire l'eccezione. Probabilmente include la seguente istruzione:>
    System.out.println(e.getMessage());
}
```

Se si ritiene che l'esecuzione del programma sia impossibile dopo la generazione dell'eccezione, il blocco `catch` può includere un'invocazione a `System.exit` per terminare il programma, come specificato di seguito:

```
catch(IOException e) {
    System.out.println(e.getMessage());
    System.out.println("Programma interrotto");
    System.exit(0);
}
```

Video
Utiliz
classi
ecce
prede

Catturare specifiche eccezioni

Sebbene sia possibile utilizzare la classe `Exception` in un blocco `catch`, come si è visto negli esempi iniziali del capitolo, è più utile catturare eccezioni più specifiche, come `IOException`.

Importare le eccezioni

La maggior parte delle eccezioni presentate in questo testo non hanno bisogno di essere importate in quanto sono presenti nel package `java.lang`. Alcune, però, sono contenute in package differenti e necessitano, quindi, di essere importate. Per esempio, la classe `IOException` si trova nel package `java.io`. Quando si esamina la documentazione di un'eccezione si deve prestare attenzione al package che la contiene, in modo da inserire l'eventuale istruzione `import` necessaria.

13.2 Definire nuove classi di eccezioni

È possibile definire nuove classi di eccezioni, che tuttavia devono essere classi derivate da una qualche classe di eccezione già definita. Una classe di eccezione può essere derivata da una classe di eccezione predefinita o anche da una nuova classe di eccezione precedentemente definita. Gli esempi che seguiranno useranno classi derivate dalla classe `Exception`.

Quando si definisce una classe di eccezione, i costruttori spesso costituiscono i metodi più importanti, se non addirittura gli unici, a parte quelli ereditati dalla classe base. Per esempio, il Listato 13.5 contiene la classe di eccezione `DivisionePerZeroException`, i cui unici metodi sono il costruttore di default e un costruttore che accetta come parametro una stringa. Per gli scopi dell'esempio proposto, questo è tutto ciò che occorre definire. In ogni caso, la classe eredita tutti i metodi della classe base `Exception`. In particolare, la classe `DivisionePerZeroException` eredita il metodo `getMessage` che restituisce un messaggio in formato stringa. Tale messaggio è specificato con la seguente istruzione posta nella definizione del costruttore di default:

```
super("Divisione per zero!");
```

Questa istruzione invoca il costruttore della classe base `Exception`. Come si è già notato, quando si passa una stringa al costruttore della classe `Exception`, il valore di tale stringa viene assegnato a una variabile di istanza di tipo `String`. Vi è la possibilità di recuperare successivamente questo valore invocando il metodo `getMessage`, che è un metodo *get* della classe `Exception` ed è ereditato dalla classe `DivisionePerZeroException`.

MyLab

LISTATO 13.5 Una nuova classe di eccezione definita dal programmatore.

```
public class DivisionePerZeroException extends Exception {

    public DivisionePerZeroException() {
        super("Divisione per zero!");
    }

    public DivisionePerZeroException(String messaggio) {
        super(messaggio);
    }
}
```

È possibile fare molto di più in un costruttore di un'eccezione, ma questa forma è comune.

`super` è un'invocazione al costruttore della classe base `Exception`.

Per esempio, il Listato 13.6 mostra un semplice programma che utilizza questa classe di eccezione. L'eccezione viene creata tramite il costruttore di default e quindi lanciata, come segue:

```
throw new DivisionePerZeroException();
```

Questa eccezione è catturata dal blocco `catch` che contiene la seguente istruzione:

```
System.out.println(e.getMessage());
```

Questa istruzione visualizza il seguente output, come mostrato dall'output di esempio nel Listato 13.6:

```
Divisione per zero!
```

La classe `DivisionePerZeroException` presentata nel Listato 13.5 definisce anche un secondo costruttore. Questo costruttore ha un parametro di tipo `String` che permette di scegliere un messaggio nel momento in cui si lancia l'eccezione. Se l'istruzione di `throw` nel Listato 13.6 fosse stata:

```
throw new DivisionePerZeroException(
        "Oops. Non avrei dovuto utilizzare lo zero.");
```

l'istruzione

```
System.out.println(e.getMessage());
```

avrebbe prodotto il seguente output:

```
Oops. Non avrei dovuto utilizzare lo zero.
```

LISTATO 13.6 Utilizzo di una nuova classe di eccezione definita dal programmatore.

```
import java.util.Scanner;

public class DividiPerZeroDemo {

    private int numeratore;
    private int denominatore;
    private double quoziente;

    public void fai() {
        try {
            System.out.println("Inserisci numeratore:");
            Scanner tastiera = new Scanner(System.in);
            numeratore = tastiera.nextInt();
            System.out.println("Inserisci denominatore:");
            denominatore = tastiera.nextInt();

            if (denominatore == 0)
                throw new DivisionePerZeroException();

            quoziente = numeratore / (double)denominatore;
            System.out.println(numeratore + "/" + denominatore +
                               " = " + quoziente);
```

Più avanti nel capitolo verrà presentata una versione migliorata di questo programma.

```
        } catch(DivisionePerZeroException e) {
            System.out.println(e.getMessage());
            daiSecondaPossibilita();
        }

        System.out.println("Fine programma.");
    }

    public void daiSecondaPossibilita() {
        System.out.println("Tenta di nuovo.");
        System.out.println("Inserisci numeratore:");
        Scanner tastiera = new Scanner(System.in);
        numeratore = tastiera.nextInt();

        System.out.println("Inserisci denominatore:");
        System.out.println("Accertati che il denominatore non sia zero.");
        denominatore = tastiera.nextInt();

        if (denominatore == 0) {
            System.out.println("Non posso dividere per zero.");
            System.out.println("Poiche' non posso fare cio' che chiedi,");
            System.out.println("il programma terminera' ora.");
            System.exit(0);
        }

        quoziente = ((double)numeratore) / denominatore;
        System.out.println(numeratore + "/" + denominatore +
                           " = " + quoziente);
    }

    public static void main(String[] args) {
        DividiPerZeroDemo unaVolta = new DividiPerZeroDemo();
        unaVolta.fai();
    }
}
```

A volte è meglio gestire un caso eccezionale senza lanciare un'eccezione.

Esempio di output 1

```
Inserisci numeratore:
5
Inserisci denominatore:
10
5/10 = 0.5
Fine programma.
```

Esempio di output 2

```
Inserisci numeratore:
5
Inserisci denominatore:
0
Divisione per zero!
Tenta di nuovo.
```

```
Inserisci numeratore:
5
Inserisci denominatore:
Accertati che il denominatore non sia zero.
10
5/10 = 0.5
Fine programma.
```

Esempio di output 3

```
Inserisci numeratore:
5
Inserisci denominatore:
0
Divisione per zero!
Tenta di nuovo.
Inserisci numeratore:
5
Inserisci denominatore:
Accertati che il denominatore non sia zero.
0
Non posso dividere per zero.
Poiche' non posso fare cio' che chiedi,
il programma terminera' ora.
```

Si noti che il blocco `try` del Listato 13.6 contiene la parte del programma che definisce la condizione normale di funzionamento. Se tutto si svolge normalmente, questo sarà l'unico codice che verrà eseguito e l'output sarà come quello del primo esempio di output. Nei casi eccezionali, ovvero quando l'utente inserisce zero come denominatore, viene lanciata l'eccezione che verrà poi catturata dal blocco `catch`. Il blocco `catch` visualizza il messaggio dell'eccezione e quindi richiama il metodo `daiSecondaPossibilita`. Il metodo `daiSecondaPossibilita` offre all'utente una seconda opportunità di inserire l'input correttamente e proseguire nella computazione. Se poi l'utente tenta una seconda volta di eseguire una divisione per zero, il metodo termina il programma senza generare una nuova eccezione. Il metodo `daiSecondaPossibilita` termina l'esecuzione del programma esclusivamente se l'utente inserisce per due volte 0 come denominatore. Il programma presentato nel Listato 13.6 mantiene la gestione del caso eccezionale in un metodo distinto, in modo da mantenere più pulita la parte di codice che tratta i casi normali.

Preservare `getMessage` nelle classi di eccezioni personalizzate

In tutte le classi di eccezioni predefinite, il metodo `getMessage` restituisce la stringa che è stata passata come argomento al costruttore. Se al costruttore non è stato passato alcun argomento (ovvero è stato chiamato il costruttore di default) `getMessage` restituisce una stringa di default. Per esempio, si supponga che l'eccezione sia stata lanciata come segue:

```
throw new Exception("Questa e' una grande eccezione!");
```

Il valore della variabile di istanza di tipo `String` è impostato a `"Questa e' una grande eccezione!"`. Se l'oggetto eccezione è chiamato `e`, il metodo `e.getMessage()` restituisce `"Questa e' una grande eccezione!"`.

Si dovrebbe preservare il comportamento del metodo `getMessage` in ogni classe di eccezione che viene definita. Si supponga, per esempio, di definire una classe di eccezione chiamata `LaMiaEccezioneSpeciale` e di lanciare un'eccezione come segue:

```
throw new LaMiaEccezioneSpeciale("Wow, questa e' un'eccezione!");
```

Se `e` è il nome dell'eccezione lanciata, `e.getMessage()` dovrebbe restituire `"Wow, questa e' un'eccezione!"`. Per assicurarsi che le classi di eccezioni personalizzate che vengono definite si comportino in questo modo, ci si deve preoccupare che abbiano un costruttore con un parametro di tipo stringa la cui definizione inizi con una chiamata a `super`, come illustrato nel seguente costruttore:

```
public LaMiaEccezioneSpeciale(String messaggio) {
    super(messaggio);
    //Può esservi del codice qui, ma spesso non vi è nient'altro.
}
```

La chiamata a `super` è un'invocazione al costruttore della classe base. Se il costruttore della classe base gestisce correttamente il messaggio, si è sicuri che lo faranno anche tutte le classi definite in questo modo. Si dovrebbe sempre includere un costruttore di default in ogni classe di eccezione. Questo costruttore di default dovrebbe impostare un valore di default recuperabile da `getMessage`. La definizione del costruttore dovrebbe iniziare con una chiamata a `super`, come illustrato nel seguente costruttore:

```
public LaMiaEccezioneSpeciale() {
    super("LaMiaEccezioneSpeciale e' stata lanciata");
    //Può esservi del codice qui, ma spesso non vi è nient'altro.
}
```

Se `getMessage` funziona come descritto per la classe base, questo costruttore di default funzionerà correttamente per la nuova classe di eccezione definita.

Caratteristiche di un oggetto eccezione

Le due caratteristiche più importanti di un oggetto eccezione sono le seguenti.

- Il tipo dell'oggetto, ovvero il nome della classe di eccezione. I prossimi paragrafi spiegheranno perché è importante.
- Il messaggio che l'oggetto si porta dietro memorizzato in una variabile di istanza di tipo `String`. Questa stringa può essere recuperata chiamando il metodo `getMessage`. La stringa permette al codice di inviare un messaggio insieme a un oggetto eccezione, in modo che il blocco `catch` possa recuperarlo.

Quando occorre definire una classe di eccezione?

Se nel codice si andrà a inserire un'istruzione `throw`, è buona norma definire una propria classe di eccezione. In questo modo, quando il codice cattura un'eccezione, i blocchi `catch` possono differenziare tra le eccezioni personalizzate e le eccezioni generate dalle eventuali invocazioni di metodi definiti nelle classi predefinite. Per esempio, nel Listato 13.6 è stata utilizzata la classe di eccezione `DivisionePerZeroException`, che era stata definita nel Listato 13.5.

La cosa da evitare è quella di cedere alla tentazione di utilizzare una classe di eccezione predefinita nella Java Class Library. Per esempio, si sarebbe tentati di utilizzare la classe predefinita `Exception` per lanciare l'eccezione nel Listato 13.6, scrivendo come segue:

```
throw new Exception("Divisione per zero!");
```

Sarebbe stato quindi possibile catturare l'eccezione nel blocco `catch` come segue:

```
catch(Exception e) {
    System.out.println(e.getMessage());
    daiSecondaPossibilita();
}
```

Sebbene questo approccio sia in grado di funzionare per il programma presentato nel Listato 13.6, questa non è la scelta ottimale, poiché il `catch` descritto sopra catturerà ogni tipo di eccezione, per esempio anche `IOException`. Ma una `IOException` potrebbe dover essere gestita in maniera differente rispetto al codice presente nel metodo `daiSecondaPossibilita`. Perciò, piuttosto che utilizzare la classe `Exception` per gestire le divisioni per 0, è meglio utilizzare la nuova classe, più specifica, `DivisionePerZeroException` definita dal programmatore, come è stato fatto nel Listato 13.6.

Classi di eccezioni definite dal programmatore

Vi è la possibilità di definire nuove classi di eccezioni, a patto di derivarle da una classe di eccezione esistente, sia essa predefinita o definita dal programmatore.

Linee guida

- Si utilizzi la classe `Exception` come base, se non vi è una particolare esigenza che porta a scegliere come classe base un'altra classe.
- Si definiscano almeno due costruttori che includono un costruttore di default e uno con un solo parametro di tipo `String`.
- Si dovrebbe iniziare ogni definizione di costruttore con una chiamata al costruttore della classe base, utilizzando `super`. Nel costruttore di default, la chiamata a `super` deve avere un argomento di tipo stringa che indichi il tipo di eccezione rappresentata. Per esempio:

```
super("Eccezione onda di marea lanciata!");
```

Se il costruttore ha un parametro di tipo `String` chiamato `messaggio` il parametro dovrebbe essere l'argomento della chiamata a `super`. Per esempio:

```
super(messaggio);
```

In questo modo, la stringa può essere recuperata utilizzando il metodo `getMessage`.

- La classe di eccezione eredita il metodo `getMessage`, che non dovrebbe essere ridefinito.
- Normalmente non è necessario definire nessun altro metodo, anche se sarebbe lecito farlo.

Esempio

```
public class EccezioneOndaDiMarea extends Exception {
    public EccezioneOndaDiMarea() {
        super("Eccezione onda di marea lanciata");
    }

    public EccezioneOndaDiMarea(String messaggio) {
        super(messaggio);
    }
}
```

`super` è una chiamata al costruttore della classe base `Exception`.

Le classi di eccezione non possono essere generiche

Se si prova a definire una classe di eccezione come in questo esempio

```
public class MiaEccezione<T> extends Exception // Illegale
```

si otterrà un errore in fase di compilazione. Lo stesso avverrà utilizzando, al posto della classe `Exception`, `Error`, `Throwable` o qualunque altra classe derivata da `Throwable`. Non è possibile costruire classi generiche i cui oggetti possano essere lanciati come eccezioni.

13.3 Approfondimenti sulle classi di eccezioni

In questa parte del capitolo si discuteranno alcune tecniche avanzate, ma comunque fondamentali, di gestione delle eccezioni.

13.3.1 Dichiarare le eccezioni

A volte è sensato ritardare la gestione di un'eccezione. Per esempio, si potrebbe avere un metodo il cui codice lancia un'eccezione, che però non cattura. Un programma che utilizza quel metodo, per esempio, potrebbe dover terminare la propria esecuzione nel momen-

to in cui si verifica l'eccezione. Un altro programma potrebbe invece essere in grado di gestire l'eccezione e di continuare l'esecuzione. La gestione dell'eccezione dipende quindi dal programma che utilizza il metodo. Di conseguenza, il metodo stesso non sarebbe in grado di gestire l'eccezione, anche se la catturasse. In queste situazioni, è sensato non catturare l'eccezione nella definizione del metodo, ma fare in modo che il codice che utilizza il metodo inserisca l'invocazione del metodo stesso in un blocco `try` e catturi l'eccezione in un blocco `catch` che segue quel blocco `try`.

Se un metodo non cattura l'eccezione, deve almeno informare il programmatore che ogni sua invocazione potrebbe causare un'eccezione. Questa avvertenza è chiamata **clausola throws** (*throws clause*). Per esempio, un metodo che può lanciare `DivisionePerZeroException` ma che non cattura l'eccezione, avrà un'intestazione come la seguente:

```
public void metodoDiEsempio() throws DivisionePerZeroException
```

La parte `throws DivisionePerZeroException` è una clausola `throws`. Essa dichiara che un'invocazione al metodo `metodoDiEsempio` può lanciare una `DivisionePerZeroException`. La maggior parte delle eccezioni che possono essere generate dall'invocazione di un metodo devono essere gestite in uno di questi due modi.

- Si cattura la possibile eccezione in un blocco `catch` definito all'interno del metodo stesso.
- Si dichiara la possibile eccezione utilizzando la clausola `throws` nell'intestazione del metodo e si delega la gestione dell'eccezione a chi utilizza il metodo.

In ogni metodo, è possibile utilizzare entrambe le alternative catturando alcune eccezioni e dichiarandone altre nella clausola `throws`.

Si è già visto come gestire le eccezioni in un blocco `catch`. La seconda tecnica prevede di scaricare la responsabilità della gestione a chi utilizzerà il metodo. Si supponga, per esempio, che il `metodoA` abbia una clausola `throws` come segue:

```
public void metodoA() throws DivisionePerZeroException
```

In questo caso, il `metodoA` è sollevato dalla responsabilità di catturare ogni eccezione di tipo `DivisionePerZeroException` che potrebbe accadere durante la sua esecuzione. Se, però, la definizione del `metodoB` include un'invocazione al `metodoA`, il `metodoB` deve gestire l'eccezione. Quando il `metodoA` aggiunge la clausola `throws`, sta "dicendo" al `metodoB`, "Se mi invochi, devi preoccuparti di ogni `DivisionePerZeroException` che potrei lanciare". In effetti, il `metodoA` trasferisce la responsabilità di ogni eccezione di tipo `DivisionePerZeroException` da sé a ogni metodo che lo invoca.

Ovviamente, così come il `metodoA` trasferisce la responsabilità al `metodoB` includendo una clausola `throws` nella propria intestazione, anche il `metodoB` può, allo stesso modo, passare la responsabilità a qualsiasi metodo lo chiami, includendo la stessa clausola `throws` nella propria definizione. In un programma ben scritto, ogni eccezione sollevata dovrebbe prima o poi essere catturata da un blocco `catch` definito in un qualche metodo che non scarica alcuna responsabilità.

Una clausola `throws` può, inoltre, contenere più tipi di eccezioni. In questi casi, i tipi di eccezioni devono essere separati da una virgola:

```
public int mioMetodo() throws IOException, DivisionePerZeroException
```

Lanciare un'eccezione può causare la terminazione di un metodo

Se un metodo lancia un'eccezione e l'eccezione non viene catturata all'interno del metodo, l'invocazione a questo metodo termina immediatamente dopo che l'eccezione è stata lanciata.

Se una classe derivata ridefinisce un metodo di una classe base che ha una clausola `throws`, non è possibile aggiungere eccezioni alla clausola `throws` al metodo ridefinito. Quindi, se il metodo in questione genera un tipo di eccezione che non è presente nella clausola `throws` del metodo ridefinito presente nella classe base, esso deve gestire l'eccezione in un blocco `try-catch`. Vi è però la possibilità di dichiarare meno eccezioni nella clausola `throws` del metodo ridefinito.

MyLab

Video 13.2 Utilizzare la clausola `throws`

Il Listato 13.7 mostra una versione riveduta del programma presentato nel Listato 13.6 in cui il comportamento del caso normale è definito nel metodo `casoNormale()`. Questo metodo può lanciare un'eccezione di tipo `DivisionePerZeroException`, ma non la cattura. Perciò è necessario dichiarare nell'intestazione del metodo questa possibile eccezione mediante la clausola `throws`. Se si organizza il programma in questo modo, il funzionamento nel caso normale rimane completamente isolato e facile da leggere. Non è nemmeno confuso da blocchi `try` e `catch`. Comunque, quando il metodo `main` invoca il metodo `casoNormale`, lo deve fare all'interno di un blocco `try`.

La clausola `throws`

Se si definisce un metodo che può lanciare un'eccezione di una particolare classe, normalmente si deve catturare l'eccezione in un blocco `catch` all'interno della definizione del metodo oppure dichiarare la classe dell'eccezione scrivendo una clausola `throws` nell'intestazione del metodo.

Sintassi

```
public tipo_di_ritorno nome_del_metodo(lista_di_parametri) throws lista_di_eccezioni
    corpo_del_metodo
```

Esempio

```
public void metodoA(int n) throws IOException, MiaEccezione {
  ...
}
```

`throw` vs. `throws`

La parola chiave `throw` è utilizzata per lanciare un'eccezione, mentre `throws` è utilizzata nell'intestazione del metodo per dichiarare un'eccezione. Quindi, un'istruzione `throw` lancia un'eccezione e una clausola `throws` ne dichiara una.

LISTATO 13.7 Scaricare la responsabilità utilizzando la clausola `throws`.

MyLab

```
import java.util.Scanner;

public class FaiDivisione {

    private int numeratore;
    private int denominatore;
    private double quoziente;

    public void casoNormale() throws DivisionePerZeroException {
        System.out.println("Inserisci numeratore:");
        Scanner tastiera = new Scanner(System.in);
        numeratore = tastiera.nextInt();
        System.out.println("Inserisci denominatore:");
        denominatore = tastiera.nextInt();

        if (denominatore == 0)
            throw new DivisionePerZeroException();

        quoziente = numeratore / (double)denominatore;
        System.out.println(numeratore + "/" + denominatore +
                           " = " + quoziente);
    }

    public static void main(String[] args) {
        FaiDivisione fai = new FaiDivisione();

        try {
            fai.casoNormale();
        } catch(DivisionePerZeroException e) {
            System.out.println(e.getMessage());
            fai.daiSecondaPossibilita();
        }
        System.out.println("Fine programma.");
    }
}
```

Il metodo `daiSecondaPossibilita` e gli esempi di input/output sono identici a quelli presentati nel Listato 13.6.

13.3.2 Tipi di eccezioni

Nei paragrafi precedenti si è detto che, nella maggioranza dei casi, un'eccezione deve essere catturata da un blocco `catch` oppure deve essere dichiarata in una clausola `throws` nell'intestazione del metodo. Quanto descritto costituisce la regola di base, ma vi sono eccezioni; ebbene sì, un'eccezione a una regola riguardante le eccezioni! Java ha alcune particolari eccezioni che non richiedono di essere trattate come tali, sebbene sia sempre possibile farlo catturandole in un blocco `catch`.

Java suddivide tutte le eccezioni in due categorie: *controllate* e *non controllate*. Un'**eccezione controllata** (*checked exception*) deve essere catturata in un blocco `catch` oppure dichiarata in una clausola `throws`. Queste eccezioni spesso indicano la presenza di seri problemi che potrebbero portare alla terminazione del programma. Le eccezioni `BadStringOperationException`, `ClassNotFoundException`, `IOException` e `NoSuchMethodException`, menzionate in precedenza in questo stesso capitolo, sono tutte eccezioni controllate della Java Class Library.

Un'**eccezione non controllata** (*unchecked exception*) o **eccezione run-time** può non essere catturata in un blocco `catch` o dichiarata in una clausola `throws`. Queste eccezioni solitamente indicano che nel codice vi è qualcosa di sbagliato, che dovrebbe essere corretto. Normalmente, per queste eccezioni non si è scritta un'istruzione `throw`. Esse sono solitamente generate durante la valutazione di un'espressione o lanciate da un metodo presente in una delle classi predefinite. Per esempio, se un programma tenta di utilizzare un indice che non rientra nei limiti di un array, viene generata un'eccezione `ArrayIndexOutOfBoundsException`. Se un'operazione aritmetica causa un problema, come una divisione per zero, viene generata un'eccezione `ArithmeticException`. Per queste eccezioni è necessario correggere il codice e non aggiungere un blocco `catch`. Un'eccezione a run-time non catturata, causa la terminazione del programma.

Come è possibile sapere se un'eccezione è controllata o non controllata? Si può consultare la documentazione della Java Class Library per conoscere la classe base dell'eccezione e da quella dedurre se è controllata o meno. La Figura 13.1 mostra la gerarchia delle classi di eccezioni predefinite. La classe `Exception` è la classe base da cui discende ogni altra classe di eccezione. In pratica, una classe di eccezione deriva direttamente dalla classe `Exception` o da una classe che a sua volta deriva (anche non direttamente) dalla classe `Exception`. Le classi di eccezioni non controllate sono derivate dalla classe `RuntimeException`. Tutte le altre classi di eccezioni sono controllate e devono essere catturate.

Non è necessario preoccuparsi troppo di quali eccezioni siano o non siano da catturare o dichiarare in una clausola `throws`. Se ci si dimentica di qualche eccezione che richiede una gestione, il compilatore ce ne informerà. A questo punto, si può decidere di catturare l'eccezione o aggiungerla in una clausola `throws`.

Tipologie di eccezioni

Ogni eccezione è discendente della classe `Exception`. `RuntimeException` è una classe derivata da `Exception` e le classi a loro volta derivate da `RuntimeException` o dalle sue discendenti rappresentano eccezioni non controllate. Tali eccezioni non richiedono di essere catturate o dichiarate in una clausola `throws` dell'intestazione di un metodo. Tutte le altre eccezioni sono controllate e devono essere catturate o dichiarate in una clausola `throws`.

13.3.3 Errori

Un **errore** (*error*) è un oggetto della classe `Error`, derivata da `Throwable`, come indicato nella Figura 13.1. Si noti che `Throwable` è anche la classe base di `Exception`. Tecnicamente parlando, la classe `Error` e le classi che da essa discendono non sono considerate classi di eccezioni, poiché non discendono dalla classe `Exception`. Sono però abbastanza

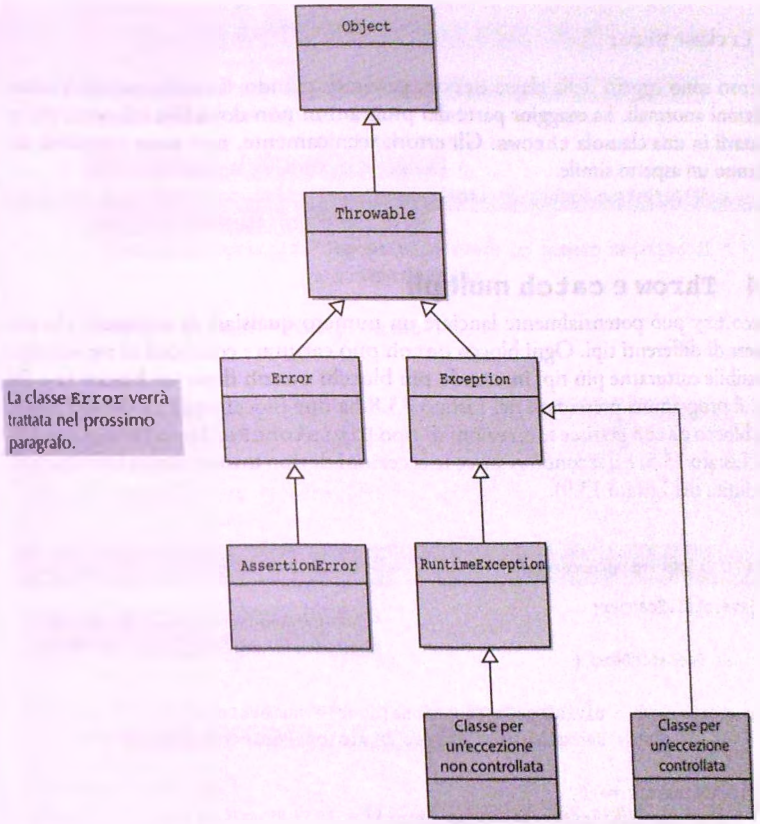


Figura 13.1 Gerarchia delle classi di eccezioni predefinite.

simili alle eccezioni. In particolare, gli oggetti della classe `Error` sono simili a eccezioni non controllate, poiché non vi è necessità di catturarli o dichiararli in una clausola `throws`, anche se questo è comunque possibile. Gli errori sono il più delle volte fuori dal controllo del programmatore. Per esempio, può verificarsi un `OutOfMemoryError` quando il programma ha esaurito la memoria disponibile. Questo significa che si deve o modificare il programma affinché utilizzi meno memoria o cambiare le impostazioni affinché Java possa accedere a più memoria o acquistare ulteriore memoria per il computer. L'aggiunta di un blocco `catch` non sarà di alcun aiuto.

Quando si è parlato dell'operatore `assert` e del controllo delle asserzioni nel Capitolo 4, si è detto che, se un programma contiene un controllo di asserzione e l'asserzione fallisce, il programma terminerà con un messaggio d'errore. Ciò che accade realmente è che viene generato un errore di tipo `AssertionError`. Come il nome suggerisce, la classe `AssertionError` è derivata dalla classe `Error`.

La classe `Error`

Gli errori sono oggetti della classe `Error`, generati quando si verificano determinate condizioni anormali. La maggior parte dei programmi non dovrebbe né catturarli, né dichiararli in una clausola `throws`. Gli errori, tecnicamente, non sono eccezioni, anche hanno un aspetto simile.

13.3.4 `Throw` e `catch` multipli

Un blocco `try` può potenzialmente lanciare un numero qualsiasi di eccezioni, che possono essere di differenti tipi. Ogni blocco `catch` può catturare eccezioni di un solo tipo, ma è possibile catturarne più tipi inserendo più blocchi `catch` dopo un blocco `try`. Per esempio, il programma presentato nel Listato 13.8 ha due blocchi `catch` dopo il blocco `try`. Un blocco `catch` gestisce le eccezioni di tipo `DivisionePerZeroException` (definita nel Listato 13.5) e il secondo gestisce le eccezioni di tipo `NumeroNegativoException` (definita nel Listato 13.9).

MyLab

LISTATO 13.8 Catturare più eccezioni.

Questo è solo un esempio di gestione delle eccezioni utilizzando due blocchi catch.

```
import java.util.Scanner;

public class DueCatchDemo {

   public static double divisioneConEccezione(double numeratore,
                double denominatore) throws DivisionePerZeroException {

      if (denominatore == 0)
          throw new DivisionePerZeroException();

      return numeratore / denominatore;
   }

   public static void main(String[] args) {

      try {
         System.out.println("Inserire il numero di oggetti prodotti:");
         Scanner tastiera = new Scanner(System.in);
         int oggetti = tastiera.nextInt();

         if (oggetti < 0)
             throw new NumeroNegativoException("oggetti");

         System.out.println("Quanti di questi erano difettosi?");
         int difettosi = tastiera.nextInt();
```

```
        if (difettosi < 0)
            throw new NumeroNegativoException("oggetti difettosi");

        double rapporto = divisioneConEccezione(oggetti, difettosi);
        System.out.println("Un oggetto ogni " + rapporto +
                           " e' difettoso");
    } catch(DivisionePerZeroException e) {
        System.out.println("Congratulazioni! Un record perfetto!");
    } catch(NumeroNegativoException e) {
        System.out.println("Impossibile avere un numero negativo di " +
                           e.getMessage());
    }
    System.out.println("Fine programma.");
  }
}
```

Esempio di output 1

```
Inserire il numero di oggetti prodotti:
1000
Quanti di questi erano difettosi?
500
Un oggetto ogni 2.0 e' difettoso
Fine programma.
```

Esempio di output 2

```
Inserire il numero di oggetti prodotti:
-10
Impossibile avere un numero negativo di oggetti
Fine programma.
```

Esempio di output 3

```
nserire il numero di oggetti prodotti:
000
uanti di questi erano difettosi?

ngratulazioni! Un record perfetto!
ne programma.
```

TATO 13.9 La classe `NumeroNegativoException`.

MyLab

```
olic class NumeroNegativoException extends Exception {

  public NumeroNegativoException() {
      super("Eccezione numero negativo!");
  }

  public NumeroNegativoException(String messaggio) {
      super(messaggio);
  }
```

Catturare prima l'eccezione più specifica

Quando si catturano più eccezioni, l'ordine dei blocchi `catch` può essere importante. Quando in un blocco `try` viene lanciata un'eccezione, i blocchi `catch` vengono esaminati nell'ordine in cui sono stati scritti nel programma. Viene eseguito il primo blocco compatibile con il tipo dell'eccezione lanciata. Di conseguenza, il seguente ordinamento di blocchi `catch` sarebbe da evitare:

```
catch(Exception e) { //Questo blocco catch non dovrebbe essere il primo.
   ...
} catch(DivisionePerZeroException e) {
   ...
}
```

Il secondo blocco `catch` non può mai essere raggiunto.

Con questo ordinamento, infatti, il blocco `catch` per `DivisionePerZeroException` non verrà mai utilizzato, poiché tutte le eccezioni vengono catturate dal primo blocco `catch`. Il corretto ordinamento prevede di invertire i blocchi di `catch`, in modo che le eccezioni più specifiche siano codificate prima rispetto a quelle più generiche, come nel seguente codice:

```
catch(DivisionePerZeroException e) {
   ...
} catch(Exception e) {
   ...
}
```

Input dell'utente

Quando un programma legge dati di input inseriti dall'utente, sarà probabilmente necessario gestire delle eccezioni. L'utente, infatti, può inserire (intenzionalmente o meno) qualsiasi cosa come input!

Gestione delle eccezioni e incapsulamento

L'invocazione di un metodo può generare un'eccezione di uno dei tipi dichiarati nella clausola `throws` del metodo. Ogni volta che viene generata un'eccezione, la si gestisce nello stesso modo indipendentemente dal fatto che sia stata generata con un'istruzione `throw` o chiamando un altro metodo: l'eccezione è gestita in un blocco `catch` o dichiarata in un'altra clausola `throws`. Quando si considera un'invocazione di un metodo che potrebbe generare un'eccezione, non è necessario preoccuparsi di dove l'eccezione venga effettivamente generata all'interno del corpo del metodo. Non importa in quale modo l'eccezione venga generata. L'unica cosa che importa è che l'invocazione del metodo potrebbe generare un'eccezione, che viene gestita allo stesso modo indipendentemente da ciò che accade nel corpo del metodo.

Gestione di più tipi di eccezione

A partire dalla versione 7 di Java, è possibile gestire più tipi di eccezioni all'interno di un singolo blocco `catch`.

Sintassi

```
try {
        ...
} catch (classe_eccezione_1 | classe_eccezione_2 | ... | classe_eccezione_n) {
        gestione_della_eccezione
}
```

Esempio

A volte può essere necessario gestire tipi diversi di eccezioni eseguendo le stesse istruzioni. Se i tipi da considerare non sono specializzazioni dello stesso tipo di eccezione, o se per qualunque motivo non è possibile utilizzare un singolo blocco `catch` per gestire un tipo di eccezione che comprenda tutti quelli da considerare, con le versioni di Java precedenti alla 7 sarebbe stato inevitabile replicare lo stesso codice in più blocchi `catch`, come in questo esempio:

```
try {
    ...
} catch (CharacterCodingException e) {
    System.out.println(e.getMessage());
} catch (CharConversionException e) {
    System.out.println(e.getMessage());
}
```

Con la versione 7 di Java si può eliminare la replicazione del codice come nell'esempio seguente:

```
try {
    ...
} catch (CharacterCodingException | CharConversionException e) {
    System.out.println(e.getMessage());
}
```

Nella clausola `catch` possono quindi essere specificati più tipi di eccezioni, separati da una barra verticale `|`, associabili al parametro `e`. Un blocco `catch` strutturato in questo modo potrà gestire eccezioni di uno qualunque dei tipi indicati.

FAQ Quando il codice dovrebbe lanciare un'eccezione?

L'uso dell'istruzione `throw` dovrebbe essere limitato ai casi in cui sia realmente indispensabile. Quando si sta valutando la possibilità di inserire un'istruzione `throw`, una buona strategia consiste nel pensare a come si scriverebbe il programma senza `throw`. Se si riesce a trovare un'alternativa che produca un codice ragionevole, probabilmente si può evitare di lanciare un'eccezione. Ma se il modo in cui si gestisce un caso anomalo dipende da come e dove viene invocato il metodo, l'approccio migliore consiste nel lasciare che il programmatore che invoca il metodo gestisca l'eccezione. In tutte le altre situazioni è preferibile evitare di lanciare eccezioni. I metodi predefiniti spesso lasciano la gestione delle eccezioni al programmatore. Nella documentazione di un metodo predefinito può essere indicato che il metodo lancia eccezioni di un certo tipo. Ci si aspetta che chiunque invochi quel metodo predefinito gestisca ogni tipo di eccezione che il metodo può lanciare.

Dove lanciare un'eccezione

Finora sono stati presentati alcuni semplici frammenti di codice per illustrare i concetti base della gestione delle eccezioni. Questi esempi sono però volutamente banali e, di conseguenza, poco realistici. In generale, il lancio e la gestione di un'eccezione andrebbero separati in metodi distinti. Per esempio, data la definizione della classe di eccezione `Eccezione`, un metodo potrebbe avere la seguente forma:

```
public void metodoA() throws Eccezione {
    ...
    throw new Eccezione("Bla Bla Bla");
    ...
}
```

mentre un altro metodo (magari addirittura in un'altra classe) potrebbe essere:

```
public void metodoB() {
    ...
    try {
        ...
        metodoA();
        ...
    } catch(Eccezione e) {
        ... //Gestione dell'eccezione.
    }
    ...
}
```

La ragione di tutto ciò è legata al discorso affrontato nel precedente riquadro FAQ, cioè a quando un'eccezione debba essere lanciata. Se un metodo sa come affrontare una certa situazione, allora la dovrebbe gestire senza lanciare alcuna eccezione. Se in-

vece viene lanciata un'eccezione da un frammento di codice all'interno di un metodo e si sta utilizzando proprio quel metodo per codificarne un altro, allora, se possibile, l'eccezione andrebbe gestita in fase di codifica del nuovo metodo. In caso contrario, è necessario dichiarare quel tipo di eccezione nella clausola `throws` del nuovo metodo e lasciare che chi utilizza il nuovo metodo la gestisca in un blocco `catch`.

Blocchi `try-catch` annidati

Sebbene sia possibile inserire un blocco `try` e i suoi blocchi `catch` all'interno di un altro blocco `try` o di un blocco `catch` più esterni, raramente questa strategia è veramente utile. Se si è tentati di annidare questi blocchi, si dovrebbe almeno valutare se esiste un altro modo per organizzare il codice. Per esempio, spesso è possibile eliminare completamente uno o più blocchi `try` oppure è possibile spostare il più interno dei blocchi `try-catch` nella definizione di un metodo e inserire l'invocazione di questo metodo come istruzione nel blocco `try` o `catch` più esterno. In ogni caso, di solito è meglio evitare di sviluppare blocchi `try-catch` annidati.

Si supponga di utilizzare comunque dei blocchi `try-catch` annidati. Se si inserisce un blocco `try` e i suoi blocchi `catch` in un `try` più esterno e se un'eccezione viene lanciata dal blocco `try` interno, ma non viene catturata da uno dei suoi blocchi `catch`, l'eccezione viene rilanciata al blocco `try` esterno e potrebbe essere catturata da uno dei suoi `catch`. Se invece si inserisce un blocco `try` e i suoi blocchi `catch` in un `catch` più esterno, è necessario usare nomi diversi per i parametri dei blocchi `catch` interni ed esterni.

13.3.5 Blocco `finally`

È possibile inserire un blocco `finally` dopo una sequenza di blocchi `catch`. Il codice nel blocco `finally` viene eseguito comunque, indipendentemente dal fatto che l'eccezione venga lanciata. Questo blocco offre la possibilità di risolvere problemi di coerenza che potrebbero crearsi in seguito a un'eccezione.

La sintassi generale è la seguente:

```
try {
    ...
}
uno_o_più_blocchi_catch
finally {
    codice_finally //Sempre eseguito.
}
```

Per comprendere il significato e i vantaggi di un blocco `finally`, si supponga che i blocchi `try-catch-finally` si trovino nella definizione di un metodo (in realtà, ogni gruppo di blocchi `try-catch-finally` si trova sempre all'interno di un metodo, compreso il metodo `main`).

Quando viene eseguito il blocco `try-catch-finally`, ci si può trovare in una delle tre situazioni seguenti.

- Il blocco `try` viene eseguito completamente e non vengono lanciate eccezioni. In questa situazione il blocco `finally` viene eseguito dopo il blocco `try`.
- Viene lanciata un'eccezione all'interno del blocco `try` e viene catturata da uno dei blocchi `catch` che seguono il blocco `try`. In questo caso il blocco `finally` viene eseguito al termine dell'esecuzione del blocco `catch`.
- Viene lanciata un'eccezione all'interno del blocco `try`, ma non esiste un blocco `catch` che sia in grado di catturarla. In questo caso viene eseguito il blocco `finally` e quindi il metodo termina rilanciando l'eccezione al metodo chiamante. Si noti che se non fosse stato previsto il blocco `finally` e se le istruzioni da eseguire fossero state poste appena dopo l'ultimo blocco `catch`, non sarebbero state eseguite.

Per il momento non sarà necessario utilizzare blocchi `finally`, questa breve descrizione è stata inclusa in questo capitolo per completezza.

13.3.6 Rilanciare un'eccezione (opzionale)

È lecito lanciare un'eccezione all'interno di un blocco `catch`. In alcuni rari casi, infatti, può presentarsi la necessità di catturare un'eccezione e di dover decidere, in base alla stringa restituita dal metodo `getMessage` o ad altri criteri, se rilanciare la stessa eccezione o una differente, in modo che venga gestita da altri blocchi `catch` più esterni.

CASO DI STUDIO
UNA CALCOLATRICE TESTUALE

Si supponga che sia stata commissionato un programma che permetta di eseguire operazioni matematiche, come una comune calcolatrice. Il programma deve eseguire correttamente somme, sottrazioni, moltiplicazioni e divisioni. Per la prima bozza del programma non sarà utilizzata un'interfaccia grafica, ma l'input e l'output saranno gestiti tramite un'interfaccia utente testuale, come nei precedenti programmi di questo capitolo.

L'interfaccia utente deve essere definita in maniera precisa. Si chiede all'utente di inserire un'operazione, uno spazio e un numero, il tutto su una stessa riga, come nell'esempio seguente:

```
+ 3.4
```

Ogni spazio bianco aggiuntivo posto prima o dopo l'operazione o il numero è opzionale. Mentre l'utente inserisce altre operazioni e numeri, il programma tiene traccia dei risultati delle operazioni eseguite fino a quel momento. Questi risultati sono paragonabili a quelli visualizzati da una normale calcolatrice e non sono altro che il risultato cumulativo delle operazioni. Sullo schermo può essere visualizzato per esempio

```
risultato aggiornato = 3.4
```

L'utente può sommare, sottrarre, moltiplicare e dividere usando istruzioni come:

```
* 3
```

Per esempio, si potrebbe avere la seguente sequenza d'interazioni tra utente e programma:

```
risultato = 0
* 80
risultato + 80 = 80
risultato aggiornato = 80
- 2
risultato - 2 = 78
risultato aggiornato = 78
f
```

Nell'interazione sopra riportata, l'input dell'utente è quello bordato dal rettangolo in grigio. Il programma presuppone che il "risultato" iniziale sia 0 e visualizza i dati inseriti, seguiti dal risultato di ogni computazione. Per terminare una sequenza di operazioni l'utente deve inserire la lettera `F`, maiuscola o minuscola.

Una volta che l'interfaccia proposta è stata approvata, si può iniziare a progettare una classe per la calcolatrice. Inizialmente verrà inserito nella classe un metodo `main` che svolge tutte le operazioni richieste, rispettando al tempo stesso le specifiche stabilite per l'interfaccia utente. Più avanti si potrebbe studiare un'interfaccia più elaborata e si potrebbe rendere la calcolatrice un po' più potente. Infine, si potrebbe aggiungere un'interfaccia grafica a finestre.

La variabile di istanza privata `risultato` tiene traccia del risultato corrente. Il programma somma, sottrae, moltiplica o divide il risultato corrente e il numero inserito. Per esempio, se l'utente inserisse

```
- 9.5
```

e il valore attuale di `risultato` fosse 80, il programma modificherebbe il valore di `risultato` in 70.5.

Nella classe dovrebbero essere presenti almeno i seguenti metodi:

- Un metodo `reset` che riporti a zero il valore di `risultato`.
- Un metodo `valuta` che calcoli il risultato di un'operazione.
- Un metodo `eseguiCalcoli` che esegua una serie di operazioni.
- Un metodo *get* `getRisultato` che restituisca il valore corrente della variabile di istanza `risultato`.
- Un metodo *set* `setRisultato` che permetta di modificare il valore della variabile di istanza `risultato`.

La definizione dei metodi `reset`, `getRisultato` e `setRisultato` è immediata, mentre i metodi `valuta` ed `eseguiCalcoli` richiedono un minimo di ragionamento. Per semplificare la prima bozza del codice, si presuppone che tutto andrà per il verso giusto, cioè si presuppone che non saranno riscontrate situazioni anomale che potrebbero generare eccezioni. Se, durante la stesura della prima bozza, si individuano potenziali situazioni anomale, potranno essere indicate nei commenti, rimandando la loro gestione a un secondo tempo, dopo aver completato il corpo principale della classe.

Per rendere più versatile la classe, il metodo `valuta` restituisce il risultato di un'operazione, anziché aggiornare direttamente la variabile `risultato`. Il metodo sarà quindi definito come segue:

```
/**
Restituisce n1 op n2,
op deve essere uno tra i seguenti operatori '+', '-', '*' e '/'.
*/
public double valuta(char op, double n1, double n2)
```

Il grosso del lavoro della calcolatrice è portato a termine dal metodo `eseguiCalcoli`, specificato come:

```
/**
Interagisce con l'utente per eseguire una serie
di operazioni e aggiornare la variabile di istanza risultato.
*/
public void eseguiCalcoli()
```

Per prima cosa verrà sviluppato proprio il metodo `eseguiCalcoli`. Il metodo deve ripetere in continuazione la seguente sequenza di operazioni finché l'utente non inserisce un valore sentinella, per esempio la lettera F:

```
char prossimaOp = (tastiera.next()).charAt(0);
double prossimoNumero = tastiera.nextDouble();
risultato = valuta(prossimaOp, risultato, prossimoNumero);
```

Dove `tastiera.next()` legge l'operatore come una stringa e `charAt(0)` lo restituisce come carattere. Le variabili `prossimaOp` e `prossimoNumero` sono locali, mentre `risultato` è la variabile di istanza introdotta in precedenza.
Il codice individuato sarà incluso in un ciclo all'interno di `eseguiCalcoli` nel seguente modo:

```
boolean fatto = false;
while (!fatto) {
    char prossimaOp = (tastiera.next()).charAt(0);
    if ((prossimaOp == 'f') || (prossimaOp == 'F'))
        fatto = true;
    else {
        double prossimoNumero = tastiera.nextDouble();
        risultato = valuta(prossimaOp, risultato,
                           prossimoNumero);
        System.out.println("risultato " + prossimaOp +
                           " " + prossimoNumero + " = " +
                           risultato);
        System.out.println("risultato aggiornato = "  +
                           risultato);
    }
}
```

Si consideri ora il metodo `valuta`. La soluzione più immediata consiste nell'utilizzare una clausola `switch` come la seguente:

```
switch (op) {
    case '+':
        risposta = n1 + n2;
        break;
    case '-':
        risposta = n1 - n2;
        break;
    case '*':
        risposta = n1 * n2;
        break;
    case '/':
        risposta = n1 / n2; //Gestire le divisione per zero
        break;
    default: //Gestire caratteri non validi
}
return risposta;
```

Nei primi test del codice si possono tranquillamente ignorare i commenti riguardanti le situazioni anomale, ma per accorciare questo caso di studio saranno fin da subito lanciate delle eccezioni. La loro gestione non sarà comunque affrontata ora. Si supponga di inserire il seguente frammento di codice nel precedente caso per la divisione:

```
if (n2 == 0.0)
    throw new DivisionePerZeroException();
```

Questo approccio è concettualmente corretto, ma c'è un problema: i numeri coinvolti sono di tipo `double`. I numeri in virgola mobile, come appunto i `double`, rappresentano solo quantità approssimate, quindi, come si è già detto nel Capitolo 3, non si dovrebbe usare un == per verificare le loro uguaglianze esatte. Il valore di `n2` potrebbe essere così vicino a 0 che, utilizzandolo come denominatore, potrebbe avere le stesse conseguenze di una divisione per 0, anche se un test direbbe comunque che è diverso da 0. Si dovrebbe lanciare un'eccezione quando il denominatore risulta molto vicino allo 0. Ma come andrebbe definito "molto vicino allo 0"? Si potrebbe, per esempio, considerare "molto vicina allo zero" una quantità inferiore a un decimillesimo. Ma poi ci si potrebbe rendere conto che questo valore non è ottimale. La migliore scelta consiste, quindi, nell'utilizzare una variabile di istanza denominata `precisione`. Per il momento si presuppone che la definizione di `precisione` sia data da:

```
private double precisione = 0.0001;
```

Di conseguenza, il caso per la divisione diventa:

```
case '/':
    if ((-precisione < n2) && (n2 < precisione))
        throw new DivisionePerZeroException();
    risposta = n1 / n2;
    break;
```

Cosa succede quando l'utente inserisce per l'operazione caratteri diversi da +, -, * o /? Si potrebbe lanciare un'altra eccezione nel caso di default della clausola `switch`:

```
default:
    throw new OpSconosciutaException(op);
```

La classe `DivisionePerZeroException` è stata definita nel Listato 13.5. La classe `OpSconosciutaException` è una nuova eccezione definita nel Listato 13.10. Si sottolinea che, quando l'utente inserisce un operatore sconosciuto, si vuole produrre un messaggio d'errore che includa anche il carattere errato. Quindi, la nuova eccezione deve avere un costruttore che accetta l'operatore come un argomento di tipo `char`.

L'intestazione del metodo `eseguiCalcoli` deve includere una clausola `throws` per le eccezioni `OpSconosciutaException` e `DivisionePerZeroException`, anche se il corpo di `eseguiCalcoli` non include nessun `throw`. La clausola `throws` è indispensabile perché `eseguiCalcoli` chiama il metodo `valuta` il quale può lanciare una `OpSconosciutaException` o una `DivisionePerZeroException`.

A questo punto, la maggior parte del programma è stata scritta, tranne la parte riguardante la gestione delle eccezioni. La versione preliminare del programma è riportata nel Listato 13.11. Ora si può eseguire il collaudo e il debugging della classe prima di scrivere il codice per la gestione delle eccezioni. Finché l'utente non inserirà un operatore sconosciuto o non tenterà di eseguire una divisione per 0, questa versione del programma funzionerà perfettamente.

MyLab

LISTATO 13.10 La classe `OpSconosciutaException`.

```
public class OpSconosciutaException extends Exception {

    public OpSconosciutaException() {
        super("OpSconosciutaException");
    }

    public OpSconosciutaException(char op) {
        super(op + " e' un operatore sconosciuto.");
    }

    public OpSconosciutaException(String messaggio) {
        super(messaggio);
    }
}
```

MyLab

LISTATO 13.11 Il caso standard.

Questa versione del programma non gestisce le eccezioni ed è quindi incompleta. In ogni caso, può essere eseguita e utilizzata per test e debug.

```
import java.util.Scanner;

/**
VERSIONE PRELIMINARE senza gestione delle eccezioni.
Semplice calcolatrice testuale. La classe può anche
essere utilizzata per creare altri programmi simili.
*/
public class CalcolatricePreliminare {
    private double risultato;
    private double precisione = 0.0001;
    //Numeri così prossimi a zero sono considerati pari a zero.
```

```
public CalcolatricePreliminare() {
    risultato = 0;
}

public void reset() {
    risultato = 0;
}

public void setRisultato(double nuovoRisultato) {
    risultato = nuovoRisultato;
}

public double getRisultato() {
    return risultato;
}

/**
Restituisce n1 op n2, ammesso che op sia +, -, * o /.
Ogni altro valore di op lancia una OpSconosciutaException.
*/
public double valuta(char op, double n1, double n2)
        throws DivisionePerZeroException, OpSconosciutaException {
    double risposta;
    switch (op) {
        case '+':
            risposta = n1 + n2;
            break;
        case '-':
            risposta = n1 - n2;
            break;
        case '*':
            risposta = n1 * n2;
            break;
        case '/':
            if ((-precisione < n2) && (n2 < precisione))
                throw new DivisionePerZeroException();
            risposta = n1 / n2;
            break;
        default:
            throw new OpSconosciutaException(op);
    }
    return risposta;
}

public void eseguiCalcoli() throws DivisionePerZeroException,
                                   OpSconosciutaException {
    Scanner tastiera = new Scanner(System.in);
    boolean fatto = false;
    risultato = 0;
    System.out.println("risultato = " + risultato);
```

`reset`, `setRisultato` e `getRisultato` non sono utilizzati in questo programma, ma potrebbero essere richiesti da altre applicazioni che usano questa classe.

```
        while (!fatto) {
            char prossimaOp = (tastiera.next()).charAt(0);
            if ((prossimaOp == 'f') || (prossimaOp == 'F'))
                fatto = true;
            else {
                double prossimoNumero = tastiera.nextDouble();
                risultato = valuta(prossimaOp, risultato, prossimoNumero);
                System.out.println("risultato " + prossimaOp + " " +
                                   prossimoNumero + " = " + risultato);
                System.out.println("risultato aggiornato = " + risultato);
            }
        }
    }

    public static void main(String[] args)
        throws DivisionePerZeroException, OpSconosciutaException {

        CalcolatricePreliminare calc = new CalcolatricePreliminare();

        System.out.println("Calcolatrice attivata.");
        System.out.print("Formato di ogni riga: ");
        System.out.println("operatore spazio numero");
        System.out.println("Per esempio: + 3");
        System.out.println("Per terminare inserire la lettera 'f'.");
        calc.eseguiCalcoli();

        System.out.println("Il risultato finale e' " + calc.getRisultato());
        System.out.println("Fine del programma.");
    }
}
```

La definizione del metodo `main` sarà modificata prima della fine di questo caso di studio.

Esempio di output

```
Calcolatrice attivata.
Formato di ogni riga: operatore spazio numero
Per esempio: + 3
Per terminare inserire la lettera 'f'.
risultato = 0.0
+ 4
risultato + 4.0 = 4.0
risultato aggiornato = 4.0
* 2
risultato * 2.0 = 8.0
risultato aggiornato = 8.0
f
Il risultato finale e' 8.0
Fine del programma.
```

Dopo aver testato la versione preliminare del programma, si può aggiungere la gestione delle eccezioni. L'eccezione più rilevante è `OpSconosciutaException` di cui è già stata fornita la definizione nel Listato 13.10. Nella versione preliminare della calcolatrice

fornita nel Listato 13.11, tale eccezione viene lanciata dal metodo `valuta` e compare come clausola `throws`. La sua gestione, però, non è stata ancora affrontata.

Il metodo `valuta` è invocato dal metodo `eseguiCalcoli`, che è a sua volta invocato dal metodo `main`. Vi sono tre approcci possibili per gestire l'eccezione.

- Catturare l'eccezione nel metodo `valuta`.
- Dichiarare l'eccezione `OpSconosciutaException` in una clausola `throws` del metodo `valuta` e catturarla nel metodo `eseguiCalcoli`.
- Dichiarare l'eccezione `OpSconosciutaException` in una clausola `throws` di entrambi i metodi `valuta` ed `eseguiCalcoli` e catturarla poi nel metodo `main`.

La scelta di quale approccio utilizzare dipenderà da cosa si vuole che accada quando viene lanciata un'eccezione. Si dovrebbe scegliere una delle prime due alternative se si vuole chiedere all'utente di reinserire l'operatore. Si dovrebbe invece usare l'ultima opzione se si vuole far ripartire l'intero processo di calcolo. Si supponga di aver scelto il terzo caso. Si inseriscono, quindi, i blocchi `try-catch` nel `main`. Questa decisione comporta la riscrittura del metodo `main` come indicato nel Listato 13.12. Tale revisione ha portato alla creazione di due nuovi metodi, `gestisciOpSconosciutaException` e `gestisciDivisionePerZeroException`. Tutto quello che resta da fare è definire questi due metodi per gestire correttamente le eccezioni.

Osservando il blocco `catch` nel metodo `main`, si nota che quando viene lanciata una `OpSconosciutaException`, essa viene gestita dal metodo `gestisciOpSconosciutaException`. Questo metodo è stato progettato in modo da dare all'utente una seconda opportunità di eseguire dei calcoli, ripartendo dall'inizio. Se l'utente inserisce un operatore sconosciuto anche durante la seconda opportunità, viene lanciata un'altra `OpSconosciutaException`, ma questa volta viene catturata nel metodo `gestisciOpSconosciutaException` e il programma termina. Esistono, ovviamente, altri modi accettabili di gestire una `OpSconosciutaException`. Il codice per gestire questo caso è inserito nel blocco `catch` del metodo `gestisciOpSconosciutaException` nel Listato 13.12.

Se l'utente prova a eseguire una divisione per 0, il programma semplicemente termina. Si potrebbe anche optare per qualcosa di più elaborato nelle versioni future del programma. Di conseguenza, il metodo `gestisciDivisionePerZeroException` è piuttosto semplice.

LISTATO 13.12 La calcolatrice testuale completa.

MyLab

```
import java.util.Scanner;
/**
Semplice calcolatrice testuale. La classe può anche
essere utilizzata per creare altri programmi simili.
*/
public class Calcolatrice {
    private double risultato;
    private double precisione = 0.0001;
    //Numeri così prossimi a zero sono considerati pari a zero.

    public Calcolatrice() {
        risultato = 0;
    }
```

```
public void gestisciDivisionePerZeroException(
            DivisionePerZeroException e) {
    System.out.println("Divisione per zero.");
    System.out.println("Programma terminato.");
    System.exit(0);
}

public void gestisciOpSconosciutaException(OpSconosciutaException e) {
    System.out.println(e.getMessage());
    System.out.println("Riprova dall'inizio:");

    try {
        System.out.print("Formato di ogni riga: ");
        System.out.println("operatore spazio numero");
        System.out.println("Per esempio: + 3");
        System.out.println("Per terminare inserire la lettera 'f'.");
        eseguiCalcoli();
    } catch(OpSconosciutaException e2) {
        System.out.println(e2.getMessage());
        System.out.println("Riprova in un secondo momento.");
        System.out.println("Fine del programma.");
        System.exit(0);

    } catch(DivisionePerZeroException e3) {
        gestisciDivisionePerZeroException(e3);
    }
}
```

La prima `OpSconosciutaException` offre all'utente un'altra possibilità.

Questo blocco cattura una `OpSconosciutaException` se viene lanciata una seconda volta.

<I metodi `reset`, `setRisultato`, `getRisultato`, `valuta` ed `eseguiCalcoli` sono gli stessi del Listato 13.11.>

```
public static void main(String[] args) {
    Calcolatrice calc = new Calcolatrice();

    try {
        System.out.println("Calcolatrice attivata.");
        System.out.print("Formato di ogni riga: ");
        System.out.println("operatore spazio numero");
        System.out.println("Per esempio: + 3");
        System.out.println("Per terminare inserire la lettera 'f'.");
        calc.eseguiCalcoli();

    } catch(OpSconosciutaException e) {
        calc.gestisciOpSconosciutaException(e);

    } catch(DivisionePerZeroException e) {
        calc.gestisciDivisionePerZeroException(e);
    }
```

```
        System.out.println("Il risultato finale e' " + calc.getRisultato());
        System.out.println("Fine del programma.");
    }
}
```

Esempio di output

```
Calcolatrice attivata.
Formato di ogni riga: operatore spazio numero
Per esempio: + 3
Per terminare inserire la lettera 'f'.
risultato = 0.0
+ 80
risultato + 80.0 = 80.0
risultato aggiornato = 80.0
- 2
risultato - 2.0 = 78.0
risultato aggiornato = 78.0
% 4
% e' un operatore sconosciuto.
Riprova dall'inizio:
Formato di ogni riga: operatore spazio numero
Per esempio: + 3
Per terminare inserire la lettera 'f'.
risultato = 0.0
+ 80
risultato + 80.0 = 80.0
risultato aggiornato = 80.0
- 2
risultato - 2.0 = 78.0
risultato aggiornato = 78.0
* 0.04
risultato * 0.04 = 3.12
risultato aggiornato = 3.12
f
Il risultato finale e' 3.12
Fine del programma.
```

La documentazione dovrebbe descrivere le possibili eccezioni

Quando si utilizzano classi e metodi scritti da un altro programmatore, sarebbe utile poter disporre della documentazione che descrive le possibili eccezioni. Una documentazione di questo tipo potrebbe, per esempio, dare qualche indicazione sul modo in cui gestire le eccezioni. Questa considerazione dovrebbe essere sempre ricordata durante la scrittura dei commenti `javadoc` e di qualsiasi altra documentazione riguardante il proprio codice.

13.4 Riepilogo

- Un'eccezione è un oggetto il cui tipo è derivato dalla classe `Exception`. Le classi che ereditano dalla classe `Error` non sono eccezioni, ma si comportano in modo simile.
- La gestione delle eccezioni permette di progettare e codificare separatamente il normale comportamento del programma e la gestione delle situazioni anomale.
- Java fornisce delle classi di eccezioni predefinite. Si possono comunque definire delle classi di eccezioni personalizzate.
- Java ha due tipi di eccezioni: controllate (*checked*) e non controllate (*unchecked* o *run-time*). Un metodo che lancia un'eccezione controllata deve gestirla o dichiararla nella propria intestazione tramite una clausola `throws`. Le eccezioni controllate devono, comunque, essere catturate, altrimenti l'esecuzione del programma termina. Le eccezioni non controllate non richiedono di essere catturate o dichiarate in una clausola `throws` e di solito non lo sono. Le eccezioni non controllate sono istanze di classi derivate dalla classe `RuntimeException`. Tutte le altre eccezioni sono controllate.
- Alcune istruzioni Java possono lanciare eccezioni. Di conseguenza, alcuni dei metodi predefiniti di Java possono lanciare eccezioni. Quando un programmatore scrive un proprio metodo, può decidere di lanciare un'eccezione utilizzando l'istruzione `throw`.
- Quando un metodo può lanciare un'eccezione, ma non la cattura, il tipo di eccezione deve essere dichiarato in una clausola `throws` nell'intestazione del metodo.
- Un'eccezione è catturata da un blocco `catch`.
- Un blocco `try` è seguito da uno o più blocchi `catch`. Se i blocchi `catch` sono più d'uno, quelli per eccezioni più specifiche devono sempre precedere quelli per eccezioni più generiche.
- Ogni eccezione ha un metodo `getMessage` che può essere utilizzato per recuperare una descrizione dell'eccezione catturata.
- Non si deve abusare delle eccezioni.

13.5 Esercizi

1. Scrivere un programma che permetta agli studenti di pianificare appuntamenti per le ore 13, 14, 15, 16, 17 o 18. Si utilizzi un array di sei stringhe per memorizzare le descrizioni degli appuntamenti associati ai sei orari. Scrivere un ciclo che si ripete finché l'array ha ancora spazio libero. In un blocco `try`, chiedere all'utente di inserire un'ora e una descrizione. Se l'ora specificata è libera, inserire la descrizione nell'array. In caso contrario, lanciare un'eccezione `OraOccupataException`. Se, invece, l'ora specificata non è nell'intervallo 13 – 18, lanciare un'eccezione `OraNonContemplataException`. Usare un blocco `catch` per ogni tipo di eccezione.

2. Scrivere un programma che permetta all'utente di calcolare il resto di una divisione fra due valori interi. Il resto della divisione di `x/y` è `x%y`. Catturare qualsiasi tipo di eccezione che può essere lanciata e fare in modo che l'utente possa inserire nuovi valori.
3. Scrivere una classe di eccezione per indicare che un'ora inserita da un utente non è valida. L'ora deve essere nel formato *ore:minuti* e seguita da "am" o "pm".
4. Derivare delle nuove classi di eccezioni da quella realizzata nell'esercizio precedente. Ogni nuova classe deve indicare il tipo d'errore. Per esempio, `OraNonValidaException` potrebbe essere usata per indicare che il valore inserito per l'ora non è un intero compreso tra 1 e 12.
5. Scrivere una classe `OraDelGiorno` che usi le classi di eccezioni definite nell'esercizio precedente. Implementare un metodo `setOra(stringaOra)` che cambia l'ora se `stringaOra` corrisponde a un'ora valida. Altrimenti lancia un'eccezione del tipo appropriato.
6. Scrivere un frammento di codice che legga una stringa dalla tastiera e la usi per impostare la variabile `miaOra` di tipo `OraDelGiorno`, definita nell'esercizio precedente. Usare dei blocchi `try-catch` per garantire che `miaOra` sia valida.
7. Creare una classe `CartaBrano` che rappresenta una carta prepagata per l'acquisto di brani musicali online. Deve avere le seguenti variabili di istanza private:
   - `brani` – il numero di brani sulla carta;
   - `attivata` – vero se la carta è stata attivata.

   Inoltre deve avere i seguenti metodi:
   - `CartaBrano(n)` – un costruttore per una carta prepagata con n brani musicali;
   - `attiva` – attiva la carta prepagata e lancia un'eccezione se la carta è già stata attivata;
   - `compraBrano` – registra l'acquisto di un brano musicale decrementando il numero di brani musicali ancora acquistabili con questa carta e lancia un'eccezione se la carta è esaurita o se non è attiva;
   - `braniRimanenti` – restituisce il numero di brani musicali che si possono ancora acquistare con questa carta.
8. Creare una classe `Razionale` che rappresenti un numero razionale. Deve avere delle variabili di istanza private per rappresentare:
   - il numeratore (un intero);
   - il denominatore (un intero);

   e i seguenti metodi:
   - `Razionale(numeratore, denominatore)` – un costruttore per un numero razionale;
   - i metodi *get* `getNumeratore` e `getDenominatore` e i metodi *set* `setNumeratore` e `setDenominatore` per il numeratore e il denominatore.

   Si deve utilizzare un'eccezione per garantire che il denominatore non sia mai uguale a 0.

9. Rivedere la classe `Razionale` descritta nell'esercizio precedente, in modo da usare un'asserzione invece che un'eccezione per garantire che il denominatore non sia mai uguale a 0.
10. Si supponga di voler creare un oggetto per contare il numero di persone in una stanza. Tale numero non può mai essere negativo. Creare una classe `ContatoreStanza` che abbia tre metodi pubblici:
    - `aggiungiPersona` – aggiunge una persona alla stanza;
    - `rimuoviPersona` – rimuove una persona dalla stanza;
    - `getContatore` – restituisce il numero di persone nella stanza.

    Qualora l'invocazione al metodo `rimuoviPersona` rendesse il numero di persone presenti nella stanza inferiore a 0, si deve lanciare un'eccezione `ContatoreNegativoException`.
11. Rivedere la classe `ContatoreStanza` descritta nell'esercizio precedente in modo da usare un'asserzione al posto di un'eccezione per evitare che il numero di persone nella stanza diventi negativo.
12. Mostrare le modifiche necessarie per aggiungere l'operazione di elevamento a potenza alla classe `Calcolatrice` nel Listato 13.12. Usare il simbolo ^ per rappresentare l'operatore esponenziale e il metodo `Math.pow` per eseguire i calcoli.
13. Scrivere una classe `CronometroGiri` che possa essere utilizzata per cronometrare i giri di una corsa. La classe dovrebbe avere le seguenti variabili di istanza private:
    - `inEsecuzione` – un valore booleano che indica se il cronometro è in esecuzione;
    - `tempoDiPartenza` – il momento in cui il cronometro è stato fatto partire;
    - `tempoDiPartenzaGiroCorrente` – il valore del cronometro all'inizio del giro corrente;
    - `durataUltimoGiro` – il tempo impiegato per effettuare l'ultimo giro; con ultimo si intende rispetto al giro corrente;
    - `tempoTotale` – il tempo totale trascorso dall'inizio della gara fino all'ultimo giro completato;
    - `giriCompletati` – il numero di giri completati finora;
    - `numeroGiri` – il numero totale di giri previsti per questa gara.

    La classe dovrebbe avere i seguenti metodi:
    - `CronometroGiri(n)` – un costruttore per una gara di n giri;
    - `parti` – fa partire il cronometro; lancia un'eccezione se la gara è già iniziata;
    - `marcaGiro` – marca la fine del giro corrente e l'inizio di un nuovo giro; lancia un'eccezione se la gara è finita;
    - `getDurataUltimoGiro` – restituisce il tempo dell'ultimo giro; lancia un'eccezione se il primo giro non è ancora stato completato;

- `getTempoTotale` – restituisce il tempo totale trascorso dall'inizio della gara fino all'ultimo giro completato; lancia un'eccezione se il primo giro non è ancora stato completato;
- `getGiriRimanenti` – restituisce il numero di giri ancora da completare, compreso il giro corrente.

Tutti i tempi devono essere espressi in secondi.

Per ottenere l'ora corrente sotto forma di millisecondi trascorsi da una certa data di riferimento, è sufficiente invocare il metodo:

```
Calendar.getInstance().getTimeInMillis()
```

Questa invocazione restituisce un valore di tipo `long`. Calcolando la differenza tra i valori restituiti da due invocazioni al metodo effettuate in momenti diversi, si ottiene il tempo trascorso, in millisecondi, tra la prima invocazione e la successiva. La classe `Calendar` è definita nel package `java.util`.

13.6 Progetti

1. Definire una classe di eccezione chiamata `MessaggioTroppoLungoException` che abbia due costruttori: uno di default che assegna al messaggio la stringa "Messaggio troppo lungo" e l'altro che accetta in ingresso un argomento di tipo `String`. Usare tale classe in un programma che chiede all'utente di inserire una riga di testo composta da non più di 20 caratteri. Se l'utente inserisce un numero accettabile di caratteri, il programma deve visualizzare il messaggio "Hai inserito *x* caratteri, che è un numero di caratteri accettabile" (al posto della lettera *x* deve essere indicato l'effettivo numero di caratteri). In caso contrario, deve essere lanciata e catturata un'eccezione del tipo `MessaggioTroppoLungoException`. In ogni caso, il programma deve ripetutamente chiedere all'utente se vuole inserire un'altra riga di testo o se desidera terminare l'esecuzione.
2. Scrivere un programma che converta un orario dalla notazione "24 ore" a quella "12 ore". Di seguito viene mostrato un esempio d'interazione tra il programma e l'utente:

```
Inserire un orario nella notazione 24 ore:
13:07
L'equivalente in notazione 12 ore e'
1:07 PM
Di nuovo? (s/n)
s
Inserire un orario nella notazione 24 ore:
10:15
L'equivalente in notazione 12 ore e'
10:15 AM
Di nuovo? (s/n)
```

```
Inserire un orario nella notazione 24 ore:
10:65
L'orario 10:65 non esiste.
Riprova.
Inserire un orario nella notazione 24 ore:
16:05
L'equivalente in notazione 12 ore e'
4:05 PM
Di nuovo? (s/n)
n
Fine del programma.
```

Definire un'eccezione chiamata `FormatoOrarioException`. Se l'utente inserisce un orario non valido, come 10:65, o addirittura senza senso, come 8%*68, il programma deve lanciare e gestire un'eccezione di tipo `FormatoOrarioException`.

3. Scrivere un programma che usi la classe `Calcolatrice` del Listato 13.12 per creare una calcolatrice più potente. Questa nuova calcolatrice deve fornire all'utente la possibilità di salvare un risultato in memoria e di poterlo riutilizzare in seguito. I comandi che la calcolatrice deve accettare sono i seguenti:

   - `f` (fine) per terminare;
   - `c` (cancella) per cancellare il risultato. Reimposta `risultato` a 0;
   - `s` (salva) per salvare in memoria l'attuale valore di `risultato`;
   - `r` (richiama) per richiamare il valore dalla memoria; visualizza il valore in memoria, ma non modifica il valore della variabile `risultato`.

   Deve essere definita una classe derivata da `Calcolatrice` che abbia un'altra variabile di istanza per rappresentare la memoria, un nuovo metodo `main` che esegua la versione migliorata della calcolatrice, una ridefinizione del metodo `gestisciOpSconosciutaException` e tutto ciò che sarà necessario ridefinire o scrivere da zero. Di seguito è riportato un esempio di una possibile esecuzione della nuova calcolatrice. Il programma non deve produrre un output identico a quello riportato, ma deve essere simile e sufficientemente chiaro.

```
Calcolatrice attivata.
risultato = 0.0
+ 4
risultato + 4.0 = 4.0
risultato aggiornato = 4.0
/ 2
risultato / 2.0 = 2.0
risultato aggiornato = 2.0
s
risultato salvato in memoria
c
risultato = 0.0
+ 99
risultato + 99.0 = 99.0
risultato aggiornato = 99.0
```

```
/ 3
risultato / 3.0 = 33.0
risultato aggiornato = 33.0
r
valore richiamato dalla memoria = 2.0
risultato = 33.0
+ 2
risultato + 2.0 = 35.0
risultato aggiornato = 35.0
f
Fine del programma.
```

4. Scrivere un programma che converta le date dal formato numerico giorno-mese al formato testuale giorno-mese. Per esempio, inserendo 31/1 o 31/01 si deve ottenere in output 31 Gennaio. L'interazione con l'utente deve essere simile a quella mostrata nel Progetto 2. Devono essere definite due classi di eccezioni, una chiamata `MeseException` e l'altra chiamata `GiornoException`. Se l'utente inserisce un numero di mese errato (dunque non compreso tra 1 e 12) il programma deve lanciare e catturare un'eccezione di tipo `MeseException`. In modo analogo, se l'utente inserisce un numero di giorno errato (dunque non compreso tra 1 e 31, 1 e 30 o 1 e 28 a seconda del mese) il programma deve lanciare e catturare un'eccezione di tipo `GiornoException`. Per semplificare le cose si supponga che Febbraio abbia sempre 28 giorni.

5. Modificare il programma *driver* del Progetto 4 nel Capitolo 10 in modo da usare tre classi di eccezioni chiamate `CilindroException`, `CaricoException` e `TrainoException`. Il numero di cilindri deve essere un intero tra 1 e 12, la capacità di carico deve essere un numero con cifre decimali tra 1 e 10, la capacità di traino deve essere un numero con cifre decimali tra 1 e 20. Qualsiasi input che non rispetti queste regole deve causare il lancio e la conseguente cattura dell'eccezione appropriata. Ovviamente devono essere definite anche le tre classi di eccezioni `CilindroException`, `CaricoException` e `TrainoException`.

6. Definire una classe di eccezione chiamata `DimensioneException` da usare nel programma *driver* del Progetto 3 del Capitolo 11. Modificare il programma in modo che lanci e catturi un'eccezione di tipo `DimensioneException` se l'utente inserisce valori minori o uguali a 0 per una dimensione.

MyLab

Video 13.3 Definire una classe di eccezione propria

7. Scrivere un programma per inserire in un array le informazioni dei dipendenti, compreso il codice fiscale e il salario. Il numero massimo di dipendenti è 100, ma il programma deve funzionare anche per un numero di dipendenti minore di 100. Il programma deve usare due classi di eccezioni, una, chiamata `LunghezzaCFException`, da utilizzare quando il codice fiscale inserito è composto da un numero di caratteri diverso da 16; l'altra, chiamata `FormatoCFException`, da utilizzare quando uno o più dei caratteri del codice fiscale non rispetta il formato standard (i primi sei devono essere lettere, seguiti da due cifre, una lettera, due cifre, una lettera, tre cifre e un'ultima lettera). Quando viene lanciata un'eccezione, il programma deve: visualizzare all'utente il dato da lui inserito che non risulta corretto, indicare il motivo per cui è errato e chiedere di reinserire il dato rifiutato. Dopo che tutti i dati sono stati inseriti, il programma deve visualizzare le informazioni di tutti i dipendenti,

segnalando, anche, se lo stipendio del dipendente è superiore o inferiore alla media. Le classi `Dipendente`, `LunghezzaCFException` e `FormatoCFException` devono essere definite. La classe `Dipendente` deve derivare dalla classe `Persona` presentata nel Listato 10.1 del Capitolo 10. Tra le altre cose, la classe `Dipendente` deve avere metodi di input e output, così come costruttori, metodi *get* e metodi *set*. Ogni oggetto `Dipendente` deve tenere traccia del nome del dipendente, del salario, del codice fiscale e di ogni altra informazione che si ritenga appropriata.

8. Un metodo che restituisce un valore particolare come codice di errore può, a volte, creare problemi. Chi lo utilizza potrebbe ignorare il codice di errore o trattarlo come un valore valido. In casi come questo, è meglio generare un'eccezione che gestire gli errori tramite valori di ritorno particolari. La classe riportata di seguito gestisce il saldo di un conto corrente e sfrutta un valore di ritorno particolare come codice di errore.

```
public class ContoCorrente {
    private double saldo;

    public ContoCorrente() {
        saldo = 0;
    }

    public ContoCorrente(double depositioIniziale) {
        saldo = depositioIniziale;
    }

    public double getSaldo() {
        return saldo;
    }

    // Restituisce il nuovo saldo o -1 in caso di errore
    public double deposita(double somma) {
        if (somma > 0)
            saldo += somma;
        else
            return -1; // Codice che segnala un errore
        return saldo;
    }

    // Restituisce il nuovo saldo o -1 in caso di errore
    public double ritira(double somma) {
        if ((somma > saldo) || (somma < 0))
            return -1;
        else
            saldo -= somma;
        return saldo;
    }
}
```

Riscrivere la classe in modo che generi eccezioni appropriate anziché restituire -1 come codice di errore. Si scriva del codice di prova che cerchi di ritirare e depositare somme non valide e gestisca le eccezioni che vengono generate.

9. Si supponga di essere il responsabile del servizio clienti di un negozio. Per ogni chiamata ricevuta, viene registrato il nome del chiamante. Scrivere una classe `RichiesteServizio` che tiene traccia del nome dei chiamanti. La classe dovrebbe avere i seguenti metodi:

- `aggiungiNome(nome)` – aggiunge un nome alla lista dei nomi e lancia un'eccezione `BackUpServizioException` se non c'è spazio libero nella lista;
- `rimuoviNome(nome)` – rimuove un nome dalla lista e lancia un'eccezione `NoRichiestaServizioException` se il nome non è nella lista;
- `getNome(i)` – restituisce l'i-esimo nome della lista;
- `getNumero` – restituisce l'attuale numero di richieste di servizio.

Scrivere un programma che utilizza un oggetto di tipo `RichiesteServizio` per tenere traccia dei clienti che hanno chiamato. Definire un ciclo che, a ogni iterazione, tenta di aggiungere un nome, rimuovere un nome o stampare tutti i nomi. Utilizzare un array di dimensione 10 per rappresentare la lista dei nomi.

Stream e I/O da file

OBIETTIVI

- Descrivere il concetto di flusso di I/O.
- Spiegare la differenza tra file di testo e file binari.
- Salvare dati (oggetti compresi) in un file
- Leggere dati (oggetti compresi) da un file.

Con l'espressione I/O ci si riferisce all'input e all'output di un programma. L'input può essere ricevuto, per esempio, dalla tastiera o da un file. Analogamente, l'output può essere inviato a uno schermo o a un file. In questo capitolo si spiegherà come scrivere programmi che leggano e scrivano su file. In questo modo, i risultati potranno essere conservati anche dopo la fine dell'esecuzione del programma.

Prerequisiti

Per comprendere gli argomenti trattati in questo capitolo, sarà necessario conoscere i concetti base dell'ereditarietà, presentati nel Capitolo 10, e la gestione delle eccezioni, descritta nel Capitolo 13. Il Paragrafo 14.5 e il Caso di Studio nel Paragrafo 14.3 richiedono la conoscenza degli array (presentati nel Capitolo 6 e 9) e delle interfacce (Capitolo 11).

14.1 Introduzione ai flussi dati e all'I/O su file

In questo paragrafo si presenta un'introduzione generale al tema dell'I/O da e su file. In particolare, verrà spiegata la differenza tra file di testo e file binari. La sintassi Java per le istruzioni di I/O sarà invece argomento dei paragrafi successivi.

14.1.1 Il concetto di stream

L'utilizzo dei file è molto comune: essi vengono sfruttati, ad esempio, per salvare classi Java e programmi, musica, foto e video. Si possono anche utilizzare i file per immagazzinare i dati di input a un programma o per salvare i dati prodotti da un programma.

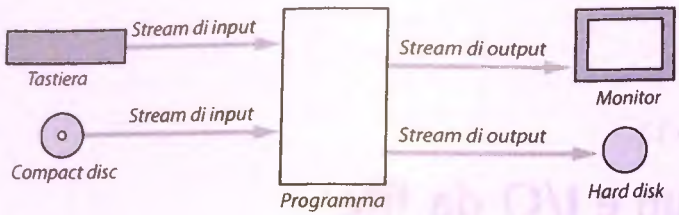


Figura 14.1 Stream di input e di output.

In Java, l'I/O da e su file, così come quelli, più semplici, da tastiera e su schermo, è gestito in termini di flussi di dati. Un **flusso di dati** (generalmente indicato con il termine inglese *stream*) può essere costituito da caratteri, numeri o generici byte. Se i dati fluiscono *nel programma*, lo stream è detto **stream di input**. Se, al contrario, i dati fluiscono *dal programma*, lo stream è detto **stream di output**. Per esempio, se uno stream di input è collegato alla tastiera, i dati fluiranno dalla tastiera al programma. Allo stesso modo, se uno stream di input è collegato a un file, il contenuto del file fluirà nel programma. La Figura 14.1 mostra alcuni esempi di stream.

In Java, gli stream sono realizzati come istanze di alcune classi speciali di tipo stream. Gli oggetti di tipo `Scanner`, utilizzati per leggere dati da tastiera, sono degli stream di input. L'oggetto `System.out` è un esempio di stream di output. In questo capitolo verranno discussi stream che consentiranno di collegare un programma a dei file, anziché a tastiera e schermo.

Stream

Uno stream è un oggetto che svolge una delle seguenti due funzioni alternative:

- trasferisce dati da un programma a una destinazione, come un file o lo schermo;
- trasferisce dati da una sorgente come un file o la tastiera a un programma.

14.1.2 Perché utilizzare l'I/O su file?

L'input da tastiera e l'output su schermo utilizzati finora gestiscono dati temporanei. Quando l'esecuzione di un programma termina, i dati inseriti tramite tastiera e quelli mostrati sullo schermo vengono persi. I file forniscono un mezzo per immagazzinarli in modo permanente. Il contenuto di un file viene preservato finché una persona o un programma non lo modificano esplicitamente.

Un file di input può essere utilizzato più e più volte da programmi diversi, senza che sia necessario riscrivere da capo i dati di input per ogni programma. Inoltre, i file costituiscono un modo comodo per gestire grandi quantità di dati. Quando un programma riceve in input dei dati da un file grande, ottiene una gran quantità di dati senza bisogno di un intervento diretto dell'utente.

14.1.3 File di testo e file binari

In qualunque tipo di file i dati sono immagazzinati per mezzo di cifre binarie (bit), cioè sotto forma di lunghe sequenze di 0 e 1. Tuttavia, in alcune situazioni si interpreta il contenuto di un file non come una sequenza di cifre binarie, ma come una sequenza di caratteri di testo. I file interpretati in questo modo, e per i quali esistono stream e metodi che gestiscono le corrispondenti sequenze binarie come sequenze di caratteri, sono detti **file di testo**. Tutti gli altri tipi di file sono detti **file binari**. Ognuno dei due tipi di file ha i propri stream e metodi per elaborarli.

I programmi Java sono salvati in file di testo. Al contrario, immagini e musica sono immagazzinati in file binari. Dato che i file di testo contengono sequenze di caratteri, solitamente vengono interpretati allo stesso modo su tutti i computer, pertanto possono essere spostati da un computer all'altro senza problemi o con poche difficoltà. Il contenuto dei file binari è generalmente basato su numeri. La struttura di alcuni tipi di file binari è standard, così che essi possano essere utilizzati su più piattaforme. Molti tipi di formati per la gestione di immagini e musica rientrano in questa categoria.

I programmi Java possono leggere e scrivere sia file di testo che file binari. La scrittura, così come la lettura, è simile nei due casi. Il tipo di file, tuttavia, determina quali classi debbano essere utilizzate per l'input e per l'output.

Il vantaggio principale dei file di testo è che è possibile crearli, visualizzarli e modificarli utilizzando un editor di testi (è ciò che si fa quando si scrive un programma in Java). Per un file binario, le operazioni di lettura e scrittura devono generalmente essere eseguite da un programma apposito. Alcuni file binari sono fatti per essere utilizzati sullo stesso tipo di computer e con lo stesso linguaggio di programmazione con i quali sono stati creati. Tuttavia, i file binari Java sono indipendenti dalla piattaforma: sarà possibile spostare i file da un computer all'altro e i programmi Java saranno ancora in grado di utilizzarli.

In un file di testo ogni carattere è rappresentato per mezzo di uno o due byte, a seconda che il sistema utilizzi la codifica ASCII o Unicode. Quando un programma scrive un valore in un file di testo, il numero di caratteri che vengono scritti è lo stesso che si avrebbe scrivendo lo stesso valore su schermo per mezzo del metodo `System.out.println`. Per esempio, la scrittura in un file di testo del numero 12345 comporta la scrittura di cinque caratteri nel file, come mostrato nella Figura 14.2. In generale, la scrittura di un numero intero comporta la scrittura di un numero di caratteri tra 1 e 11.

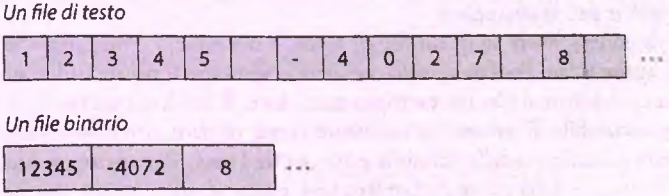


Figura 14.2 Un file di testo e uno binario contenenti gli stessi valori numerici.

I file binari immagazzinano tutti i valori dello stesso tipo primitivo nello stesso formato. Ogni valore è quindi salvato come sequenza dello stesso numero di byte. Per esempio, tutti i valori di tipo `int` occupano ognuno quattro byte, come mostrato nella Figura 14.2. Un programma Java interpreta questi byte in modo molto simile a quanto fa con i dati nella memoria principale. È per questo motivo che la gestione dei file binari è molto efficiente.

FAQ È meglio utilizzare un file di testo o uno binario?

I file di testo vanno utilizzati quando si vuole che possano essere creati, visualizzati e modificati per mezzo di un editor di testi. In tutti gli altri casi è opportuno utilizzare file binari, che solitamente occupano meno spazio.

Uso dei termini input e output

Il termine *input* significa che i dati vengono fatti fluire nel programma, non nel file. Analogamente, il termine *output* indica che i dati fluiscono dal programma, non dal file.

14.2 I/O con file di testo

In questo paragrafo vengono presentati i modi più comuni per gestire l'I/O con file di testo in Java.

14.2.1 Creare un file di testo

La classe `PrintWriter` nella Java Class Library definisce i metodi per creare file di testo e scrivere in essi. Questa è la classe da utilizzare preferibilmente per l'output su file di testo. La classe fa parte del package `java.io`, quindi sarà necessario importare esplicitamente la classe all'inizio del programma. Sarà poi necessario importare anche altre classi che verranno descritte successivamente.

Prima di poter scrivere su di un file di testo, è necessario collegarlo a uno stream di output, cioè **aprire** il file. Per fare questo, occorre conoscere il nome utilizzato dal sistema operativo per individuare il file, per esempio `out.txt`. È inoltre necessario dichiarare una variabile, detta **variabile di stream**, da utilizzare come riferimento allo stream che gestisce il file. In questo caso, il tipo della variabile è `PrintWriter`. Per aprire un file per l'output si invoca il costruttore della classe `PrintWriter` passandogli il nome del file come argomento. Poiché questa operazione può generare un'eccezione, le istruzioni devono essere incluse in un blocco `try`.

Le istruzioni che seguono effettuano l'apertura del file `out.txt` per l'output:

```
String nomeFile = "out.txt"; //Lo si potrebbe chiedere all'utente
PrintWriter outputStream = null;
try {
    outputStream = new PrintWriter(nomeFile);
} catch (FileNotFoundException e) {
    System.out.println("Errore nell'apertura del file " + nomeFile);
    System.exit(0);
}
```

Anche la classe `FileNotFoundException` deve essere importata dal package `java.io`.

Si noti che il nome del file (in questo caso, `out.txt`) è fornito sotto forma di un valore di tipo `String`. In generale, il nome del file sarà probabilmente letto da qualche parte e non inserito direttamente nel codice. Il nome del file viene passato come argomento al costruttore di `PrintWriter`. L'oggetto risultante è assegnato alla variabile `outputStream`.

Quando si collega un file a uno stream di output in questo modo, il programma parte sempre da un file vuoto. Se il file specificato esisteva già, il contenuto precedente andrà perso. Se invece il file non esisteva, ne verrà creato uno vuoto.

Poiché la chiamata al costruttore di `PrintWriter` può generare un'eccezione di tipo `FileNotFoundException`, essa deve essere inclusa in un blocco `try`. Le eventuali eccezioni sono gestite nel blocco `catch`. Anche se il costruttore dovesse generare un'eccezione di questo tipo, non è detto che essa sia dovuta al fatto che il file non è stato trovato (d'altronde, è previsto che se il file non esiste già ne venga creato uno nuovo). In tal caso, l'eccezione significherebbe che non è stato possibile creare il file, per esempio perché il nome specificato è già utilizzato come nome di una directory.

Una volta che il file è stato aperto (cioè una volta che è stato collegato a uno stream di output) è possibile scrivervi dei dati. Il metodo `println` della classe `PrintWriter` consente di scrivere dati in un file di testo esattamente nello stesso modo in cui il metodo `System.out.println` consente di scrivere sullo schermo. La classe `PrintWriter` ha anche un metodo `print` che si comporta esattamente come il metodo `System.out.print`, ad eccezione del fatto che in questo caso l'output è scritto su file.

Dopo che il file è stato aperto, ci si riferisce ad esso utilizzando sempre la variabile associata allo stream, e non il nome del file. La variabile `outputStream` si riferisce allo stream di output (cioè l'oggetto `PrintWriter`) appena creato, quindi è questo l'oggetto da utilizzare per invocare `println`. Si noti che la variabile `outputStream` è dichiarata fuori dal blocco `try`, così che essa è disponibile anche all'esterno del blocco.

Si supponga di voler scrivere più dati nel file. Le istruzioni che seguono saranno successive a quelle necessarie per l'apertura del file:

```
outputStream.println("Questa è la riga 1.");
outputStream.println("Ecco la riga 2.");
```

La classe `PrintWriter` non invia immediatamente l'output al file, ma aspetta di aver accumulato una quantità abbastanza grande di dati. Pertanto, l'output prodotto da una chiamata a `println` non viene inviato al file immediatamente, ma è salvato in un'area di memoria detta **buffer**, insieme all'output generato da invocazioni precedenti di `print` e `println`. Quando il buffer è pieno, il suo contenuto viene effettivamente scritto nel file. Quindi, l'output da più chiamate a `println` viene scritto nel file nello stesso momento. Questa tecnica è detta **buffering** e consente un'elaborazione più veloce dei file.

Una volta che l'intero file di testo è stato scritto, lo si disconnette dallo stream, cioè lo si chiude, per mezzo dell'istruzione

```
outputStream.close();
```

La chiusura dello stream fa sì che il sistema liberi qualunque risorsa utilizzata per connettere il file allo stream e svolga altre operazioni di pulizia. Se non si chiude esplicitamente uno stream, Java lo farà automaticamente al termine dell'esecuzione del programma. Tuttavia, è opportuno chiudere lo stream esplicitamente chiamando il metodo `close`. Infatti, si ricordi che quando si richiede la scrittura di dati in un file, questi potrebbero non essere scritti immediatamente. Chiudendo lo stream, tutti i dati ancora nel buffer vengono scritti nel file immediatamente. Se non si chiude lo stream e l'esecuzione del programma termina inaspettatamente, Java potrebbe non aver modo di chiuderlo e si potrebbero perdere dei dati. Quindi, prima si chiude uno stream, meno probabilità si avranno che accada ciò.

Se un programma scrive dati in un file e successivamente legge dallo stesso file, deve chiudere lo stream quando ha finito di scrivere e riaprire il file per la lettura (in realtà, Java offre una classe che consente di aprire un file sia in lettura che in scrittura; tale classe non sarà discussa in questo testo). Si noti che tutte le classi di tipo stream, come `PrintWriter`, hanno un metodo chiamato `close`.

Non è necessario che le chiamate a `println` e `close` siano incluse in un blocco `try`, dato che non generano eccezioni che debbano essere gestite obbligatoriamente. Il Listato 14.1 contiene un programma semplice ma completo che crea un file di testo a partire da dati inseriti dall'utente. Si noti che le righe nel file di testo risultante sono le stesse che comparirebbero se si scrivesse sullo schermo. Il file può essere letto utilizzando un editor di testi o un altro programma Java, come si mostrerà in seguito.

MyLab

LISTATO 14.1 Scrivere in un file di testo.

```
import java.io.PrintWriter;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FileDiTestoOutputDemo {
    public static void main(String[] args) {
        String nomeFile = "out.txt";     //Il nome potrebbe anche
                                         //essere letto da tastiera
        PrintWriter outputStream = null;
        try {
            outputStream = new PrintWriter(nomeFile);
        } catch (FileNotFoundException e) {
            System.out.println("Errore nell'apertura del file " + nomeFile);
            System.exit(0);
        }

        System.out.println("Inserire tre righe di testo:");
        Scanner tastiera = new Scanner(System.in);
        for (int contatore = 1; contatore <= 3; contatore++) {
            String riga = tastiera.nextLine();
            outputStream.println(contatore + " " + riga);
        }
```

```
        outputStream.close();
        System.out.println("Le righe sono state scritte su " + nomeFile);
    }
}
```

Esempio di output

```
Inserire tre righe di testo:
Un albero alto
in una foresta bassa è come
un pesce grande in uno stagno piccolo.
Le righe sono state scritte su out.txt
```

File risultante

```
1 Un albero alto
2 in una foresta bassa è come
3 un pesce grande in uno stagno piccolo.
```

Si può utilizzare un editor di testi per leggere questo file.

Creare un file di testo

Sintassi

```
// Aprire il file in scrittura
PrintWriter nome_stream_di_output = null;
try {
    nome_stream_di_output = new PrintWriter(nome_file);
} catch (FileNotFoundException e) {
    istruzioni per la gestione dell'eccezione
}
// Scrivere dati nel file utilizzando le seguenti istruzioni:
nome_stream_di_output.println(...);
nome_stream_di_output.print(...);
// Chiudere il file
nome_stream_di_output.close();
```

Esempio

Si veda il Listato 14.1.

Un programma dovrebbe fornire informazioni

Un programma che crei un file dovrebbe informare l'utente quando ha finito di scrivere nel file. Altrimenti si avrebbe un cosiddetto **programma silenzioso**, e l'utente potrebbe chiedersi se il programma abbia completato le sue operazioni con successo o se non abbia incontrato qualche problema. Questo suggerimento si applica sia al caso dei file di testo sia a quello dei file binari.

Un file ha due nomi all'interno di un programma

Ogni file utilizzato da un programma, indipendentemente dal fatto che sia utilizzato in lettura o in scrittura, compare con due nomi: il nome vero e proprio del file utilizzato dal sistema operativo e il nome dello stream collegato al file. Il nome del file serve per connettere il file allo stream, mentre il nome dello stream è utilizzato da quel momento in poi per lavorare sul file. Il nome dello stream non esiste più dopo che l'esecuzione del programma è terminata, mentre il nome del file rimane. Si noti che, poiché un oggetto di tipo stream può essere referenziato da più di una variabile, un file può in realtà avere anche più di due nomi. Tuttavia, la cosa importante qui è distinguere tra il nome del file e il nome dello stream utilizzato per gestirlo.

FAQ Quali regole vanno seguite per dare i nomi ai file?

Le regole da seguire nella scelta dei nomi per i file dipendono dal sistema operativo, non da Java. Quando si passa il nome di un file al costruttore di uno stream, non gli si sta fornendo un riferimento a un oggetto Java, ma una stringa contenente il nome del file. La maggior parte dei sistemi operativi permette di utilizzare lettere, cifre e punti nel nome di un file. Molti sistemi operativi permettono di utilizzare anche altri caratteri, ma lettere, cifre e punti dovrebbero essere sufficienti per la maggior parte degli scopi. L'estensione, come `.txt` in `out.txt`, non ha alcun significato particolare per un programma Java. Spesso si utilizza questa estensione per indicare un file di testo, ma si tratta soltanto di una convenzione di uso comune. Si può utilizzare qualunque nome di file consentito dal sistema operativo, ma si tenga presente che alcuni sistemi operativi potrebbero nascondere l'estensione automaticamente.

Un blocco `try` è un blocco

Si consideri di nuovo il Listato 14.1. Non è per ragioni stilistiche che si è scelto di dichiarare la variabile `outputStream` al di fuori del blocco `try`. Si supponga infatti di spostarne la dichiarazione all'interno del blocco, come segue:

```
try {
   PrintWriter outputStream = new PrintWriter(nomeFile);
}
```

Questo spostamento può sembrare a prima vista innocuo, ma in realtà rende la variabile `outputStream` locale al blocco `try`. Di conseguenza, non sarebbe possibile utilizzarla al di fuori del blocco. Se si provasse a farlo, si otterrebbe un messaggio di errore secondo il quale `outputStream` è un identificativo sconosciuto.

Nel Capitolo 10 è stato suggerito di definire sempre un metodo `toString` nelle classi. Tale metodo produce una rappresentazione sotto forma di stringa dei dati di una classe. I metodi `print` e `println` di `System.out` invocano automaticamente il metodo `toString` quando viene loro passato un oggetto come argomento. Lo stesso accade con i metodi `print` e `println` di `PrintWriter`.

Per esempio, si potrebbe aggiungere un metodo `toString` alla classe `Specie` del Capitolo 8. La classe ha tre variabili di istanza: `nome`, `popolazione` e `tassoCrescita`. Quindi, si potrebbe definire `toString` come segue:

```
public String toString() {
   return "Nome = " + nome + "\n" + "Popolazione = " + popolazione + "\n" +
          "Tasso di crescita = " + tassoCrescita + "%";
}
```

Avendo definito anche un costruttore che accetta in ingresso il nome, la popolazione e quindi il tasso di crescita, le istruzioni

```
Specie unEsemplare = new Specie("Condor della California", 27, 0.02);
System.out.println(unEsemplare.toString());
```

produrranno l'output

```
Nome = Condor della California
Popolazione = 27
Tasso di crescita = 0.02%
```

Inoltre,

```
System.out.println(unEsemplare);
```

richiamerà automaticamente il metodo `toString` e produrrà quindi lo stesso output.

Lo stesso vale se si scrive in un file di testo. Sia l'istruzione

```
outputStream.print(unEsemplare);
```

che

```
outputStream.println(unEsemplare);
```

scriveranno lo stesso output visto prima nel file di testo collegato allo stream `outputStream`.

Il programma `EsempioScritturaSpecieFileTesto.java`, incluso nel codice sorgente scaricabile dal sito web del testo, mostra questo comportamento.

Definire il metodo `toString`

Dato che i metodi `print` e `println` invocano automaticamente il metodo `toString`, sia che essi appartengano a `System.out` o a un qualunque altro oggetto di tipo stream, è opportuno definire il metodo `toString` in tutte le classi.

Sovrascrivere un file

Quando si apre in scrittura un file (di testo o binario), si parte sempre con un file vuoto. Se non esiste un file con il nome specificato, il costruttore dello stream ne creerà uno nuovo, ma se esiste già un file con quel nome, il suo contenuto verrà eliminato e il nuovo output verrà scritto in quel file. Il Paragrafo 14.3 mostrerà come controllare se un file esiste già per evitare di sovrascriverlo accidentalmente.

14.2.2 Aggiungere dati a un file di testo

Le operazioni per l'apertura di un file mostrate nel Listato 14.1 garantiscono di partire sempre da un file vuoto. Se esiste già un file con il nome specificato, il suo contenuto viene perso. A volte, tuttavia, questo comportamento non è quello desiderato: si potrebbe voler semplicemente aggiungere altri dati alla fine del file. Per poter **aggiungere** l'output di un programma a un file di testo il cui nome è riportato nella variabile di tipo `String` `nomeFile`, la connessione tra il file e lo stream `outputStream` dovrà essere effettuata in questo modo:

```
outputStream = new PrintWriter(new FileOutputStream(nomeFile, true));
```

Poiché la classe `PrintWriter` non offre direttamente un costruttore che consenta l'operazione di aggiunta, si ricorre alla classe `FileOutputStream`, che dovrà essere anch'essa importata dal package `java.io`. Il secondo argomento (`true`) passato al costruttore di `FileOutputStream` indica che si vogliono aggiungere dati al file se questo esiste già. Quindi, se il file esiste, il contenuto originale viene conservato e l'output del programma verrà inserito dopo di esso. Se però il file non esiste ancora, Java creerà un file nuovo vuoto e vi inserirà l'output del programma. In questo secondo caso, il risultato sarà lo stesso del Listato 14.1.

Quando si aggiungono dati a un file di testo in questo modo, si utilizzano ancora i blocchi `try` e `catch` come nel Listato 14.1. Una versione del programma del Listato 14.1 che aggiunge dati alla fine del file `out.txt` è riportata nel file `AggiungiAFileTesto.java`, incluso nel codice sorgente scaricabile dal sito web del testo.

Apertura di un file di testo per l'aggiunta di dati

È possibile creare uno stream di tipo `PrintWriter` che aggiunga dati alla fine di un file di testo.

Sintassi

```
PrintWriter nome_stream_output =
            new PrintWriter(new FileOutputStream(nome_file, true));
```

Esempio

```
PrintWriter outputStream =
            new PrintWriter(new FileOutputStream("out.txt", true));
```

Dopo queste istruzioni, si possono utilizzare i metodi `print` e `println` per la scrittura; il nuovo testo prodotto verrà aggiunto dopo il testo già presente nel file (in un caso realistico, è opportuno separare la dichiarazione della variabile di stream dalla chiamata al costruttore, come mostrato nel Listato 14.1, in modo da poter gestire un'eventuale eccezione di tipo `FileNotFoundException` che potrebbe essere generata quando si cerca di aprire il file).

14.2.3 Leggere da un file di testo

Le due classi di tipo stream più utilizzate per leggere file di testo sono `Scanner` e `BufferedReader`. Entrambi gli approcci verranno descritti nel seguito. La classe `Scanner` offre un insieme di metodi più ricco ed è la soluzione preferibile per la lettura di dati da un file di testo. Tuttavia, anche la classe `BufferedReader` è molto utilizzata e costituisce una scelta ragionevole.

14.2.4 Leggere un file di testo con la classe `Scanner`

Il Listato 14.2 contiene un semplice programma che legge dati da un file di testo utilizzando la classe `Scanner` e li mostra sullo schermo. Il file `out.txt` è un file di testo che potrebbe essere stato creato da qualcuno utilizzando un editor di testi o da un programma Java (come quello del Listato 14.1) utilizzando la classe `PrintWriter`. Si noti che la classe `Scanner` è la stessa utilizzata nei capitoli precedenti per leggere dati da tastiera. In quel caso, si passava `System.in` come argomento al costruttore della classe `Scanner`.

LISTATO 14.2 Leggere dati da un file di testo con la classe `Scanner`.

MyLab

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

public class FileDiTestoInputConScannerDemo {
    public static void main(String[] args) {
        String nomeFile = "out.txt";
        Scanner inputStream = null;
        System.out.println("Il file " + nomeFile + "\ncontiene le righe
                                    seguenti:\n");

        try {
            inputStream = new Scanner(new File(nomeFile));
        } catch (FileNotFoundException e) {
            System.out.println("Errore nell'apertura del file " + nomeFile);
            System.exit(0);
        }
        while (inputStream.hasNextLine()) {
            String riga = inputStream.nextLine();
            System.out.println(riga);
        }
```

```
            inputStream.close();
        }
    }
```

Esempio di output

```
Il file out.txt
contiene le righe seguenti:
1 Un albero alto
2 in una foresta bassa è come
3 un pesce grande in uno stagno piccolo.
```

Non si può passare direttamente il nome del file al costruttore di `Scanner`. Nonostante la classe abbia un costruttore che accetta un argomento di tipo `String`, in quel caso la stringa è interpretata come sequenza di dati e non come il nome di un file. La classe `Scanner` ha però un costruttore che accetta come argomento un'istanza della classe standard `File`, la quale ha un costruttore che accetta come argomento un nome di file (il prossimo paragrafo descriverà la classe `File` più dettagliatamente). Quindi, un'istruzione come quella che segue aprirà il file in lettura:

```
Scanner nome_stream = new Scanner(new File(nome_file));
```

Se si prova ad aprire in lettura un file che non esiste, il costruttore di `Scanner` genererà una `FileNotFoundException`. Inoltre, come visto nel paragrafo precedente, questa eccezione può essere generata anche in altri casi.

Si noti che il programma del Listato 14.2 è simile a quello del Listato 14.1, che crea un file di testo. Entrambi aprono il file all'interno di un blocco `try-catch`, eseguono alcune operazioni sul file e alla fine lo chiudono. Le istruzioni del Listato 14.2 che leggono e mostrano su schermo l'intero contenuto del file sono le seguenti:

```
while (inputStream.hasNextLine()) {
    String riga = inputStream.nextLine();
    System.out.println(riga);
}
```

Questo ciclo legge e mostra ogni riga del file, una per volta, finché non viene raggiunta la fine del file. L'output di esempio riportato nel Listato 14.2 è quello che si ottiene se il file `out.txt` è quello creato nel Listato 14.1.

Tutti i metodi della classe `Scanner` già visti per la lettura di dati da tastiera possono essere utilizzati anche con i file di testo e funzionano allo stesso modo. Alcuni di questi metodi, compreso `nextLine`, sono riportati nella Figura 2.7 del Capitolo 2. Tuttavia, in precedenza non è mai stato utilizzato il metodo `hasNextLine`. Questo metodo restituisce `true` se nel file è presente ancora almeno un'altra riga da leggere. La Figura 14.3 riporta questo metodo e alcuni altri metodi simili.

MyLab

Video 14.1 Scrivere e leggere un file di testo

Leggere un file di testo con la classe `Scanner`

Sintassi

```
// Aprire il file
Scanner nome_stream_di_input = null;
```

```
try {
    nome_stream_di_input = new Scanner(new File(nome_file));
} catch (FileNotFoundException e) {
    istruzioni per la gestione dell'eccezione
}
// Leggere dati dal file utilizzando istruzioni del tipo:
nome_stream_di_input.metodo_Scanner();
// Chiudere il file
nome_stream_di_input.close();
```

Esempio

Si veda il Listato 14.2.

nome_oggetto_scanner.hasNext() Restituisce **true** se sono disponibili altri dati da leggere mediante il metodo **next**.
nome_oggetto_scanner.hasNextDouble() Restituisce **true** se sono disponibili altri dati da leggere mediante il metodo **nextDouble**.
nome_oggetto_scanner.hasNextInt() Restituisce **true** se sono disponibili altri dati da leggere mediante il metodo **nextInt**.
nome_oggetto_scanner.hasNextLine() Restituisce **true** se sono disponibili altri dati da leggere mediante il metodo **nextLine**.

Figura 14.3 Altri metodi della classe `Scanner` (si veda anche la Figura 2.7).

14.2.5 Leggere un file di testo con la classe `BufferedReader`

Prima dell'introduzione della classe `Scanner` nella versione 5.0 di Java, la classe `BufferedReader` era la classe di tipo stream preferibile per leggere un file di testo e anche ora è utilizzata spesso per questa funzione. L'utilizzo della classe `BufferedReader` è illustrato nel Listato 14.3, che contiene un programma che legge delle righe di testo dal file `originale.txt` e le scrive sullo schermo. Il file `originale.txt` potrebbe essere stato creato da qualcuno utilizzando un editor di testi o da un altro programma Java tramite la classe `PrintWriter`. Il programma apre in lettura il file `originale.txt` in questo modo:

```
BufferedReader inputStream =
                        new BufferedReader(new FileReader("originale.txt"));
```

La classe `BufferedReader`, come la classe `Scanner`, non ha un costruttore che accetti un nome di file come argomento, quindi è necessario utilizzare un'altra classe (in questo caso, la classe `FileReader`) per convertire il nome del file in un oggetto che possa essere passato come argomento al costruttore di `Bufferedreader`.

```
        inputStream.close();
    }
}
```

Esempio di output

```
Il file out.txt
contiene le righe seguenti:
1 Un albero alto
2 in una foresta bassa è come
3 un pesce grande in uno stagno piccolo.
```

Non si può passare direttamente il nome del file al costruttore di `Scanner`. Nonostante la classe abbia un costruttore che accetta un argomento di tipo `String`, in quel caso la stringa è interpretata come sequenza di dati e non come il nome di un file. La classe `Scanner` ha però un costruttore che accetta come argomento un'istanza della classe standard `File`, la quale ha un costruttore che accetta come argomento un nome di file (il prossimo paragrafo descriverà la classe `File` più dettagliatamente). Quindi, un'istruzione come quella che segue aprirà il file in lettura:

```
Scanner nome_stream = new Scanner(new File(nome_file));
```

Se si prova ad aprire in lettura un file che non esiste, il costruttore di `Scanner` genererà una `FileNotFoundException`. Inoltre, come visto nel paragrafo precedente, questa eccezione può essere generata anche in altri casi.

Si noti che il programma del Listato 14.2 è simile a quello del Listato 14.1, che crea un file di testo. Entrambi aprono il file all'interno di un blocco `try-catch`, eseguono alcune operazioni sul file e alla fine lo chiudono. Le istruzioni del Listato 14.2 che leggono e mostrano su schermo l'intero contenuto del file sono le seguenti:

```
while (inputStream.hasNextLine()) {
   String riga = inputStream.nextLine();
   System.out.println(riga);
}
```

Questo ciclo legge e mostra ogni riga del file, una per volta, finché non viene raggiunta la fine del file. L'output di esempio riportato nel Listato 14.2 è quello che si ottiene se il file `out.txt` è quello creato nel Listato 14.1.

Tutti i metodi della classe `Scanner` già visti per la lettura di dati da tastiera possono essere utilizzati anche con i file di testo e funzionano allo stesso modo. Alcuni di questi metodi, compreso `nextLine`, sono riportati nella Figura 2.7 del Capitolo 2. Tuttavia, in precedenza non è mai stato utilizzato il metodo `hasNextLine`. Questo metodo restituisce `true` se nel file è presente ancora almeno un'altra riga da leggere. La Figura 14.3 riporta questo metodo e alcuni altri metodi simili.

MyLab

Video 14.1
Scrivere e leggere un file di testo

Leggere un file di testo con la classe `Scanner`

Sintassi

```
// Aprire il file
Scanner nome_stream_di_input = null;
```

```
try {
    nome_stream_di_input = new Scanner(new File(nome_file));
} catch (FileNotFoundException e) {
    istruzioni per la gestione dell'eccezione
}
// Leggere dati dal file utilizzando istruzioni del tipo:
nome_stream_di_input.metodo_Scanner();
// Chiudere il file
nome_stream_di_input.close();
```

Esempio

Si veda il Listato 14.2.

nome_oggetto_scanner`.hasNext()` Restituisce `true` se sono disponibili altri dati da leggere mediante il metodo `next`.
nome_oggetto_scanner`.hasNextDouble()` Restituisce `true` se sono disponibili altri dati da leggere mediante il metodo `nextDouble`.
nome_oggetto_scanner`.hasNextInt()` Restituisce `true` se sono disponibili altri dati da leggere mediante il metodo `nextInt`.
nome_oggetto_scanner`.hasNextLine()` Restituisce `true` se sono disponibili altri dati da leggere mediante il metodo `nextLine`.

Figura 14.3 Altri metodi della classe `Scanner` (si veda anche la Figura 2.7).

14.2.5 Leggere un file di testo con la classe `BufferedReader`

Prima dell'introduzione della classe `Scanner` nella versione 5.0 di Java, la classe `BufferedReader` era la classe di tipo stream preferibile per leggere un file di testo e anche ora è utilizzata spesso per questa funzione. L'utilizzo della classe `BufferedReader` è illustrato nel Listato 14.3, che contiene un programma che legge delle righe di testo dal file `originale.txt` e le scrive sullo schermo. Il file `originale.txt` potrebbe essere stato creato da qualcuno utilizzando un editor di testi o da un altro programma Java tramite la classe `PrintWriter`. Il programma apre in lettura il file `originale.txt` in questo modo:

```
BufferedReader inputStream =
                        new BufferedReader(new FileReader("originale.txt"));
```

La classe `BufferedReader`, come la classe `Scanner`, non ha un costruttore che accetti un nome di file come argomento, quindi è necessario utilizzare un'altra classe (in questo caso, la classe `FileReader`) per convertire il nome del file in un oggetto che possa essere passato come argomento al costruttore di `Bufferedreader`.

```
public class FileDiTestoInputConBufferedReaderDemo {
    public static void main(String[] args) {

        String nomeFile = "originale.txt";
        BufferedReader inputStream = null;
        System.out.println("Il file " + nomeFile +
                           "\ncontiene le righe seguenti:\n");

        try {
            inputStream = new BufferedReader(new FileReader(nomeFile));
        } catch (FileNotFoundException e) {
            System.out.println("Problema nell'apertura del file.");
        }

        try {
            String riga = inputStream.readLine();
            while (riga != null) {
                System.out.println(riga);
                riga = inputStream.readLine();
            }
            inputStream.close();
        } catch (IOException e) {
            System.out.println("Errore nella lettura da " +  nomeFile);
        }
    }
}
```

Esempio di output

```
Il file originale.txt
contiene le righe seguenti:

Oggi e' una bella giornata,
domani piovera',
dopodomani ci sarà invece il sole.
```

La classe `BufferedReader` ha un metodo chiamato `readLine` che è simile al metodo `nextLine` della classe `Scanner`. Il Listato 14.3 illustra l'utilizzo del metodo `readLine` per leggere un file di testo. Se il programma tenta di leggere oltre la fine del file, il metodo restituisce un valore speciale per segnalare che la fine del file è stata raggiunta. Tale valore speciale è `null`. Di conseguenza, come illustrato nel Listato 14.3, il programma può constatare di aver raggiunto la fine del file verificando se `readLine` restituisce `null`.

Le classi `BufferedReader` e `FileReader` appartengono al package `java.io`.

`public Bufferedreader(Reader oggettoReader)`

Questo è l'unico costruttore che si utilizza comunemente. Poiché non esiste un costruttore che accetti come argomento il nome di un file, se si vuole specificare un file mediante il suo nome bisogna utilizzare

`new BufferedReader(new FileReader(`*nome_file*`))`

Utilizzato in questo modo, il costruttore di `FileReader` (e quindi l'invocazione del costruttore di `BufferedReader`) può generare una `FileNotFoundException`, che è una `IOException`.

Se si vuole creare uno stream a partire da un oggetto della classe `File` (la classe `File` è descritta più avanti; qui la si cita per completezza), si usa l'istruzione

`new Bufferedreader(new FileReader(`*oggetto_file*`))`

Anche in questo caso, il costruttore di `FileReader` può generare una `FileNotFoundException`.

`public String readLine() throws IOException`

Legge una riga dallo stream di input e la restituisce. Se la lettura supera la fine del file, viene restituito `null` (si noti che non viene generata una `EOFException` alla fine del file; la fine del file è segnalata restituendo il valore `null`).

`public int read() throws IOException`

Legge un singolo carattere dallo stream di input e lo restituisce sotto forma di valore di tipo `int`. Se la lettura oltrepassa la fine del file, viene restituito il valore `-1`. Si noti che il valore restituito è di tipo `int`. Per ottenere un `char`, è necessario effettuare una conversione. La fine del file è segnalata dal valore `-1`, dato che tutti i caratteri ordinari corrispondono a numeri interi positivi.

`public long skip(long n) throws IOException`

Salta n caratteri.

`public void close() throws IOException`

Chiude lo stream eliminando il collegamento con il file.

ura 14.4 Alcuni metodi della classe `BufferedReader`.

Figura 14.4 illustra alcuni metodi della classe `BufferedReader`. Come si può notare, lasse offre solo due metodi per la lettura di dati da un file di testo, `readLine` e `read`. ietodo `readLine` è stato discusso sopra. Il metodo `read` legge un singolo carattere tuendo però un valore di tipo `int` che corrisponde al carattere letto e non il carattere o. Di conseguenza, per ottenere il carattere, occorre effettuare una conversione di tipo e di seguito:

```
char prossimo = (char)(inputStream.read());
```

Questa istruzione assegna a prossimo il primo carattere nel file connesso allo stream `inputStream` che non è ancora stato letto. Come il metodo `readLine`, anche il metodo `read` restituisce un valore speciale nel caso in cui si raggiunga la fine del file. Tale valore speciale è `-1`. Tale scelta è motivata dal fatto che il valore `int` corrispondente a ogni carattere ordinario è un intero positivo.

Si noti infine che il programma del Listato 14.3 gestisce due tipi di eccezione: `FileNotFoundException` e `IOException`. Il tentativo di apertura di un file può generare una `FileNotFoundException`, mentre ogni chiamata a `inputStream.readLine()` può generare una `IOException`. Poiché `FileNotFoundException` è una specializzazione di `IOException`, entrambi i tipi di eccezione potrebbero essere gestiti utilizzando solo il blocco per la `IOException`. Tuttavia, così facendo si avrebbero meno informazioni sulle possibili cause dell'eccezione: non si potrebbe sapere se l'eccezione è stata generata aprendo il file o leggendo i dati dal file aperto.

Leggere un file di testo con la classe `BufferedReader`

Per creare uno stream di tipo `BufferedReader` e collegarlo in lettura a un file di testo:

Sintassi

```
BufferedReader oggetto_stream = new BufferedReader(new FileReader(nome_file));
```

Esempio

```
BufferedReader inputStream =
                    new BufferedReader(new FileReader("originale.txt"));
```

Dopo questa istruzione, si possono utilizzare i metodi `readLine` e `read` per leggere i dati. Quando viene utilizzato in questo modo, il costruttore della classe `FileReader`, e quindi l'invocazione del costruttore di `BufferedReader`, possono generare un'eccezione di tipo `FileNotFoundException`, che è una `IOException`.

Leggere numeri con `BufferedReader`

A differenza della classe `Scanner`, la classe `BufferedReader` non offre metodi per la lettura di valori numerici da un file. È necessario leggere i numeri come stringhe e poi convertire le stringhe in valori di un qualche tipo numerico, come `int` o `double`. Per leggere un singolo numero che occupa da solo una riga, si utilizzi il metodo `readLine` e successivamente `Integer.parseInt`, `Double.parseDouble` o metodi simili per convertire la stringa in un numero. Se sulla stessa riga compaiono più numeri, si legga la riga intera con `readLine` e poi si utilizzi la classe `StringTokenizer` per suddividere la stringa nelle varie parti. Infine, si utilizzi `Integer.parseInt` o simili per convertire ogni parte della stringa in un numero.

I metodi `Integer.parseInt`, `Double.parseDouble` e altri metodi simili che convertono stringhe in numeri sono stati presentati nel Capitolo 9 nel Paragrafo 9.2.5 "Classi wrapper".

La classe `StringTokenizer` è presente nel package `java.util`. Probabilmente l'utilizzo più comune che si fa di tale classe consiste nel decomporre una riga di testo nelle sue parole. L'esempio che segue illustra un tipico utilizzo della classe:

```
StringTokenizer riga = new StringTokenizer("4 7 9");
while (riga.hasMoreTokens()) {
   System.out.println(riga.nextToken());
}
```

L'esecuzione di questo codice produrrà il seguente output:

```
4
7
9
```

L'invocazione al costruttore

```
new StringTokenizer("4 7 9");
```

produrrà un nuovo oggetto di tipo `StringTokenizer`. È chiaramente possibile utilizzare una qualsiasi stringa al posto di "`4 7 9`". L'oggetto così creato può essere usato per produrre le singole parole della stringa passata come argomento al costruttore. Queste singole parole sono chiamate token.

Il metodo `nextToken` restituisce il primo token quando è invocato la prima volta, restituisce il secondo token quando è invocato la seconda volta e così via.

Il metodo `hasMoreTokens` restituisce un valore booleano: `true` fintantoché il metodo `nextToken` non ha restituito tutti i token presenti nella stringa, `false` dopo che il metodo `nextToken` ha restituito tutti i token nella stringa.

Individuare la fine del file con `BufferedReader`

Quando, leggendo un file di testo mediante i metodi `readLine` o `read` della classe `BufferedReader`, il programma cerca di leggere dati oltre la fine del file, il metodo scelto restituirà un valore speciale per segnalare che è stata raggiunta la fine del file. Il metodo `readLine` restituisce il valore `null`, come mostrato nel Listato 14.3. Il metodo `read` restituisce invece il valore `-1`, che può essere utilizzato per segnalare la fine del file perché i valori interi corrispondenti ai caratteri normali sono sempre positivi.

14.3 Tecniche generiche per la gestione dei file

Questo paragrafo presenta alcune tecniche che possono essere utilizzate sia con i file di testo che con quelli binari, anche se gli esempi riportati qui considereranno sempre file di testo. Per prima cosa viene descritta la classe `File`, già utilizzata nel paragrafo precedente nella lettura da un file di testo.

14.3.1 La classe `File`

La classe `File` consente una gestione omogenea dei file a partire dal loro nome. Una stringa come "`tesoro.txt`", infatti, potrebbe rappresentare il nome di un file, ma Java la interpreterà come una semplice stringa e non come il nome di un file. D'altra parte, se si passa il nome di un file sotto forma di stringa al costruttore della classe `File`, questo produce un oggetto che può essere interpretato come un identificativo del file. In altre parole, si tratta di un'astrazione indipendente dalla piattaforma, anziché di un vero file. Per esempio, l'oggetto

```
new File("tesoro.txt")
```

non è una semplice stringa, ma un oggetto che "sa" di essere interpretato come l'identificativo di un file.

Nonostante alcune classi abbiano un costruttore che accetta come argomento un nome di file, altre non lo hanno. Alcune classi di tipo stream hanno solo costruttori che accettano come argomento un oggetto di tipo `File`. Si è già visto, in precedenza, che non è possibile passare un nome di file sotto forma di stringa al costruttore della classe `Scanner`. Al contrario, la classe `PrintWriter`, utilizzata per scrivere in un file di testo, ha sia un costruttore che accetta una stringa come nome del file, sia un costruttore che accetta un'istanza di `File`.

Prima di procedere con la descrizione dei metodi della classe `File`, verrà mostrato un esempio nel quale il nome del file viene chiesto all'utente. Nonostante l'esempio utilizzi un file di testo, si potrebbe procedere in modo analogo anche con file binari.

ESEMPIO DI PROGRAMMAZIONE
LEGGERE IL NOME DI UN FILE DA TASTIERA

Fino a questo momento, i nomi dei file da utilizzare sono stati riportati direttamente come stringhe nel codice. Tuttavia, il nome del file da utilizzare potrebbe essere ancora sconosciuto quando si scrive un programma, quindi si vuole richiedere all'utente di inserirlo tramite la tastiera durante l'esecuzione del programma. Ciò è semplice da realizzare: basta far leggere al programma il nome e salvarlo in una variabile di tipo `String`, come mostrato nel Listato 14.4. Il programma è simile a quello del Listato 14.2, ma legge da tastiera il nome del file da utilizzare.

LISTATO 14.4 Lettura del nome di un file.

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

public class FileDiTestoInputConScannerDemo2 {
    public static void main(String[] args) {
        System.out.print("Inserire il nome di un file: ");
        Scanner tastiera = new Scanner(System.in);
        String nomeFile = tastiera.next();
        Scanner inputStream = null;
        System.out.println("Il file " + nomeFile + "\n" +
                           "contiene le righe seguenti:\n");
```

```
        try {
            inputStream = new Scanner(new File(nomeFile));
        } catch (FileNotFoundException e) {
            System.out.println("Errore nell'apertura del file " + nomeFile);
            System.exit(0);
        }
        while (inputStream.hasNextLine()) {
            String riga = inputStream.nextLine();
            System.out.println(riga);
        }
        inputStream.close();
    }
}
```

Esempio di output

```
Inserire il nome di un file: out.txt
Il file out.txt
contiene le righe seguenti:

1 Mela
2 Pera
3 Banana
4 Fragola
```

Si noti che il programma legge il proprio input da due sorgenti diverse. L'oggetto Scanner di nome `tastiera` è utilizzato per leggere il nome di un file dalla tastiera. L'oggetto Scanner chiamato `inputStream` è collegato al file specificato ed è utilizzato per leggere dati dal file.

14.3.2 Percorsi

Quando si utilizza il nome di un file per aprirlo in uno dei modi visti in precedenza, si presuppone che il file si trovi nella stessa directory (cartella) nella quale viene eseguito il programma. Tuttavia, se il file si trova in una directory diversa da quella del programma, la si può specificare utilizzando un **percorso** (***path name***) anziché il solo nome del file. Un percorso **assoluto** o **completo** (***full path name***), come il nome suggerisce, fornisce le indicazioni per individuare il file a partire dalla directory radice. Un **percorso relativo** (***relative path name***) fornisce il percorso per raggiungere il file relativamente alla directory di esecuzione del programma. Il modo in cui vanno specificati i percorsi dipende dal sistema operativo e verranno qui discussi solo alcuni esempi.

Un esempio di un tipico percorso per i sistemi operativi tipo UNIX è

```
/user/daniela/lavoro/dati.txt
```

Per creare uno stream di input collegato a questo file, si scriverebbe

```
Scanner inputStream = new Scanner(new File("/user/daniela/lavoro/dati.txt"));
```

Il sistema operativo Windows utilizza il carattere \ (*backslash*) al posto di / (*slash*) nei percorsi. Un tipico percorso per un file in un sistema Windows è

```
C:\lavoro\dati.txt
```

In questo caso, per creare uno stream di input si scriverebbe

```
Scanner inputStream = new Scanner(new File("C:\\lavoro\\dati.txt"));
```

Si noti che è necessario utilizzare \\ al posto di \, altrimenti Java interpreterà un backslash seguito da un altro carattere (per esempio, \d) some una sequenza di escape.

Nonostante di norma occorra fare attenzione ai caratteri backslash nelle stringhe, questo problema non si verifica quando si legge una stringa da tastiera. Supponendo di eseguire il programma del Listato 14.4 nel modo seguente

```
Inserire il nome di un file: C:\lavoro\dati.txt
```

il programma interpreterà il percorso nel modo corretto. Non è necessario che l'utente scriva il percorso in questo modo

```
C:\\lavoro\\dati.txt
```

Al contrario, utilizzare \\ potrebbe produrre un'errata interpretazione del percorso. Durante l'inserimento da tastiera, Java "capisce" che la sequenza \d va interpretata come un carattere backslash seguito da una d, e non come un carattere di escape.

Un modo per evitare tutti questi problemi legati all'interpretazione dei caratteri è utilizzare sempre la notazione UNIX per la composizione dei percorsi. Infatti, un programma Java accetterà percorsi scritti sia nel formato Windows che nel formato UNIX, anche se viene eseguito su un sistema operativo che utilizza un formato diverso. Quindi, un modo alternativo per creare uno stream di input collegato al file Windows

```
C:\lavoro\dati.txt
```

è il seguente:

```
Scanner inputStream = new Scanner(new File("C:/lavoro/dati.txt"));
```

14.3.3 Metodi della classe `File`

I metodi della classe `File` possono essere utilizzati per analizzare le proprietà dei file. Per esempio, è possibile verificare se un file ha un nome specificato o se è leggibile.

Si supponga di creare un'istanza di `File` di nome `oggettoFile` utilizzando il seguente codice:

```
File oggettoFile = new File("tesoro.txt");
```

Si ricordi che un oggetto file non è esso stesso un file, ma un'astrazione, indipendente dal sistema, del percorso del file (in questo caso, `tesoro.txt`).

Dopo aver creato l'oggetto `oggettoFile`, si può utilizzare il metodo `exists` della classe `File` per verificare se esiste un file con il nome specificato. Per esempio, si può scrivere

```
if (!oggettoFile.exists())
  System.out.println("Non esiste alcun file con quel nome.");
```

Se il file esiste, si può utilizzare il metodo `canRead` per verificare se il sistema operativo consentirà di leggere il file. Per esempio, si potrebbe scrivere

```
if (!oggettoFile.canRead())
  System.out.println("Non è consentito leggere dal file specificato.");
```

La maggior parte dei sistemi operativi consente di designare alcuni file come non leggibili o leggibili solo da certi utenti. Il metodo `canRead` è una buona soluzione per verificare se un file è stato reso non leggibile, volontariamente o accidentalmente.

Le seguenti istruzioni potrebbero essere aggiunte al programma del Listato 14.4 per controllare che il file sia pronto per la lettura:

```
File oggettoFile = new File(nomeFile);
boolean fileOK = false;
while (!fileOK) {
    if (!oggettoFile.exists())
        System.out.println("Il file non esiste");
    else if (!oggettoFile.canRead())
        System.out.println("Il file non può essere letto.");
    else
        fileOK = true;

    if (!fileOK) {
        System.out.println("Reinserire il nome del file:");
        nomeFile = tastiera.next( );
        oggettoFile = new File(nomeFile);
    }
}
```

Il programma del Listato 14.4, modificato nel modo appena descritto, è disponibile nel file `FileDiTestoInputConScannerDemo3.java` incluso nel codice sorgente scaricabile dal sito web del testo.

Il metodo `canWrite` è simile a `canRead` e consente di verificare se il sistema operativo consente di scrivere nel file. La maggior parte dei sistemi operativi consente infatti di indicare alcuni file come non modificabili o modificabili solo da parte di certi utenti. La Figura 14.5 elenca i due metodi citati e alcuni altri della classe `File`.

La classe `File`

La classe `File` è utilizzata per rappresentare identificativi di file. Il costruttore della classe `File` richiede come argomento una stringa e produce un oggetto che può essere interpretato come un identificativo del file con quel nome. Un oggetto di tipo `File` e i suoi metodi possono essere utilizzati per rispondere a domande come: Il file esiste? Il programma ha il permesso di leggere il file? Il programma ha il permesso di scrivere nel file? La Figura 14.5 riassume alcuni dei metodi della classe `File`.

Esempio

```
File oggettoFile = new File("dati.txt");

if (!oggettoFile.exists())
   System.out.println("Non esiste un file di nome dati.txt.");
else if (!oggettoFile.canRead())
   System.out.println("Il file dati.txt non può essere letto.");
```

`public boolean canRead()` Verifica se il programma può leggere dal file.
`public boolean canWrite()` Verifica se il programma può scrivere nel file.
`public boolean delete()` Prova ad eliminare il file. Restituisce `true` se è stato possibile rimuovere il file.
`public boolean exists()` Verifica se esiste un file con il nome utilizzato come argomento al costruttore quando è stato creato l'oggetto `File`.
`public String getName()` Restituisce il nome del file (si noti che si tratta solo del nome, non di un percorso).
`public String getPath()` Restituisce il percorso del file.
`public long length()` Restituisce la lunghezza del file in byte.

Figura 14.5 Alcuni metodi della classe `File`.

FAQ Qual è la differenza tra un file e un oggetto `File`?

Un file è un insieme di dati immagazzinati su una periferica reale, come un disco. Un oggetto `File` è un'astrazione, indipendente dal sistema operativo, del percorso di un file.

CASO DI STUDIO
ELABORAZIONE DI UN FILE CON VALORI SEPARATI DA VIRGOLE

Un file con valori separati da virgole (*comma-separated value* o CSV) è un semplice tipo di file di testo utilizzato per immagazzinare liste di registrazioni. Si utilizza una virgola per separare i campi (detti anche colonne) di ogni riga. Questo formato è spesso utilizzato per trasferire dati tra fogli di calcolo, database e altri programmi. Come esempio, si consideri un registratore di cassa che mantiene un resoconto delle vendite della giornata in un file CSV chiamato `Transazioni.txt`. Questo file di testo contiene i seguenti dati:

```
Codice,Quantità,Prezzo,Descrizione
4039,50,0.99,ARANCIATA
9100,5,9.50,T-SHIRT
1949,30,110.00,LIBRO SU JAVA
5199,25,1.50,BISCOTTI
```

La prima riga del file è un'intestazione che descrive i campi. Il primo campo è un codice univoco associato a ogni prodotto. Il secondo campo è la quantità dei prodotti corrispondenti a quel codice venduti nella transazione. Il terzo campo è il prezzo unitario e l'ultimo una descrizione del prodotto venduto. Per esempio, la seconda riga indica che sono state vendute 50 aranciate a 0.99 € l'una e che il codice associato all'aranciata è 4039.

Questi dati potrebbero essere elaborati in molti modi, ma in questo caso di studio si presenta una strategia semplice per leggere tutti i campi dal file, mostrare ogni transazione in un formato più facilmente leggibile e calcolare l'ammontare delle vendite per il registratore di cassa. L'algoritmo generale è il seguente:

1. Leggere e saltare la riga di intestazione
2. Ripetere finché non è stata raggiunta la fine del file:
   a. Leggere dal file una riga completa sotto forma di stringa
   b. Creare, a partire dalla riga intera, un array di stringhe dove `array[0]` è il valore del primo campo, `array[1]` del secondo e così via
   c. Convertire ogni campo numerico nell'array di stringhe nel tipo numerico appropriato
   d. Elaborare i campi

Il passo 2b dell'algoritmo potrebbe sembrare complicato. Fino a questo punto è stata semplicemente letta una riga dal file. Nel caso della prima riga dell'esempio, si avrà la stringa "`4039,50,0.99,ARANCIATA`" in una variabile di tipo `String`. Poiché è stata utilizzata una virgola per separare i campi, si potrebbe cercare la prima occorrenza di una virgola nella stringa, estrarre la sottostringa che va dall'inizio della riga alla posizione della virgola per ottenere il primo campo, e ripetere il tutto per i campi successivi. Tuttavia, il metodo `split` della classe `String` fa già tutto questo automaticamente:

```
public String[] split(String separatore)
```

Il metodo suddivide la stringa in blocchi separati da `separatore` e restituisce un array delle stringhe risultanti. Il parametro `separatore` è interpretato come un'espressione regolare, il che rende l'uso del metodo un modo molto flessibile e potente per individuare strutture nelle stringhe. In questo caso di studio si utilizzerà semplicemente una stringa costituita dalla sola virgola, come nel seguente esempio:

```
String riga = "4039,50,0.99,ARANCIATA";
String[] array = riga.split(",");
System.out.println(array[0]);                    // Scrive 4039
System.out.println(array[1]);                    // Scrive 50
System.out.println(array[2]);                    // Scrive 0.99
System.out.println(array[3]);                    // Scrive ARANCIATA
```

Il Listato 14.5 applica questa tecnica all'esempio del registratore di cassa. Tutto quello che rimane da fare è leggere il file, convertire il campo quantità in un intero, convertire il prezzo in `double` e calcolare il totale delle vendite sommando i prodotti delle quantità vendute per i prezzi corrispondenti. Nel programma si utilizza il metodo `System.out.printf` per formattare il prezzo e il totale con due cifre dopo la virgola.

MyLab

LISTATO 14.5 Elaborazione di un file CSV.

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.File;
import java.util.Scanner;

public class LetturaTransazioni {
    public static void main(String[] args) {
        String nomeFile = "Transazioni.txt";
        try {
            Scanner inputStream = new Scanner(new File(nomeFile));
            // Salta l'intestazione leggendola e poi ignorandola
            String riga = inputStream.nextLine();
            // Vendite totali
            double totale = 0;
            // Legge il resto del file riga per riga
            while (inputStream.hasNextLine()) {
                // Contiene Codice,Quantità,Prezzo,Descrizione
                riga = inputStream.nextLine();
                // Trasforma la stringa in un array di stringhe
                String[] array = riga.split(",");
                // Estrae ogni elemento in una variabile opportuna
                String codice = array[0];
                int quantita = Integer.parseInt(array[1]);
                double prezzo = Double.parseDouble(array[2]);
                String descrizione = array[3];
                // Stampa la registrazione
                System.out.printf("Venduti %d di %s (codice: %s) a " +
                    "€%1.2f l'uno.\n", quantita, descrizione, codice, prezzo);
                // Calcola il totale
                totale += quantita * prezzo;
            }
            System.out.printf("Vendite totali: €%1.2f\n", totale);
            inputStream.close( );
        } catch(FileNotFoundException e) {
            System.out.println("Impossibile trovare il file " + nomeFile);
        } catch(IOException e) {
            System.out.println("Errore nella lettura del file " + nomeFile);
        }
    }
}
```

Esempio di output

```
Venduti 50 di ARANCIATA (codice: 4039) a €0.99 l'uno.
Venduti 5 di T-SHIRT (codice: 9100) a €9.50 l'uno.
Venduti 30 di LIBRO SU JAVA (codice: 1949) a €110.00 l'uno.
Venduti 25 di BISCOTTI (codice: 5199) a €1.50 l'uno.
Vendite totali: €3434.50
```

14.4 Basi dell'I/O con file binari

In questo paragrafo verranno utilizzate le classi `ObjectInputStream` e `ObjectOutputStream` per leggere e scrivere file binari. Queste classi forniscono metodi per leggere e scrivere dati un byte alla volta. Questi stream possono anche convertire numeri e caratteri in byte che possono essere salvati in un file binario, consentendo di scrivere i programmi come se i dati scritti nel file o letti da esso fossero costituiti non da semplici byte, ma da valori di uno dei tipi primitivi di Java (come `int`, `char` e `double`), da stringhe o persino da oggetti di tipo classe, così come da array. Se non è necessario poter visualizzare o modificare un file tramite un editor di testi, il modo più semplice ed efficiente per leggere e scrivere dati da e in un file è utilizzare `ObjectOutputStream` per scrivere file binari e `ObjectInputStream` per leggerli.

Per prima cosa, si mostrerà come creare un file binario e successivamente si discuterà come scrivere in un file binario dati di tipo primitivo e stringhe. Infine, si considererà l'utilizzo di oggetti e array per l'input e output con file binari.

14.4.1 Creare un file binario

Per creare un file binario si può utilizzare la classe di tipo stream `ObjectOutputStream`. Il Listato 14.6 mostra un programma che scrive numeri interi in un file binario. Si analizzeranno ora di seguito i dettagli di questo programma.

LISTATO 14.6 Usare `ObjectOutputStream` per scrivere in un file.

MyLab

```
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Scanner;

public class FileBinarioOutputDemo {
   public static void main(String[] args) {
       String nomeFile = "numeri.dat";
       try {
           ObjectOutputStream outputStream = new ObjectOutputStream(new
                                   FileOutputStream(nomeFile));
           Scanner tastiera = new Scanner(System.in);
           System.out.println("Inserire degli interi non negativi.");
           System.out.println("Inserire un numero negativo per terminare.");

           int unIntero;
```

```
            unIntero = tastiera.nextInt();
            outputStream.writeInt(unIntero);
        } while (unIntero >= 0);

        System.out.println("I numeri e il valore di terminazione");
        System.out.println("sono stati scritti nel file " + nomeFile);
        outputStream.close();
    } catch (FileNotFoundException e) {
        System.out.println("Errore nell'apertura del file " + nomeFile);
    } catch (IOException e) {
        System.out.println("Errore nella scrittura nel file " + nomeFile);
    }
  }
}
```

La chiusura di un file binario avviene allo stesso modo di quella di un file di testo.

Esempio di output

```
Inserire degli interi non negativi.
Inserire un numero negativo per terminare.
1 2 3 -1
I numeri e il valore di terminazione
sono stati scritti nel file numeri.dat
```

Il contenuto del file binario dopo l'esecuzione del programma è:

Questo è un file binario. Non può essere visualizzato utilizzando un editor di testi.

1	2	3	-1

Il -1 in questo file è un cosiddetto "valore sentinella". Come si vedrà in seguito, non è indispensabile terminare un file con un valore sentinella.

Si noti che la parte principale del programma è inclusa in un blocco `try`. Infatti, tutte le istruzioni per l'I/O con file binari che verranno descritte qui possono generare una `IOException`. Gestendo queste eccezioni, il programma può esaminare i messaggi d'errore e terminare normalmente.

Il modo per creare uno stream di output per il file binario `numeri.dat` è il seguente:

```
ObjectOutputStream outputStream = new ObjectOutputStream(new
                                        FileOutputStream("numeri.dat"));
```

Come nel caso dei file di testo, questa operazione è detta apertura del file. Se il file specificato non esiste, questa istruzione creerà un nuovo file vuoto con quel nome. Se, al contrario, esiste già un file con il nome specificato, il suo contenuto preesistente verrà cancellato, in modo da partire con un file vuoto. Il comportamento è sostanzialmente lo stesso già visto per i file di testo, con l'unica differenza che ora si utilizza una classe diversa. Si noti che il costruttore della classe `ObjectOutputStream` non può ricevere come argomento una stringa, cosa che invece può fare il costruttore di `FIleOutputStream`. Inoltre, `ObjectOutputStream` ha un costruttore che accetta un oggetto di tipo

`FileOutputStream` come argomento. Quindi, così come in precedenza si è passato un oggetto `File` al costruttore di `Scanner` per leggere un file di testo, qui si passa un oggetto `FileOutputStream` al costruttore di `ObjectOutputStream`. Si noti che il costruttore di `ObjectOutputStream` può generare una `IOException`, mentre il costruttore di `FileOutputStream` può generare una `FileNotFoundException`.

14.4.2 Scrivere valori di tipo primitivo in un file binario

La classe `ObjectOutputStream` non offre un metodo `println`, a differenza delle classi per la scrittura su schermo o nei file di testo. Tuttavia, questa classe offre un metodo `writeInt` che scrive in un file binario un singolo valore di tipo `int`, oltre ad altri metodi di scrittura che saranno discussi a breve. Quindi, una volta che è stato ottenuto uno stream `outputStream` di tipo `ObjectOutputStream` connesso al file, è possibile scrivere valori interi nel file utilizzando l'istruzione seguente, mostrata nel Listato 14.6:

```
outputStream.writeInt(unIntero);
```

Il metodo `writeInt` può generare una `IOException`.

Il Listato 14.6 mostra il contenuto del file `numeri.dat` come se fossero scritti in un formato direttamente comprensibile. Tuttavia, non è in questo modo che i dati vengono effettivamente salvati nel file. In un file binario non ci sono righe o altri separatori tra i dati, ma questi ultimi sono scritti, sotto forma di sequenze di byte, uno dopo l'altro. Di conseguenza, i valori codificati in questo modo, solitamente, non possono essere letti utilizzando un editor di testo. File codificati in questo modo saranno comprensibili solo ad altri programmi Java.

Uno stream di tipo `ObjectOutputStream` può essere utilizzato per scrivere dati di un qualunque tipo primitivo. Ogni tipo primitivo ha un metodo corrispondente nella classe `ObjectOutputStream`, come `writeLong`, `writeDouble`, `writeFloat` e `writeChar`.

Il metodo `writeChar` può essere usato per scrivere un singolo carattere. Per esempio, la seguente istruzione scriverà il carattere `'A'` nel file connesso allo stream `outputStream`:

```
outputStream.writeChar('A');
```

Il metodo `writeChar` ha una proprietà piuttosto particolare: si aspetta che l'argomento che gli viene passato sia di tipo `int`. Quindi, se si ha una variabile di tipo `char`, sarà necessario convertire il valore in `int` prima di passarlo a `writeChar`. Di conseguenza, l'istruzione precedente è equivalente a questa:

```
outputStream.writeChar((int)'A');
// La conversione di tipo può essere omessa
```

Dopo aver finito di scrivere nel file binario, lo si chiude esattamente come si fa con i file di testo, utilizzando l'istruzione

```
outputStream.close();
```

La Figura 14.6 riassume alcuni metodi della classe `ObjectOutputStream`, inclusi alcuni non ancora discussi fino a questo punto. Molti di questi metodi possono generare una `IOException`.

`public ObjectOutputStream(OutputStream oggettoStream)`

Crea uno stream di output collegato al file binario specificato. Non esiste un costruttore che accetti come argomento il nome del file. Per creare uno stream a partire dal nome del file, bisogna utilizzare

`new ObjectOutputStream(new FileOutputStream(`*nome_file*`))`

oppure, utilizzando la classe `File`

`new ObjectOutputStream(new FileOutputStream(new File(`*nome_file*`)))`

Entrambe le istruzioni creano un file vuoto. Se esisteva un file di nome *nome_file*, il contenuto preesistente viene perso.

Il costruttore di `FileOutputStream` può generare una `FileNotFoundException`. Se ciò non accade, il costruttore di `ObjectOutputStream` può generare una `IOException`.

`public void writeInt(int n) throws IOException`

Scrive il valore `n` di tipo `int` nello stream di output.

`public void writeLong(long n) throws IOException`

Scrive il valore `n` di tipo `long` nello stream di output.

`public void writeDouble(double x) throws IOException`

Scrive il valore `x` di tipo `double` nello stream di output.

`public void writeFloat(float x) throws IOException`

Scrive il valore `x` di tipo `float` nello stream di output.

`public void writeChar(int c) throws IOException`

Scrive un valore `char` nello stream di output. Si noti che il parametro `c` è di tipo `int`. Tuttavia, Java convertirà automaticamente un valore `char` in un `int`. Quindi, la seguente istruzione rappresenta un utilizzo corretto del metodo:

```
outputStream.writeChar('A');
```

`public void writeBoolean(boolean b) throws IOException`

Scrive il valore `b` di tipo `boolean` nello stream di output.

`public void writeUTF(String unaStringa) throws IOException`

Scrive la stringa `unaStringa` nello stream di output. La sigla UTF si riferisce a una particolare codifica per le stringhe. Per leggere la stringa dal file, si utilizzerà il metodo `readUTF` della classe `ObjectInputStream`, come discusso nel prossimo paragrafo.

Figura 14.6 Alcuni metodi della classe `ObjectOutputStream`. (*segue*)

`public void writeObject(Object unOggetto) throws IOException, NotSerializableException, InvalidClassException`
Scrive l'oggetto `unOggetto` nello stream di output. L'argomento deve essere un oggetto di una classe serializzabile, argomento discusso più avanti in questo capitolo. Il metodo genera un'eccezione `NotSerializableException` se l'oggetto è di una classe non serializzabile. Genera una `InvalidClassException` se c'è stato un problema nella serializzazione. Il metodo `writeObject` sarà discusso in modo più approfondito più avanti in questo capitolo.
`public void close() throws IOException`
Chiude lo stream.

Figura 14.6 Alcuni metodi della classe `ObjectOutputStream`.

Creare un file binario

Sintassi

```
try {
    // Aprire il file
    ObjectOutputStream nome_stream_di_output =
            new ObjectOutputStream(new FileOutputStream(nome_file));
    // Scrivere il file utilizzando istruzioni della forma:
    nome_stream_di_output.nome_metodo(argomento);  // Si veda la Figura 14.6
    // Chiudere il file
    nome_stream_di_output.close();
} catch (FileNotFoundException e) {
    Istruzioni_per_la_gestione_dell'eccezione
} catch (IOException e) {
    Istruzioni_per_la_gestione_dell'eccezione
}
```

Esempio

Si veda il Listato 14.6.

14.4.3 Scrivere stringhe in un file binario

Per scrivere stringhe in un file binario si utilizza il metodo `writeUTF`. Per esempio, se `outputStream` è uno stream di tipo `ObjectOutputStream`, la seguente istruzione scriverà la stringa "`Ciao Mamma`" nel file collegato allo stream:

```
outputStream.writeUTF("Ciao Mamma");
```

Ovviamente, con ognuno dei metodi della classe `ObjectOutputStream` si può utilizzare una variabile di tipo appropriato (in questo caso, `String`) al posto di una costante.

È possibile scrivere dati di tipo diverso nello stesso file binario. Per esempio, si potrebbe scrivere una combinazione di valori `int`, `double` e `String`. Tuttavia, mescolare

dati di tipo diverso nello stesso file richiede particolare attenzione affinché in seguito i dati possano essere letti correttamente. In particolare, occorre tenere traccia dell'ordine nel quale i dati sono stati scritti nel file, dato che, come si vedrà di seguito, si utilizza un metodo diverso per leggere ogni tipo di dato.

FAQ Che cosa significa UTF?

Per scrivere un valore `int` in uno stream di tipo `ObjectOutputStream` si usa il metodo `writeInt`, per scrivere un `double`, si usa `writeDouble` e così via. Tuttavia, per scrivere una stringa si usa il metodo `writeUTF`: non esiste un metodo `writeString` in `ObjectOutputStream`. Perché questo nome strano? La sigla UTF è l'acronimo di *Unicode Text Format*. Di per sé, il nome non spiega molto. Il significato è il seguente. Si ricordi che Java utilizza l'insieme di caratteri Unicode, che include anche molti caratteri utilizzati in lingue basate su alfabeti molto diversi da quello inglese. La maggior parte degli editor di testo e dei sistemi operativi utilizza l'insieme di caratteri ASCII, che comprende solo i caratteri utilizzati comunemente nella lingua inglese e nei programmi Java. L'insieme di caratteri ASCII è un sottoinsieme dell'Unicode, quindi l'insieme Unicode contiene molti caratteri che di solito non si utilizzano. Nei paesi di lingua inglese, la codifica Unicode è poco efficiente. La codifica UTF è uno schema di codifica alternativo che consente di rappresentare tutti i caratteri Unicode ma privilegia l'insieme ASCII. Ciò si ottiene assegnando codici brevi ed efficienti da utilizzare ai caratteri ASCII e codici più lunghi e meno efficienti agli altri caratteri Unicode. Se non si utilizzano molto i caratteri Unicode, questo approccio è effettivamente vantaggioso.

14.4.4 Alcuni dettagli sul metodo `writeUTF`

Il metodo `writeInt` scrive valori interi in un file, utilizzando sempre lo stesso numero di byte (quindi lo stesso numero di bit a 0 o a 1) per qualunque numero intero. Analogamente, il metodo `writeLong` usa lo stesso numero di byte per salvare qualunque valore di tipo `long`. I due metodi utilizzano però numeri diversi di byte l'uno rispetto all'altro. La situazione è la stessa per tutti gli altri metodi di scrittura per i tipi primitivi. Il metodo `writeUTF`, al contrario, usa un numero variabile di byte per scrivere stringhe diverse in un file binario. Le stringhe più lunghe richiederanno più byte di quelle più corte. Ciò può rappresentare un problema per Java, dato che in un file binario non esistono separatori tra i singoli dati. Per ovviare a questo problema, Java inserisce delle informazioni aggiuntive all'inizio di ogni stringa. Queste informazioni specificano da quanti byte è composta la stringa, così che il metodo `readUTF` sa quanti byte leggere e decodificare (questo metodo sarà descritto più avanti in questo capitolo, ma come è facile immaginare serve per leggere una stringa da un file binario).

In realtà, il comportamento di `writeUTF` è ancora più complicato di quello appena descritto. Infatti, si è detto che le informazioni all'inizio della stringa specificano quanti byte debbano essere letti e non da quanti caratteri sia composta la stringa. Questi due numeri non coincidono. Nella codifica UTF, caratteri diversi possono essere codificati utilizzando un numero diverso di byte. In ogni caso, tutti i caratteri ASCII sono salvati utilizzando un singolo byte. Quindi, se si utilizzano solo caratteri ASCII, questa distinzione rimane più che altro teorica.

14.4.5 Leggere da un file binario

Se un file binario è stato creato usando un `ObjectOutputStream`, lo si può leggere sfruttando la classe `ObjectInputStream`. La Figura 14.7 mostra alcuni dei metodi più comunemente utilizzati di questa classe. Confrontando questi metodi con quelli della Figura 14.6, si può vedere che ogni metodo di scrittura ha un corrispondente metodo per la lettura. Per esempio, se si scrive un valore intero in un file mediante il metodo `writeInt` di `ObjectOutputStream`, si può leggere quello stesso valore utilizzando il metodo `readInt` di `ObjectInputStream`. Se si scrive un `double` con `writeDouble`, lo si può leggere con `readDouble` e così via.

L'apertura di un file binario in lettura avviene in modo simile a quella in scrittura. Il programma del Listato 14.7 apre un file binario e lo collega a uno stream chiamato `inputStream` in questo modo:

```
ObjectInputStream inputStream =
                    new ObjectInputStream(new FileInputStream(nomeFile));
```

Si noti che questa istruzione è analoga a quella del Listato 14.6, con la differenza che ora si usano le classi `ObjectInputStream` e `FileInputStream` al posto, rispettivamente, di `ObjectOutputStream` e `FileOutputStream`. Di nuovo, la classe da utilizzare per la lettura non ha un costruttore che accetti come argomento una stringa con il nome del file da aprire. Il costruttore di `FileInputStream` può generare una `FileNotFoundException`, che è una sottoclasse di `IOException`. Se invece questo costruttore non genera eccezioni, quello di `ObjectInputStream` può comunque generare una `IOException`.

Un `ObjectInputStream` permette di leggere dati di tipo diverso dallo stesso file. Per esempio, è possibile leggere una combinazione di valori `int`, `double` e `String`. Tuttavia, se si prova a leggere un dato di tipo diverso da quello atteso, il risultato non sarà quello desiderato. Per esempio, se un programma scrive un intero utilizzando `writeInt`, qualunque altro programma che legga quel valore dovrà utilizzare `readInt`. Se invece si utilizzassero, per esempio `readLong` o `readDouble`, il comportamento del programma potrebbe essere imprevedibile.

`public ObjectInputStream(InputStream oggettoStream)`

Crea uno stream di input collegato al file binario specificato. Non esiste un costruttore che accetti come argomento il nome del file. Per creare uno stream a partire dal nome del file, bisogna utilizzare

`new ObjectInputStream(new FileInputStream(`*nome_file*`))`

oppure, utilizzando la classe `File`

`new ObjectInputStream(new FileInputStream(new File(`*nome_file*`)))`

Entrambe le istruzioni creano un file vuoto. Se esisteva un file di nome *nome_file*, il contenuto preesistente viene perso.

Il costruttore di `FileInputStream` può generare una `FileNotFoundException`. Se ciò non accade, il costruttore di `ObjectInputStream` può generare una `IOException`.

Figura 14.7 Alcuni metodi della classe `ObjectInputStream`. *(segue)*

`public int readInt() throws EOFException, IOException` Legge un valore di tipo `int` dallo stream di input e lo restituisce. Se il metodo cerca di leggere un valore che non è stato scritto utilizzando il metodo `writeInt` della classe `ObjectOutputStream` (o in qualche modo equivalente) si avranno problemi. Se il tentativo di lettura oltrepassa la fine del file, viene generata una `EOFException`.
`public long readLong() throws EOFException, IOException` Legge un valore di tipo `long` dallo stream di input e lo restituisce. Se il metodo cerca di leggere un valore che non è stato scritto utilizzando il metodo `writeLong` della classe `ObjectOutputStream` (o in qualche modo equivalente) si avranno problemi. Se il tentativo di lettura oltrepassa la fine del file, viene generata una `EOFException`. Si noti che non è possibile scrivere un valore intero usando `writeLong` e in seguito leggerlo usando `readInt` o, viceversa, scriverlo con `writeInt` e leggerlo con `readLong`. Se si prova a fare ciò, si otterranno risultati imprevedibili.
`public double readDouble() throws EOFException, IOException` Legge un valore di tipo `double` dallo stream di input e lo restituisce. Se il metodo cerca di leggere un valore che non è stato scritto utilizzando il metodo `writeDouble` della classe `ObjectOutputStream` (o in qualche modo equivalente) si avranno problemi. Se il tentativo di lettura oltrepassa la fine del file, viene generata una `EOFException`.
`public float readFloat() throws EOFException, IOException` Legge un valore di tipo `float` dallo stream di input e lo restituisce. Se il metodo cerca di leggere un valore che non è stato scritto utilizzando il metodo `writeFloat` della classe `ObjectOutputStream` (o in qualche modo equivalente) si avranno problemi. Se il tentativo di lettura oltrepassa la fine del file, viene generata una `EOFException`. Si noti che non è possibile scrivere un valore in virgola mobile usando `writeDouble` e in seguito leggerlo usando `readFloat` o, viceversa, scriverlo con `writeFloat` e leggerlo con `readDouble`. Se si prova a fare ciò, si otterranno risultati imprevedibili, così come con qualunque altra coppia di tipi primitivi diversi, come provando a scrivere un valore con `writeInt` e a leggerlo con `readFloat` o `readDouble`.
`public char readChar() throws EOFException, IOException` Legge un valore di tipo `char` dallo stream di input e lo restituisce. Se il metodo cerca di leggere un valore che non è stato scritto utilizzando il metodo `writeChar` della classe `ObjectOutputStream` (o in qualche modo equivalente) si avranno problemi. Se il tentativo di lettura oltrepassa la fine del file, viene generata una `EOFException`.
`public boolean readBoolean() throws EOFException, IOException` Legge un valore di tipo `boolean` dallo stream di input e lo restituisce. Se il metodo cerca di leggere un valore che non è stato scritto utilizzando il metodo `writeBoolean` della classe `ObjectOutputStream` (o in qualche modo equivalente) si avranno problemi. Se il tentativo di lettura oltrepassa la fine del file, viene generata una `EOFException`.

Figura 14.7 Alcuni metodi della classe `ObjectInputStream`. (*segue*)

`public String readUTF() throws IOException, UTFDataFormatException` Legge un valore di tipo `String` dallo stream di input e lo restituisce. Se il metodo cerca di leggere un valore che non è stato scritto utilizzando il metodo `writeUTF` della classe `ObjectOutputStream` (o in qualche modo equivalente) si avranno problemi. Può generare una `UTFDataFormatException` o una `IOException`.
`public Object readObject() throws ClassNotFoundException, InvalidClassException, OptionalDataException, IOException` Legge un oggetto dallo stream di input e lo restituisce. Genera una `ClassNotFoundException` se non è stato possibile trovare un oggetto di una classe serializzabile. Genera una `InvalidClassException` se c'è stato un problema con una classe serializzabile. Genera una `OptionalDataException` se nello stream è stato trovato un valore di un tipo primitivo al posto di un oggetto. Genera una `IOException` se c'è stato qualche altro tipo di problema di I/O. Il metodo `readObject` sarà analizzato con maggiore dettaglio nel prossimo paragrafo.
`public void close() throws IOException` Chiude lo stream.

Figura 14.7 Alcuni metodi della classe `ObjectInputStream`.

LISTATO 14.7 Usare `ObjectInputStream` per la lettura da un file.

MyLab

Presuppone che sia già stato eseguito il programma del Listato 14.6.

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;

public class FileBinarioInputDemo {
    public static void main(String[] args) {
        String nomeFile = "numeri.dat";
        try {
            ObjectInputStream inputStream = new ObjectInputStream(new
                                FileInputStream(nomeFile));
            System.out.println("Lettura dei numeri non negativi");
            System.out.println("nel file " + nomeFile);

            int unIntero = inputStream.readInt();
            while (unIntero >= 0) {
                System.out.println(unIntero);
                unIntero = inputStream.readInt();
            }
            System.out.println("Fine della lettura dal file.");
            inputStream.close();
        } catch (FileNotFoundException e) {
            System.out.println("Errore nell'apertura del file " + nomeFile);
```

```
        } catch (EOFException e) {
            System.out.println("Errore nella lettura del file " + nomeFile);
            System.out.println("Raggiunta la fine del file.");
        } catch (IOException e) {
            System.out.println("Errore nella lettura del file " + nomeFile);
        }
    }
}
```

Esempio di output

```
Lettura dei numeri non negativi
nel file numeri.dat
1
2
3
Fine della lettura dal file.
```

Si noti che il valore sentinella -1 viene letto dal file ma non viene mostrato sullo schermo.

Leggere da un file binario

Sintassi

```
try {
    // Aprire il file
    ObjectInputStream nome_stream_di_input =
            new ObjectInputStream(new FileInputStream(nome_file));
    // Leggere dal file utilizzando istruzioni della forma:
    nome_stream_di_input.nome_metodo(argomento);  // Si veda la Figura 14.7
    // Chiudere il file
    nome_stream_di_input.close();
} catch (FileNotFoundException e) {
    Istruzioni_per_la_gestione_dell'eccezione
} catch (EOFException e) {
    Istruzioni_per_la_gestione_dell'eccezione
} catch (IOException e) {
    Istruzioni_per_la_gestione_dell'eccezione
}
```

Esempio

Si veda il Listato 14.7.

`FileInputStream` e `FileOutputStream`

In questo testo, le classi `FileInputStream` e `FileOutputSream` vengono utilizzate solamente per sfruttarne i costruttori. Entrambe le classi hanno un costruttore che accetta come argomento un nome di file sotto forma di stringa e si utilizzano tali costruttori per produrre degli oggetti da passare come argomenti ai costruttori delle classi di stream, come `ObjectInputStream` e `ObjectOutputStream`, che invece non accettano direttamente un nome di file come argomento. Di seguito sono riportati due esempi di questo utilizzo di `FileInputStream` e `FileOutputStream`:

```
ObjectInputStream fileInput = new ObjectInputStream(
                                    new FileInputStream("datigrezzi.dat"));

ObjectOutputStream fileOutput = new ObjectOutputStream(
                                    new FileOutputStream("datielaborati.dat"));
```

Istruzioni simili a queste sono state utilizzate rispettivamente nei Listati 14.6 e 14.7.

I costruttori delle classi `FileInputStream` e `FileOutputStream` possono generare un'eccezione di tipo `FileNotFoundException`, che è una sottoclasse di `IOException`.

Utilizzare `ObjectInputStream` per leggere un file di testo

I dati nei file binari sono codificati in maniera diversa rispetto ai file di testo. Pertanto, uno stream che si aspetta di leggere un file binario, come uno stream della classe `ObjectInputStream`, avrà problemi nel leggere un file di testo. Se si cerca di leggere un file di testo utilizzando uno stream della classe `ObjectInputStream`, il programma leggerà "valori spazzatura" o incorrerà in qualche altro tipo di errore. Analoghi problemi si incontreranno cercando di utilizzare la classe `Scanner` per leggere un file binario come se fosse un file di testo.

14.4.6 La classe EOFException

Molti dei metodi per la lettura da file binari generano una `EOFException` quando cercano di leggere oltre la fine del file. Come mostrato nel Listato 14.8, la classe `EOFException` può essere utilizzata per verificare se è stata raggiunta la fine del file quando si usa un `ObjectInputStream`. Nell'esempio, le istruzioni che leggono dal file sono inserite in un ciclo `while` la cui espressione di controllo è la costante `true`. Nonostante sembri un ciclo infinito, in realtà terminerà: quando viene raggiunta la fine del file, viene generata un'eccezione, così che il blocco `try` viene terminato e il controllo è ceduto al blocco `catch`.

È istruttivo confrontare il programma del Listato 14.8 con quello del Listato 14.7. Quest'ultimo controlla se è stata raggiunta la fine del file cercando un numero negativo. Questo approccio è valido, ma implica che non si possano salvare nel file numeri negativi, a meno che non siano utilizzati come valori sentinella. Il programma del Listato 14.8, invece, controlla se è stata raggiunta la fine del file utilizzando una `EOFException` e quindi può gestire file che contengano numeri interi di qualunque tipo, negativi compresi.

LISTATO 14.8 Utilizzo della classe `EOFException`.

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;
```

Presuppone che sia già stato eseguito il programma del Listato 14.6.

```
public class EOFExceptionDemo {
    public static void main(String[] args) {
        String nomeFile = "numeri.dat";
        try {
            ObjectInputStream inputStream = new ObjectInputStream(new
                                FileInputStream(nomeFile));
            System.out.println("Lettura di TUTTI gli interi");
            System.out.println("nel file " + nomeFile);

            try {
                while (true) {
                    int unIntero = inputStream.readInt();
                    System.out.println(unIntero);
                }
            } catch (EOFException e) {
                System.out.println("Fine della lettura dal file.");
            }
            inputStream.close();
        } catch (FileNotFoundException e) {
            System.out.println("Errore nell'apertura del file " + nomeFile);
        } catch (IOException e) {
            System.out.println("Errore nella lettura del file " + nomeFile);
        }
    }
}
```

Il ciclo termina quando viene generata un'eccezione.

Esempio di output

```
Lettura di TUTTI gli interi
nel file numeri.dat
1
2
3
-1
Fine della lettura dal file.
```

Quando si utilizza una `EOFException` per terminare la lettura di un file, si possono leggere file che contengono interi di ogni tipo, come, in questo caso, il `-1`, che viene trattato come qualunque altro intero.

La classe `EOFException`

Quando si leggono dati da un file binario utilizzando i metodi della classe `ObjectInputStream` mostrati nella Figura 14.7, se si cerca di leggere oltre la fine del file viene generata una `EOFException`. Questa eccezione può essere sfruttata per terminare un ciclo che legga i dati dal file.

La classe `EOFException` è derivata da `IOException`.

Controllare sempre se è stata raggiunta la fine di un file

Se un programma cerca di leggere oltre la fine di un file, si avranno problemi. Cosa accadrà esattamente dipenderà da come è scritto il programma: il programma potrebbe entrare in un ciclo infinito o terminare in modo anomalo. Quindi è opportuno assicurarsi sempre che un programma controlli se è stata raggiunta la fine di un file e agisca in modo corretto quando ciò accade. Anche quando si ritiene che il programma non cercherà di leggere oltre la fine del file, è bene considerare comunque questa eventualità, nel caso si verificasse qualche imprevisto.

Controllare se è stata raggiunta la fine di un file in modo errato

Metodi diversi (solitamente posti in classi diverse) per la lettura di dati da file controllano se è stata raggiunta la fine del file in modo diverso. Alcuni generano un'eccezione di tipo `EOFException` quando cercano di leggere oltre la fine del file. Altri restituiscono invece un valore speciale, come `null`. Quando si leggono dati da un file, bisogna fare attenzione a controllare se è stata raggiunta la fine del file nel modo corretto, a seconda del metodo che si sta utilizzando. Se si effettua il controllo nel modo sbagliato, probabilmente si verificherà una di queste due possibilità: il programma entrerà in un ciclo infinito imprevisto, o terminerà in modo anomalo.

Non tutti i metodi genereranno una `EOFException` cercando di leggere oltre la fine di un file. Per le classi presentate in questo testo, la regola generale è la seguente: se il programma sta leggendo dati da un file binario, genererà una `EOFException`. Se invece sta leggendo da un file di testo, una volta raggiunta la fine del file restituirà un qualche valore speciale, come `null`, e non genererà alcuna `EOFException`.

Controllare se è stata raggiunta la fine di un file binario

È possibile leggere tutti i dati contenuti in un file binario in uno dei due modi seguenti:

- individuando un valore sentinella scritto alla fine del file;
- gestendo una `EOFException`.

ESEMPIO DI PROGRAMMAZIONE
ELABORAZIONE DEI DATI DI UN FILE BINARIO

Il Listato 14.9 contiene un programma che svolge delle semplici elaborazioni di dati. Il programma richiede all'utente i nomi di due file, legge dei numeri dal primo file, li raddoppia e scrive i risultati nel secondo file. Le operazioni da svolgere sono semplici, ma il programma mostra l'utilizzo di varie tecniche di uso comune nella gestione dell'I/O

yLab

eo 14.2
vere e
gere un
binario

con file. In particolare, si noti che le variabili che referenziano gli oggetti di tipo stream connessi ai file sono variabili di istanza, e che le operazioni da svolgere sono suddivise tra più metodi.

Nell'esempio, i blocchi `try` sono stati mantenuti di piccole dimensioni, così che quando viene generata un'eccezione, essa è gestita da un blocco `catch` vicino all'istruzione che l'ha generata. Se si utilizzassero pochi blocchi `try` più lunghi, sarebbe più difficile capire in quale punto del codice è stata generata l'eccezione.

yLab

LISTATO 14.9 Elaborazione di un file binario.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Scanner;

public class Raddoppia {
    private ObjectInputStream inputStream = null;
    private ObjectOutputStream outputStream = null;

    /**
    Raddoppia gli interi in un file e scrive il risultato in un altro file.
    */
    public static void main(String[] args) {
        Raddoppia moltiplicatore = new Raddoppia();
        moltiplicatore.collegaFileDiInput();
        moltiplicatore.collegaFileDiOutput();
        moltiplicatore.moltiplicaPerDue();
        moltiplicatore.chiudiFile();

        System.out.println("Numeri letti da un file di input");
        System.out.println("raddoppiati e copiati in un file di output.");
    }

    public void collegaFileDiInput() {
        String nomeFileDiInput =
                        leggiNomeFile("Inserisci il nome del file di input:");
        try {
            inputStream = new ObjectInputStream(new
                                        FileInputStream(nomeFileDiInput));
        } catch (FileNotFoundException e) {
            System.out.println("File " + nomeFileDiInput + " non trovato.");
            System.exit(0);
        } catch (IOException e) {
            System.out.println("Errore nell'apertura del file di input " +
                                    nomeFileDiInput);
            System.out.println(e.getMessage());
```

```
        System.exit(0);
    }
}

private String leggiNomeFile(String messaggio) {
    String nomeFile = null;
    System.out.println(messaggio);
    Scanner tastiera = new Scanner(System.in);
    nomeFile = tastiera.next();
    return nomeFile;
}

public void collegaFileDiOutput() {
    String nomeFileDiOutput =
                    leggiNomeFile("Inserisci il nome del file di output:");
    try {
        outputStream = new ObjectOutputStream(new
                        FileOutputStream(nomeFileDiOutput));
    } catch (IOException e) {
        System.out.println("Errore nell'apertura del file di output " +
                                    nomeFileDiOutput);
        System.out.println(e.getMessage());
        System.exit(0);
    }
}
```

Un programma utile in un caso reale trasformerebbe probabilmente i dati in modo più complicato prima di scriverli nel file di output e avrebbe quindi, probabilmente, dei metodi aggiuntivi per farlo.

```
public void moltiplicaPerDue() {
    try {
        while (true) {
            int prossimo = inputStream.readInt();
            outputStream.writeInt(2 * prossimo);
        }
    } catch (EOFException e) {
        // Non fa niente, serve solo per terminare il ciclo.
    } catch (IOException e) {
        System.out.println("Errore nella lettura o
                            nella scrittura dei file.");
        System.out.println(e.getMessage());
        System.exit(0);
    }
```

```
            System.out.println("Errore nella chiusura dei file.");
            System.out.println(e.getMessage());
            System.exit(0);
        }
    }
}
```

Eccezioni, eccezioni, eccezioni

Molte operazioni che non generano eccezioni quando coinvolgono file di testo, ne generano invece quando riguardano file binari. Quindi, quando si lavora con file binari è necessario dedicare più spazio alla gestione delle eccezioni rispetto a quando si lavora con file di testo. Per esempio, la chiusura di un file di testo collegato a uno stream di tipo `PrintWriter` non genera mai eccezioni, mentre la chiusura di un file binario può generare una `IOException`. Come si è visto, lavorando con file binari praticamente qualunque operazione può generare eccezioni. Anche i metodi `writeObject` e `readObject` possono generare una lunga lista di eccezioni, come mostrato nelle Figure 14.6 e 14.7.

14.5 I/O su file binari di oggetti e array

In questo paragrafo si discute l'I/O per file binari in relazione a oggetti e array (che, come si ricorderà, sono anch'essi, in realtà, oggetti). Per fare questo, si utilizzeranno le classi `ObjectInputStream` e `ObjectOutputStream`.

14.5.1 I/O binario con oggetti di tipo classe

Si è già visto come leggere e scrivere nei file binari valori di tipo primitivo e stringhe. Come si possono leggere e scrivere oggetti di tipo diverso? Ovviamente, si potrebbero salvare in un file i valori delle variabili di istanza dell'oggetto e ricostruire in qualche modo un nuovo oggetto a partire dai dati salvati quando si legge il file. Tuttavia, poiché le stesse variabili di istanza potrebbero essere oggetti con a loro volta altri oggetti come variabili di istanza e così via, procedere in questo modo potrebbe rivelarsi molto complicato.

Fortunatamente, Java offre un modo semplice, chiamato **serializzazione degli oggetti**, per rappresentare un oggetto sotto forma di una sequenza di byte che possono essere scritti in un file binario. Questa procedura avviene automaticamente per gli oggetti delle classi **serializzabili**. Rendere serializzabile una classe è molto semplice: basta aggiungere la specifica `implements Serializable` all'intestazione della definizione della classe, come in questo esempio:

```
public class Specie implements Serializable
```

In realtà, occorre fare attenzione ad alcuni dettagli relativi alle variabili di istanza di una classe, ma dato che tali dettagli non saranno rilevanti nel primo esempio che verrà riportato, si rimanderà la loro discussione al prossimo paragrafo, nel quale si analizzerà anche più a fondo il significato della serializzazione.

L'interfaccia `Serializable` fa parte della libreria standard Java che appartiene al package `java.io`. Questa interfaccia è vuota, quindi non ci sono metodi aggiuntivi che debbano essere implementati. Anche se un'interfaccia vuota può sembrare del tutto inutile, il fatto di specificare che una classe la implementa indica a Java di rendere la classe serializzabile. Per rendere l'interfaccia accessibile al programma, si usa la seguente istruzione `import`:

```
import java.io.Serializable;
```

Nel Listato 14.10, la classe `Specie`, definita nel Listato 8.14 del Capitolo 8, è stata resa serializzabile e le sono stati aggiunti dei costruttori e un metodo `toString`. Questa classe sarà ora utilizzata per illustrare la lettura e la scrittura di oggetti nei file binari.

La scrittura di oggetti di classi serializzabili in un file binario avviene per mezzo del metodo `writeObject` della classe `ObjectOutputStream`, mentre la lettura può essere effettuata tramite il metodo `readObject` della classe `ObjectInputStream`. Il Listato 14.11 mostra un esempio. Per scrivere un oggetto della classe `Specie` in un file binario, si passa l'oggetto come argomento al metodo `writeOject`, come in

```
outputStream.writeObject(unaSpecie);
```

dove `outputStream` è uno stream di tipo `ObjectOutputStream` e `unaSpecie` è un'istanza di `Specie`.

Un oggetto scritto mediante il metodo `writeObject` può essere letto utilizzando il metodo `readObject` della classe `ObjectInputStream`, come mostrato nel seguente esempio tratto dal Listato 14.11:

```
letta = (Specie)inputStream.readObject();
```

Qui, `inputStream` è uno stream di tipo `ObjectInputStream` collegato al file nel quale erano stati precedentemente scritti dati con il metodo `writeObject` della classe `ObjectOutputStream`. I dati consistono in questo caso di oggetti di tipo `Specie`, e quindi la variabile `letta` è di tipo `Specie`. Si noti che il metodo `readObject` restituisce un oggetto di tipo `Object`, quindi è necessario convertirlo esplicitamente nel tipo corretto, in questo caso `Specie`.

Prima di proseguire, è opportuno chiarire un punto spesso interpretato erroneamente. La classe `Specie` ha un metodo `toString`, necessario per produrre un output leggibile quando si scrive su schermo o in un file di testo utilizzando il metodo `println`. Tuttavia, il metodo `toString` non ha nulla a che fare con l'I/O di oggetti su file binari, che funzionerebbe correttamente anche se la classe non avesse il metodo `toString`.

Scrivere oggetti in un file binario

Le istanze di una classe possono essere scritte in un file binario utilizzando il metodo `writeObject` solo se la classe è serializzabile. Per verificare se una classe della libreria standard è serializzabile, è sufficiente controllare nella documentazione se la classe implementa l'interfaccia `Serializable`.

LISTATO 14.10 La classe `Specie` resa serializzabile per l'I/O su file binari.

```
import java.io.Serializable;
import java.util.Scanner;

/**
Classe serializzabile per descrivere le specie a rischio.
*/
public class Specie implements Serializable {
    private String nome;
    private int popolazione;
    private double tassoCrescita;

    public Specie(String nomeIniziale, int popolazioneIniziale,
                          double tassoCrescitaIniziale) {
        nome = nomeIniziale;
        if (popolazioneIniziale >= 0)
            popolazione = popolazioneIniziale;
        else {
            System.out.println("ERRORE: Popolazione negativa.");
            System.exit(0);
        }
        tassoCrescita = tassoCrescitaIniziale;
    }

    public Specie() {
        this(null, 0, 0);
    }

    public String toString() {
        return ("Nome = " + nome + "\n" + "Popolazione = " + popolazione +
                "\n" + "Tasso di crescita = " + tassoCrescita + "%");
    }

    <Gli altri metodi sono gli stessi del Listato 8.14 del Capitolo 8,
     ma non serviranno per la discussione in questo capitolo.>
}
```

Questa è una versione migliorata della classe `Specie` e sostituisce la definizione del Listato 8.14 nel Capitolo 8.

Queste due parole è l'istruzione che importa l'interfaccia `Serializable` rendono la classe serializzabile.

LISTATO 14.11 I/O su file binari di oggetti di tipo classe.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class IOOggettiClasseDemo {
    public static void main(String[]args) {
        String nomeFile = "specie.registrazioni";
```

```
        ObjectOutputStream outputStream = null;
        try {
            outputStream = new ObjectOutputStream(new
                                FileOutputStream(nomeFile));
    } catch (IOException e) {
        System.out.println("Errore nell'apertura del file di output " +
                           nomeFile + ".");
        System.exit(0);
    }

    Specie condorCalifornia = new
                              Specie("Condor della California", 27, 0.02);
    Specie rinoceronteNero = new
                             Specie("Rinoceronte Nero", 100, 1.0);

    try {
        outputStream.writeObject(condorCalifornia);
        outputStream.writeObject(rinoceronteNero);
        outputStream.close();
    } catch (IOException e) {
        System.out.println("Errore nella scrittura del file "
                           + nomeFile + ".");
        System.exit(0);
    }
    System.out.println("Le registrazioni sono state scritte nel file " +
                       nomeFile + ".");
    System.out.println("Ora il file verrà riaperto e verranno mostrate "
                       + "le registrazioni.");
    ObjectInputStream inputStream = null;
    try {
        inputStream = new ObjectInputStream(new FileInputStream(nomeFile));
    } catch (IOException e) {
        System.out.println("Errore nell'apertura del file di input "
                           + nomeFile + ".");
        System.exit(0);
    }

    Specie lettaUno = null, lettaDue = null;
    try {
        lettaUno = (Specie)inputStream.readObject();
        lettaDue = (Specie)inputStream.readObject();
        inputStream.close();

    } catch (Exception e) {
        System.out.println("Error nella lettura del file " + nomeFile + ".");
        System.exit(0);
    }

    System.out.println("Sono stati letti dal file " + nomeFile +
                       "\ni seguenti dati.");
```

Si notino le conversioni di tipo esplicite.

Sarebbe meglio utilizzare un blocco `catch` separato per ogni tipo di eccezione. Qui ne è stato utilizzato uno solo per ragioni di spazio.

```
        System.out.println(lettaUno);
        System.out.println();
        System.out.println(lettaDue);
        System.out.println("Fine del programma.");
    }
}
```

Esempio di output

```
Le registrazioni sono state scritte nel file specie.registrazioni.
Ora il file verrà riaperto e verranno mostrate le registrazioni.
Sono stati letti dal file specie.registrazioni
i seguenti dati.
Nome = Condor della California
Popolazione = 27
Tasso di crescita = 0.02%

Nome = Rinoceronte Nero
Popolazione = 100
Tasso di crescita = 1.0%
Fine del programma.
```

14.5.2 Alcuni dettagli sulla serializzazione

L'introduzione alla serializzazione presentata nel paragrafo precedente ha omesso alcuni dettagli che devono essere ora affrontati per poter ottenere una definizione completa di cos'è una classe serializzabile. Si ricordi che è stato sottolineato come una variabile di istanza possa essere essa stessa un oggetto che ha altri oggetti come variabili di istanza. Quando una classe serializzabile ha delle variabili di istanza di tipo classe, anche le classi delle variabili di istanza devono essere serializzabili, e così via per tutti i livelli di variabili di istanza nelle classi.

Quindi, affinché una classe sia serializzabile, devono essere verificate tutte le seguenti condizioni:

- la classe implementa l'interfaccia `Serializable`;
- tutte le variabili di istanza di tipo classe sono istanze di classi serializzabili;
- la superclasse diretta della classe, se esiste, è serializzabile o definisce un costruttore di default.

Per esempio, la classe `Specie` implementa l'interfaccia `Serializable` e ha una variabile di istanza di tipo `String`, e la classe `String` è serializzabile. Poiché qualunque classe derivata da una classe serializzabile è serializzabile, una classe che estenda `Specie` sarà anch'essa serializzabile.

Qual è l'effetto del rendere serializzabile una classe? Da un certo punto di vista, non c'è alcun effetto diretto sulla classe, ma solo su come Java gestisce l'I/O su file con gli oggetti di quella classe. Se una classe è serializzabile, Java assegna un numero di serie a ogni oggetto della classe che viene scritto in uno stream di tipo `ObjectOutputStream`. Se lo stesso oggetto viene scritto nello stream più volte, dopo la prima scrittura Java riporta solo il numero di serie dell'oggetto, invece di riscrivere l'oggetto più volte. Ciò rende le operazioni di I/O più efficienti e riduce le dimensioni dei file. Quando un file così costruito viene letto tramite uno stream di tipo `ObjectInputStream`, i numeri di

serie duplicati vengono restituiti come riferimenti allo stesso oggetto. Si noti che, di conseguenza, se due variabili contengono riferimenti allo stesso oggetto e vengono scritte in un file, quando in seguito si legge il file, i due oggetti ottenuti si riferiranno in realtà allo stesso oggetto. Quindi scrivendo gli oggetti in un file e poi rileggendo il file non si perde nulla della struttura originale.

La serializzabilità sembra una caratteristica molto utile. Perché allora le classi non vengono rese tutte serializzabili? In certi casi è per motivi di sicurezza. Il sistema basato sui numeri di serie rende infatti semplice per i programmatori accedere agli oggetti salvati su memorie secondarie. In altri casi, scrivere oggetti in memoria secondaria potrebbe non avere alcun senso, magari perché i dati sarebbero privi di significato se letti in un secondo tempo. Per esempio, se gli oggetti contengono informazioni sullo stato attuale di un sistema, potrebbero non avere significato successivamente.

14.5.3 Array nei file binari

Dato che Java tratta gli array come oggetti, è possibile utilizzare il metodo `writeObject` per scrivere un intero array in un file binario e successivamente rileggerlo tramite il metodo `readObject`. Quando si fa questo, se il tipo base dell'array è una classe, questa deve essere serializzabile. Quindi, se tutti i dati il cui tipo è una classe serializzabile sono organizzati in un array, è possibile scriverli tutti in un file binario tramite una singola chiamata a `writeObject`.

Per esempio, si supponga che `gruppo` sia un array di oggetti di tipo `Specie`. Se `fileDaScrivere` è un'istanza di `ObjectOutputStream` associata a un file binario, si può scrivere l'array nel file utilizzando l'istruzione

```
fileDaScrivere.writeObject(gruppo);
```

Una volta che il file è stato così costruito, si può rileggere l'array mediante l'istruzione

```
Specie[] unArray = (Specie[])fileDaLeggere.readObject();
```

dove `fileDaLeggere` è un'istanza di `ObjectInputStream` associata al file appena creato.

Si noti che la classe che costituisce il tipo base dell'array, `Specie`, è serializzabile. Si noti inoltre la necessità di effettuare una conversione di tipo esplicita leggendo il file dall'array. Dato che `readObject` restituisce un valore di tipo `Object`, è necessario convertirlo al tipo corretto, in questo caso `Specie[]`.

MyLab

Il Listato 14.12 contiene un semplice programma che scrive e legge un array in un file binario. Si noti che gli array `unArray` e `unAltroArray` sono stati definiti all'esterno dei blocchi `try`, in modo che entrambi esistano al di fuori di tali blocchi. Si noti anche che `unAltroArray` è inizializzato a `null`, e non a `new Specie[2]`. Infatti, se si allocasse un nuovo array in quel punto, esso sarebbe poi sostituito dalla successiva chiamata a `readObject`. In altre parole, `readObject` crea un nuovo array e non si limita a inserire dati in un array già esistente.

Video 14.3 Utilizzare un file binario di oggetti e array

LISTATO 14.12 I/O su file di array.

MyLab

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
```

```
public class IOArrayDemo {
    public static void main(String[] args) {
        Specie[] unArray = new Specie[2];
        unArray[0] = new Specie("Condor della California", 27, 0.02);
        unArray[1] = new Specie("Rinoceronte Nero", 100, 1.0);
        String nomeFile = "array.dat";

        try {
            ObjectOutputStream outputStream = new ObjectOutputStream(new
                                FileOutputStream(nomeFile));
            outputStream.writeObject(unArray);
            outputStream.close();
        } catch (IOException e) {
            System.out.println("Errore nella scrittura del file " +
                               nomeFile + ".");
            System.exit(0);
        }

        System.out.println("L'array è stato scritto nel file " + nomeFile +
                           " e il file è stato chiuso.");
        System.out.println("Ora il file verrà riaperto
                           e verrà stampato l'array.");

        Specie[] unAltroArray = null;
        try {
            ObjectInputStream inputStream = new ObjectInputStream(new
                                FileInputStream(nomeFile));
            unAltroArray = (Specie[])inputStream.readObject();
            inputStream.close();

        } catch (Exception e) {
            System.out.println("Errore nella lettura del file " +
                               nomeFile + ".");
            System.exit(0);
        }

        System.out.println("I seguenti dati sono stati letti dal file " +
                           nomeFile + ":");
        for (int i = 0; i < unAltroArray.length; i++) {
            System.out.println(unAltroArray[i]);
            System.out.println();
        }
        System.out.println("Fine del programma.");
    }
}
```

Si noti la conversione di tipo esplicita.

Sarebbe meglio utilizzare un blocco catch separato per ogni tipo di eccezione. Qui ne è stato utilizzato uno solo per ragioni di spazio.

Esempio di output

```
L'array è stato scritto nel file array.dat e il file è stato chiuso.
Ora il file verrà riaperto e verrà stampato l'array.
```

```
I seguenti dati sono stati letti dal file array.dat:
Nome = Condor della California
Popolazione = 27
Tasso di crescita = 0.02%

Nome = Rinoceronte Nero
Popolazione = 100
Tasso di crescita = 1.0%

Fine del programma.
```

14.6 Riepilogo

- I file che vengono interpretati come sequenze di caratteri dai programmi Java e dagli editor di testo sono detti file di testo. Tutti gli altri file sono detti file binari.
- Si può utilizzare la classe `PrintWriter` per scrivere in un file di testo e la classe `Scanner` o la classe `BufferedReader` per leggere da un file di testo.
- Quando si legge un file, bisogna sempre verificare se è stata raggiunta la fine del file e in tal caso eseguire le operazioni appropriate. Il modo in cui si può verificare se è stata raggiunta la fine del file dipende dal tipo di file (di testo o binario) che si sta leggendo.
- Il nome di un file può essere letto da tastiera in una variabile di tipo `String`.
- La classe `File` può essere utilizzata per controllare se esiste un file con un dato nome. Può inoltre essere sfruttata per verificare se il programma ha i permessi per leggere o scrivere nel file.
- Per la scrittura nei file binari si può usare la classe `ObjectOutputStream`, mentre per la lettura dei file binari è disponibile la classe `ObjectInputStream`.
- È possibile utilizzare il metodo `writeObject` della classe `ObjectOutputStream` per scrivere oggetti di tipo classe o array in un file binario. Oggetti e array possono essere letti da un file binario tramite il metodo `readObject` della classe `ObjectInputStream`.
- Affinché si possano utilizzare i metodi `writeObject` della classe `ObjectOutputStream` e `readObject` della classe `ObjectInputStream`, ogni classe le cui istanze vengono scritte nel file deve implementare l'interfaccia `Serializable`.

14.7 Esercizi

1. Scrivere un programma che scriva in un file di testo il Discorso di Gettysburg. Si scriva ogni frase del discorso in una linea a parte del file.
2. Modificare il programma dell'esercizio precedente in modo che legga il nome del file da utilizzare dalla tastiera.

3. Si scriva del codice che chieda all'utente di inserire una tra le parole `aggiungi` e `nuovo`. A seconda della risposta dell'utente, si apra un file già esistente per aggiungervi altri dati o si crei un file nuovo vuoto per l'inserimento dei dati. In entrambi i casi, si supponga che il nome del file da utilizzare sia contenuto nella variabile `nomeFile`.
4. Si scriva un programma che registri gli acquisti effettuati in un negozio. Per ogni acquisto, il programma dovrà leggere da tastiera il nome del prodotto, il prezzo e la quantità acquistata. Si calcoli il costo totale della merce acquistata (quantità per prezzo) e lo si scriva in un file di testo, mostrando anche sullo schermo il costo totale dei prodotti acquistati fino a questo momento. Una volta che sono stati registrati tutti gli acquisti, si scriva il costo totale sia sullo schermo che nel file. Dato che si vuole tenere traccia di tutti gli acquisti effettuati, i dati dovranno essere di volta in volta aggiunti alla fine del file.
5. Si modifichi la classe `CronometroGiri`, descritta nell'Esercizio 13 del Capitolo 13, come segue:
   - si aggiunga un attributo per uno stream sul quale scrivere i tempi;
   - si aggiunga un costruttore

     ```
     CronometroGiri(n, persona, nomeFile)
     ```

     per una gara da *n* giri. Il nome della persona e quello del file per la registrazione dei tempi sono passati al costruttore come stringhe. Occorrerà aprire il file e scrivervi il nome della persona. Nel caso in cui il file non possa essere aperto, si generi un'eccezione.
6. Si scriva una classe `NumeroDiTelefono` per gestire un numero di telefono. Un oggetto di questa classe dovrà avere i seguenti attributi:
   - `prefissoInternazionale` (un numero di due cifre)
   - `prefissoNazionale` (un numero di due cifre)
   - `numero` (un numero di sei cifre)

   e i metodi:
   - `NumeroDiTelefono(unaStringa)`: un costruttore che crea una nuova istanza della classe data una stringa nella forma `xx-xx-xxxxxx` o, se non viene specificato il prefisso internazionale, `xx-xxxxxx`. Si generi un'eccezione se il formato non è valido (*suggerimento*: per semplificare il costruttore, si possono sostituire i trattini con degli spazi. Per accettare un numero con dei trattini, si potrebbe scorrere la stringa un carattere alla volta o imparare a utilizzare la classe `Scanner` per leggere parole separate da un carattere, come il trattino, diverso dallo spazio).
   - `toString`: restituisce una stringa in uno dei formati descritti per il costruttore.

   Utilizzando un editor di testi, si crei un file di testo contenente alcuni numeri di telefono nei due formati precedentemente descritti. Si scriva poi un programma che legga il file, mostri i numeri sullo schermo e li inserisca in un array avente tipo base `NumeroDiTelefono`. Si consenta all'utente di aggiungere o eliminare un numero di telefono. Si scrivano i dati così modificati nel file, sostituendo il contenuto originale. Infine, si leggano e si visualizzino i numeri nel file modificato.

7. Si scriva una classe `InformazioniContatto` per salvare le informazioni su di una persona. La classe dovrebbe avere attributi per nome, numero di telefono dell'ufficio, numero di telefono di casa, numero di cellulare, indirizzo e-mail e indirizzo di casa della persona. Dovrebbe inoltre avere un metodo `toString` che restituisca i dati formattati in una stringa, usando valori appropriati nel caso in cui alcuni valori non siano stati specificati. La classe dovrà avere un costruttore `InformazioniContatto(unaStringa)` che crei una nuova istanza della classe utilizzando i dati presenti nella stringa `unaStringa`. Il costruttore dovrà utilizzare, per il parametro, un formato compatibile con quello prodotto dal metodo `toString`.

   Utilizzando un editor di testi, si crei un file di testo di informazioni relative a varie persone, come descritto sopra. Si scriva poi un programma che legga il file, mostri i dati sullo schermo e crei un array di tipo base `InformazioniContatto`. Si permetta all'utente di svolgere una delle seguenti operazioni: modificare i dati di un contatto, aggiungere un contatto o eliminarne uno. Infine, si sovrascriva il file con i dati modificati.

8. Si scriva un programma che legga ogni riga di un file di testo, rimuova la prima parola da ogni riga e scriva le righe modificate in un nuovo file di testo.

9. Si ripeta l'esercizio precedente scrivendo le nuove righe in un file binario anziché in un file di testo.

10. Scrivere un programma che copi un file di testo riga per riga. I nomi del file originale e di quello nuovo dovranno essere letti da tastiera. Si usino i metodi della classe `File` per verificare se il file originale esiste e può essere letto. Se il file non esiste o non può essere letto, si mostri un messaggio di errore e si termini il programma. Analogamente, si controlli se il file destinazione esiste già e in tal caso si mostri un avvertimento e si chieda all'utente se terminare il programma, proseguire sovrascrivendo il file esistente oppure inserire un altro nome per il file destinazione.

11. Si supponga di avere un file di testo che contiene nomi completi di persone. Ogni nome completo è costituito da nome e cognome. Sfortunatamente, il programmatore che ha creato il file non si è assicurato che ogni nome fosse scritto interamente in una singola riga, con una riga per nome. Si legga il file e si scrivano i nomi in un nuovo file, uno per riga, nel modo corretto. Per esempio, se il file originale contenesse le righe

```
Mario Rossi Federico
Bianchi Luca
Verdi
Alberto
Neri
```

    il risultato dovrebbe essere

```
Mario Rossi
Federico Bianchi
Luca Verdi
Alberto Neri
```

12. Si supponga di avere un file binario che contiene numeri di tipo `int` o `double`. Non si conosce l'ordine con il quale i numeri sono stati inseriti nel file, ma si sa che tale ordine è stato riportato in una stringa all'inizio del file. La stringa è composta

dalle lettere *i* (per `int`) e *d* (per `double`) nell'ordine corrispondente a quello con il quale sono stati inseriti i numeri. La stringa è stata scritta utilizzando il metodo `writeUTF`.

Per esempio, la stringa `"iddiiddd"` indica che il file contiene otto valori: un `int`, seguito da due `double`, seguiti da due `int`, seguiti da tre `double`. Si legga il file binario e si crei un nuovo file di testo con i valori scritti uno per riga.

13. Si supponga di voler salvare in un file binario dell'audio digitalizzato. Un segnale audio tipicamente cambia poco tra un campione e l'altro. In tal caso, si occuperebbe meno memoria salvando solo la variazione rispetto al valore precedente anziché i dati veri e propri. In questo esercizio si sfrutterà questa idea.

    Si scriva un programma `SalvaSegnale` che legga degli interi positivi, ognuno differente dal precedente di non più di 127 unità in eccesso o in difetto, dalla tastiera (o da un file di testo, se si preferisce). Si scriva il primo intero in un file binario. Per ogni intero successivo, si calcoli la differenza con quello precedente, si converta tale differenza in un byte e si scriva il risultato nel file binario. Si interrompa l'operazione quando si trova un numero negativo.

14. Si scriva un programma `RecuperaSegnale` che legga il file binario scritto dal programma dell'esercizio precedente. Si mostrino i valori numerici sullo schermo.

15. Anche se un file binario non è un file di testo, può contenere del testo codificato. Per scoprire se un file ha questa caratteristica, si scriva un programma che apra un file binario e lo legga un byte alla volta. Si mostrino il valore intero di ciascun byte e il carattere corrispondente, se esiste, nella codifica ASCII.

    Dettagli tecnici: per convertire un byte in un intero, si usi l'istruzione

    ```
    char[] arrayChar = Character.toChars(valoreByte);
    ```

    L'argomento `valoreByte` del metodo `toChars` è un `int` il cui valore è pari a quello del byte letto dal file. Il carattere rappresentato dal byte sarà `arrayChar[0]`. Poiché un intero è composto da quattro byte, `valoreByte` può rappresentare quattro caratteri. Il metodo `toChars` prova a convertire ognuno dei quattro byte in un carattere e inserisce i risultati in un array di `char`. In questo caso, il carattere interessante è quello nella posizione `0`. Se un byte nel file non corrisponde a un carattere, il metodo genererà una `IllegalArgumentException`. Se viene generata un'eccezione di questo tipo, si mostri solo il valore numerico del byte e si prosegua con quello successivo.

14.8 Progetti

1. Scrivere un programma che esegua una ricerca in un file contenente numeri e mostri il massimo, il minimo e la media dei numeri contenuti nel file. Non si assuma che i numeri siano stati scritti nel file in un ordine particolare. Il programma deve chiedere il nome del file all'utente. Si utilizzi un file di testo o un file binario. Nel caso del file di testo, si supponga che i numeri siano stati scritti uno per riga. Nel caso del file binario, si utilizzino numeri di tipo `double` scritti utilizzando `writeDouble`.

2. Scrivere un programma che legga da un file dei numeri di tipo `int` e li scriva, senza duplicati, in un altro file. Si supponga che nel file di input i numeri siano ordinati in ordine crescente. Dopo che il programma è stato eseguito, il nuovo file dovrà contenere tutti i numeri di quello originale, ma nessuno di essi comparirà più di una volta. Anche nel nuovo file i numeri dovranno essere scritti in ordine crescente. Il programma deve chiedere all'utente di inserire i nomi di entrambi i file. Si utilizzino file di testo o file binari. Nel caso dei file di testo, si supponga che i numeri siano scritti uno per riga. Nel caso dei file binari, si usino numeri di tipo `int` scritti con `writeInt`.

3. Si scriva un programma che corregga alcuni problemi di formattazione e punteggiatura di un file di testo. Il programma dovrà chiedere all'utente di inserire i nomi di un file di input e di uno di output. Successivamente, dovrà copiare il testo dal file di input a quello di output con le seguenti modifiche: (1) ogni stringa composta da due o più spazi deve essere sostituita da uno spazio singolo; (2) tutte le frasi devono iniziare con una lettera maiuscola. Relativamente al punto (2), devono essere interpretate come frasi separate tutte quelle, oltre alla prima, che iniziano dopo un punto, un punto interrogativo o un punto esclamativo seguiti da uno spazio.

4. Scrivere un programma simile a quello del Listato 14.11 che scriva in un file binario un numero arbitrario di oggetti di tipo `Specie` (la classe `Specie` completa è stata definita nel Listato 8.16 del Capitolo 8). Si leggano il nome del file e i dati da utilizzare per costruire gli oggetti da un file di testo creato utilizzando un editor di testi. Poi si scriva un altro programma che effettui una ricerca nel file binario creato dal primo programma e mostri all'utente i dati relativi a ogni specie a rischio specificata. Il programma dovrà mostrare tutti i dati relativi alla specie o comunicare che la specie non è presente nel file. Si consenta all'utente di inserire nuovi nomi di specie o terminare l'esecuzione del programma.

5. Si scriva un programma che legga il file creato dal programma del progetto precedente e mostri sullo schermo i dati relativi alla specie con la popolazione meno numerosa e a quella con la popolazione più numerosa. Non si assuma che i dati siano stati salvati secondo un ordine particolare. Si chieda all'utente il nome del file da utilizzare.

6. Il Progetto 4 chiede, tra le altre cose, di scrivere un programma che crei un file binario contenente oggetti della classe `Specie`. Si scriva un programma che legga un file creato da quel programma e scriva in un nuovo file gli oggetti dopo aver modificato la popolazione di ogni specie con il valore che essa avrà dopo 100 anni. Si utilizzi il metodo `prediciPopolazione` della classe `Specie`, assumendo di conoscere il tasso di crescita di ogni specie.

7. I messaggi di testo sono un mezzo di comunicazione molto utilizzato. Nei messaggi si utilizzano spesso delle abbreviazioni che sarebbero però poco appropriate per comunicazioni più formali. Si supponga che tali abbreviazioni siano salvate, una per riga, in un file di testo chiamato `abbreviazioni.txt`. Per esempio, il file potrebbe contenere le righe seguenti:

```
lol
:)
:(
```

Si scriva un programma che legga un messaggio da un altro file di testo e racchiuda ogni abbreviazione in una coppia di parentesi angolari <>. Si scriva il testo risultante in un nuovo file.

Per esempio, se il messaggio da elaborare è

```
Ciao! Stai x andare al mare? Divertiti! :)
```

il nuovo testo sarà

```
Ciao! Stai <x> andare al mare? Divertiti! <:)>
```

8. Si modifichi la classe `NumeroDiTelefono` dell'Esercizio 6 in modo che sia serializzabile. Si scriva un programma che crei un array con tipo base `NumeroDiTelefono` leggendo i dati da tastiera. Si scriva l'array in un file binario utilizzando il metodo `writeObject`. Quindi si leggano i dati dal file utilizzando il metodo `readObject` e si mostrino i dati sullo schermo. Si consenta all'utente di modificare, aggiungere ed eliminare numeri di telefono finché non comunica che le modifiche sono terminate. Alla fine si scrivano i dati modificati sul file, sovrascrivendo quelli originali.
9. Si modifichi la classe `Animale`, definita nel Listato 9.1 del Capitolo 9, in modo che sia serializzabile. Si scriva un programma che consenta di scrivere e leggere da un file oggetti di tipo `Animale`. Il programma dovrà chiedere all'utente se intenda scrivere in un file o leggere da un file. In entrambi i casi, il programma dovrà chiedere all'utente il nome del file. Se l'utente ha richiesto di scrivere nel file, potrà inserire un numero arbitrario di registrazioni. Se invece ha chiesto di leggere dal file, il programma mostrerà tutte le registrazioni in esso contenute. Ci si assicuri che le registrazioni non scorrano tanto velocemente da non poter essere lette (*suggerimento*: si pensi a un modo per mettere in pausa il programma dopo che è stato mostrato un certo numero di righe).
10. Si scriva un programma che legga oggetti di tipo `Animale` da un file creato dal programma del progetto precedente e mostri sullo schermo il nome e il peso dell'animale più pesante, il nome e il peso di quello più leggero, il nome e l'età di quello più giovane e il nome e l'età di quello più vecchio.
11. Questo progetto riguarda il seguente indovinello: "Trovare una parola, a parte tremendo, orrendo e stupendo, che finisca in do". Si supponga di avere a disposizione il file di testo `parole.txt` che contenga tutte le parole della lingua italiana. Si scriva un programma che legga le parole dal file e stampi solo quelle che finiscono in "do".

Strutture dati dinamiche

OBIETTIVI

- Descrivere l'idea generale delle strutture dinamiche concatenate e la loro implementazione in Java.
- Gestire le liste concatenate (*linked list*).
- Gestire le tabelle di *hash*.
- Gestire gli insiemi.
- Gestire gli alberi.

Una struttura dati concatenata consiste di blocchi di dati, denominati **nodi** (*node*), connessi tra loro tramite dei **collegamenti** (*link*). I collegamenti possono essere visualizzati come frecce e interpretati come passaggi a senso unico da un nodo a un altro. Il tipo più semplice di struttura dati concatenata consiste in una singola catena di nodi, ognuno dei quali è collegato al successivo da un collegamento. Una struttura di questo tipo è chiamata lista **concatenata** (*linked list*).

Se si definisce una lista concatenata in Java, i nodi sono realizzati come oggetti di una classe nodo, mentre i collegamenti sono in genere realizzati sotto forma di riferimenti, cioè come variabili di istanza del tipo della classe nodo stessa. Quindi, in Java un nodo in una lista concatenata è collegato al nodo successivo tramite una variabile di istanza di tipo nodo contenente il riferimento al nodo successivo.

Java fornisce una classe `LinkedList`, che fa parte del package `java.util`. In molti casi ha senso utilizzare questa classe perché è stata progettata e verificata accuratamente. Tuttavia, limitarsi a utilizzare questa classe non consente di imparare a realizzare da zero strutture dati concatenate in Java. Pertanto, verrà presentata l'implementazione in Java di una lista concatenata semplificata.

Dopo aver discusso le liste concatenate, verranno presentate strutture dati dinamiche più complesse, come gli insiemi, le tabelle di *hash* e gli alberi.

Prerequisiti

È possibile saltare questo capitolo e leggere direttamente il Capitolo 16 su collezioni, mappe e iteratori.

Questo capitolo richiede la conoscenza di quanto presentato nei Capitoli da 1 a 4 e i Capitoli 8 e 9.

Il Paragrafo 15.1.9 richiede il Capitolo 13 che tratta la gestione delle eccezioni. Infine il Paragrafo 15.5 sugli alberi richiede anche il Capitolo 7 sulla ricorsione.

15.1 Liste concatenate

Una lista concatenata è una struttura dati costituita da una singola catena di nodi, ognuno dei quali è connesso al nodo successivo da un collegamento. Si tratta del tipo più semplice di struttura concatenata, ma è comunque molto utilizzato. In questo paragrafo verranno presentati esempi di liste concatenate e si spiegherà come implementarle ed utilizzarle in Java.

15.1.1 Generalità sulle liste concatenate

Una **lista concatenata** (*linked list*) è una struttura dati dinamica che collega l'uno all'altro gli elementi di una lista. La Figura 15.1 mostra un esempio di lista concatenata. Come tutte le strutture dati concatenate, anche una lista concatenata è composta da oggetti noti come **nodi** (*node*). Nella figura, i nodi sono rappresentati come rettangoli divisi da una linea orizzontale. In una parte del nodo sono contenuti i dati, nell'altra il **collegamento** (*link*) a un altro nodo. I collegamenti sono rappresentati come frecce che puntano al nodo cui sono collegati. In Java, i collegamenti sono implementati come riferimenti a un nodo e, in pratica, sono variabili di istanza del tipo del nodo. Tuttavia, per introdurre le liste concatenate, si può semplicemente pensare ai collegamenti come a frecce. In una lista concatenata, ogni nodo contiene solo un collegamento e i nodi sono posizionati uno dopo l'altro così da formare una lista, come nella Figura 15.1. Intuitivamente, il programma si muove da nodo a nodo, seguendo i collegamenti.

Il collegamento `testa` (*head*) non è nella lista dei nodi; infatti non è un nodo, ma un collegamento che punta al primo nodo. Nelle implementazioni, `testa` conterrà un riferimento a un nodo, così `testa` è una variabile del tipo di nodo. Un programma può facilmente muoversi attraverso la lista in ordine, dal primo nodo all'ultimo nodo, seguendo le "frecce".

Lista concatenata

Una lista concatenata è una struttura dati costituita da oggetti chiamati nodi. Ogni nodo può contenere dati e anche un riferimento a un altro nodo; in questo modo, i nodi si collegano a formare una lista, come illustrato nella Figura 15.1.

Ora si vedrà ora come implementare in Java una lista concatenata. Ogni nodo è un oggetto di una classe che ha due variabili di istanza: una per i dati e una per il collegamento. Il Listato 15.1 fornisce la definizione di una classe Java che può rappresentare i nodi di una lista concatenata come quella rappresentata nella Figura 15.1. In questo caso, i dati del

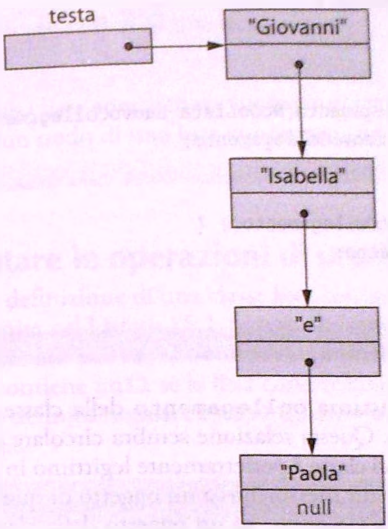


Figura 15.1 Una lista concatenata.

nodo sono costituiti da un valore di tipo `String`[1]. Come indicato nel Listato 15.1, questa classe nodo sarà resa privata.

LISTATO 15.1 Una classe nodo.

```
public class NodoLista {
    private String dati;
    private NodoLista collegamento;

    public NodoLista() {
        collegamento = null;
        dati = null;
    }

    public NodoLista(String valoreDati, NodoLista valoreCollegamento) {
        dati = valoreDati;
        collegamento = valoreCollegamento;
    }

    public void setDati(String nuoviDati) {
        dati = nuoviDati;
    }
```

Più avanti in questo capitolo, si nasconderà questa classe rendendola privata

[1] Tecnicamente parlando, il nodo non "contiene" la stringa, ma solo un riferimento a essa, come accade per ogni variabile di tipo `String`. Tuttavia, per gli scopi di questa trattazione, si può considerare il nodo come un contenitore della stringa.

```
    public String getDati() {
        return dati;
    }

    public void setCollegamento(NodoLista nuovoCollegamento) {
        collegamento = nuovoCollegamento;
    }

    public NodoLista getCollegamento() {
        return collegamento;
    }
}
```

Si noti che la variabile di istanza `collegamento` della classe `NodoLista` nel Listato 15.1 è di tipo `NodoLista`. Questa relazione sembra circolare e, in un certo senso, lo è. Questo tipo di definizione di classe è perfettamente legittimo in Java: una variabile di tipo classe, ha al proprio interno un riferimento a un oggetto di quella stessa classe. In pratica la variabile di istanza `collegamento` di un oggetto della classe `NodoLista` conterrà un riferimento a un altro oggetto sempre della classe `NodoLista`. Di conseguenza, come mostrano le frecce nel diagramma rappresentato nella Figura 15.1, ogni oggetto nodo di una lista concatenata contiene nella propria variabile di istanza `collegamento` un riferimento a un altro oggetto della classe `NodoLista`; questo altro oggetto contiene a sua volta un riferimento a un altro oggetto della classe `NodoLista` e così via, fino alla fine della lista concatenata.

Quando si utilizza una lista concatenata, il codice deve essere in grado di collocarsi sul primo nodo e poi di scoprire quando raggiunge l'ultimo nodo. Per raggiungere il primo nodo, si usa una variabile di tipo `NodoLista` che contiene un riferimento al primo nodo. Come si è detto in precedenza, questa variabile non è un nodo della lista. Nella Figura 15.1, la variabile contenente un riferimento al primo nodo è rappresentata da un rettangolo etichettato con `testa`. Il primo nodo in una lista concatenata è chiamato **nodo di testa** (*head node*) ed è consuetudine utilizzare il nome `testa` (*head*) per una variabile che contiene un riferimento a questo primo nodo. Infatti, `testa` è chiamato **riferimento di testa** (*head reference*).

In Java si indica la fine di una lista concatenata impostando a `null` la variabile di istanza `collegamento` dell'ultimo oggetto, come illustrato nella Figura 15.1. In questo modo, il programma può verificare se un nodo è l'ultimo in una lista concatenata semplicemente controllando se la variabile di istanza `collegamento` del nodo è `null`. Per controllare se una variabile contiene `null` si usa l'operatore `==`. La variabile di istanza `dati` rappresentata nel Listato 15.1 è di tipo `String`; contrariamente a `collegamento`, per controllare l'uguaglianza di due variabili di tipo `String` si utilizza il metodo `equals`. Di norma, le liste concatenate nascono vuote. Poiché si suppone che la variabile `testa` contenga un riferimento al primo nodo di una lista concatenata, per indicare che una lista è vuota si assegna a `testa` il valore `null`. Questa tecnica è molto utilizzata dagli algoritmi che gestiscono liste concatenate.

Si usa `null` per indicare la fine di una lista (e per le liste vuote)

Il riferimento di testa di una lista concatenata vuota contiene `null`, così come la parte di collegamento dell'ultimo nodo di una lista concatenata non vuota.

15.1.2 Implementare le operazioni di una lista concatenata

Il Listato 15.2 contiene la definizione di una classe lista concatenata che utilizza la definizione della classe nodo fornita nel Listato 15.1. Si noti che questa nuova classe ha solo una variabile di istanza, denominata `testa`, che contiene un riferimento al primo nodo della lista concatenata oppure contiene `null` se la lista concatenata è vuota (cioè non contiene nodi). L'unico costruttore definito imposta a `null` questa variabile di istanza `testa`, per indicare che la lista è vuota.

LISTATO 15.2 Una classe lista concatenata.

```
public class ListaConcatenataDiStringhe {
   private NodoLista testa;

   public ListaConcatenataDiStringhe() {
      testa = null;
   }

   /**
   Mostra i dati della lista.
   */
   public void mostraLista() {
      NodoLista posizione = testa;
      while (posizione != null) {
         System.out.println(posizione.getDati());
         posizione = posizione.getCollegamento();
      }
   }

   /**
   Restituisce il numero di nodi che compongono la lista.
   */
   public int lunghezza() {
      int conteggio = 0;
      NodoLista posizione = testa;
      while (posizione != null) {
         conteggio++;
         posizione = posizione.getCollegamento();
      }
      return conteggio;
   }
```

Più avanti in questo capitolo, si fornirà un'altra definizione di questa classe.

```
    /**
    Aggiunge all'inizio della lista
    un nodo contenente datiDaAggiungere.
    */
    public void aggiungiNodoInTesta(String datiDaAggiungere) {
        testa = new NodoLista(datiDaAggiungere, testa);
    }

    /**
    Elimina il primo nodo della lista.
    */
    public void eliminaNodoDiTesta() {
        if (testa != null)
            testa = testa.getCollegamento();
        else {
            System.out.println("Si sta eliminando da una lista vuota.");
            System.exit(0);
        }
    }

    /**
    Verifica se elemento è nella lista.
    */
    public boolean nellaLista(String elemento) {
        return trova(elemento) != null;
    }

    // Restituisce un riferimento al primo nodo che contiene elemento.
    // Se elemento non è nella lista, restituisce null.
    private NodoLista trova(String elemento) {
        boolean trovato = false;
        NodoLista posizione = testa;
        while ((posizione != null) && !trovato) {
            String datiAllaPosizione = posizione.getDati();
            if (datiAllaPosizione.equals(elemento))
                trovato = true;
            else
                posizione = posizione.getCollegamento();
        }
        return posizione;
    }
}
```

Prima di parlare dell'aggiunta ed eliminazione dei nodi in una lista concatenata, si supponga che una lista concatenata contenga già alcuni nodi e che sia necessario presentare sullo schermo il contenuto di tutti i nodi.

Lo si può fare con le seguenti istruzioni:

```
NodoLista posizione = testa;
while (posizione != null) {
    System.out.println(posizione.getDati());
    posizione = posizione.getCollegamento();
}
```

La variabile `posizione` contiene un riferimento a un nodo. Inizialmente `posizione` contiene lo stesso riferimento della variabile di istanza `testa`; in questo modo, inizia posizionandosi sul primo nodo. Dopo aver visualizzato i dati contenuti in un nodo, il riferimento contenuto in `posizione` passa da un nodo al successivo grazie all'assegnamento.

```
posizione = posizione.getCollegamento();
```

Questo processo è illustrato nella Figura 15.2.

Per osservare che questo assegnamento "sposta" la variabile `posizione` al nodo successivo, si noti che `posizione` contiene un riferimento al nodo indicato dalla freccia di `posizione` nella Figura 15.2. In questo modo, `posizione` è un nome per quel nodo e `posizione.collegamento` è un nome per la porzione di collegamento di quel nodo. Quindi, `posizione.collegamento` si riferisce al nodo successivo. Il valore di `collegamento` è ottenuto invocando il metodo *get* `getCollegamento`. Dunque, in una lista concatenata, un riferimento al nodo successivo è dato da `posizione.getCollegamento()`. La variabile `posizione` si "sposta" assegnandole il valore `posizione.getCollegamento()`. Il ciclo precedente continua a spostare la variabile `posizione` verso la fine della lista concatenata, mostrando a ogni avanzamento i dati contenuti in ogni nodo. Quando `posizione` raggiunge l'ultimo nodo, ne visualizza i dati ed esegue ancora:

```
posizione = posizione.getCollegamento();
```

Se si analizza la Figura 15.2, si noterà che, a questo punto, il valore di `posizione` è `null`. Quando si verifica questa situazione, è necessario interrompere il ciclo; pertanto si itera il ciclo fintantoché `posizione` è diverso da `null`.

Il metodo `mostraLista` contiene il ciclo appena trattato. Il metodo `lunghezza` utilizza un ciclo simile, ma, invece di mostrare i dati in ogni nodo, `lunghezza` conta solamente i nodi, mentre il ciclo si sposta da nodo a nodo. Quando il ciclo termina, `lunghezza` restituisce il numero di elementi presenti nella lista.

Un processo che si muove da nodo a nodo in una lista concatenata si dice che **attraversa** la lista. Dunque sia `mostraLista` sia `lunghezza` attraversano la lista.

Il metodo `aggiungiNodoInTesta` aggiunge un nodo all'inizio della lista concatenata così che il nuovo nodo diventi il primo della lista. Questa operazione viene eseguita dalla singola istruzione:

```
testa = new NodoLista(datiDaAggiungere, testa);
```

In altre parole, alla variabile `testa` viene assegnato un nuovo nodo, che diverrà il primo nodo della lista concatenata. Per collegare questo nuovo nodo al resto della lista basta assegnare alla variabile di istanza `collegamento` del nuovo nodo il riferimento al vecchio primo nodo. Ma tale operazione viene svolta direttamente dall'istruzione `new NodoLista(datiDaAggiungere, testa)`. Infatti, il costruttore assegna alla variabile di istanza `posizione` del nuovo nodo il vecchio primo nodo referenziato da `testa`. Questo processo è illustrato nella Figura 15.3.

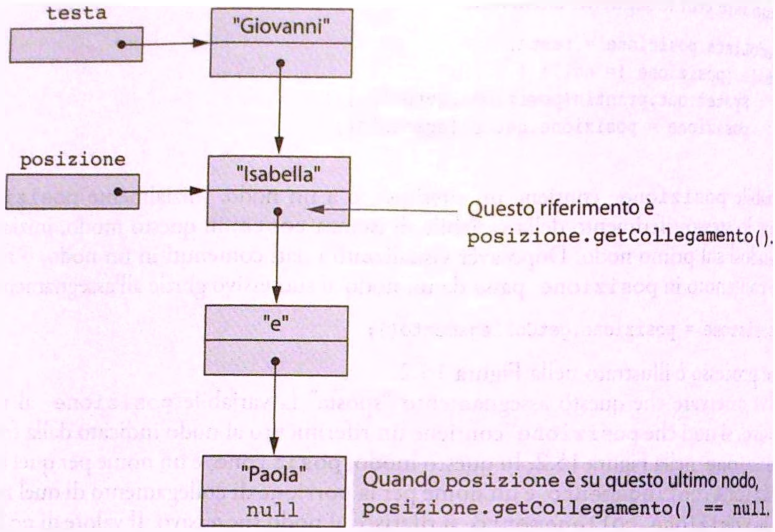


Figura 15.2 Visitare una lista.

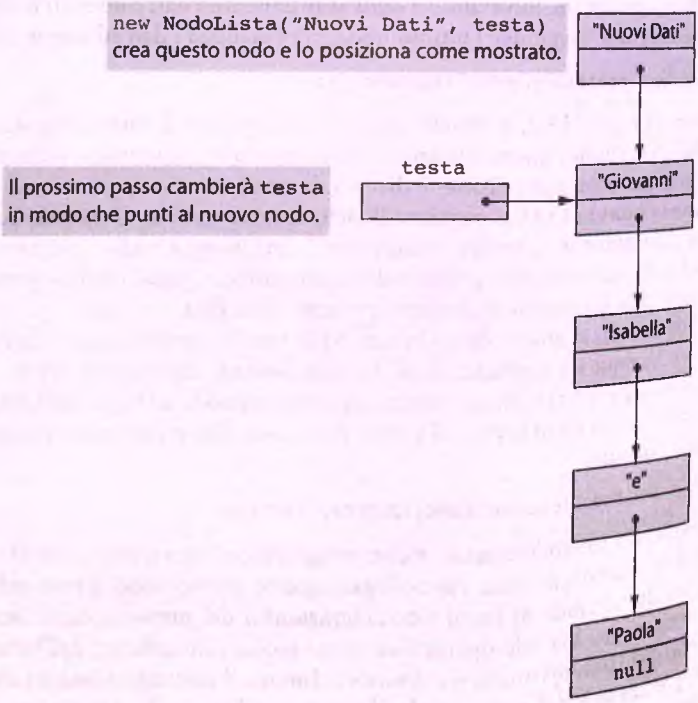


Figura 15.3 Aggiungere un nodo in testa alla lista concatenata.

Finora, è stato aggiunto un nodo solo all'inizio della lista concatenata. Più avanti si vedrà come aggiungere nodi in altre posizioni di una lista concatenata. L'inizio della lista è il punto in cui è più facile aggiungere o anche cancellare un nodo.

Il metodo `eliminaNodoDiTesta` rimuove il primo nodo dalla lista concatenata e fa sì che la variabile `testa` faccia riferimento a quello che prima era il secondo nodo della lista concatenata. Si lascia al lettore il compito di verificare che il seguente assegnamento realizzi correttamente questa eliminazione:

```
testa = testa.getCollegamento();
```

Il Listato 15.3 contiene un semplice programma che illustra il comportamento di alcuni dei metodi trattati.

FAQ Cosa accade a un nodo cancellato?

Quando il codice cancella un nodo da una lista concatenata, rimuove dalla lista concatenata il riferimento a quel nodo. In questo modo il nodo non farà più parte della lista. Tuttavia, non è stato dato alcun comando per la distruzione del nodo che, pertanto, deve ancora essere da qualche parte nella memoria del computer. Se non ci sono altri riferimenti al nodo cancellato, lo spazio di memoria da esso occupato potrebbe essere reso disponibile per altri usi. In molti linguaggi di programmazione, il programmatore deve tenere traccia dei nodi cancellati e dare espliciti comandi per liberare la memoria occupata, in modo da recuperarla per altri usi. Questo processo è chiamato **garbage collection** (letteralmente "raccolta della spazzatura"). In Java, questo compito viene svolto automaticamente.

LISTATO 15.3 Uso della lista concatenata.

```
public class ListaConcatenataDiStringheDemo {

  public static void main(String[] args) {
      ListaConcatenataDiStringhe lista = new ListaConcatenataDiStringhe();
      lista.aggiungiNodoInTesta("Uno");
      lista.aggiungiNodoInTesta("Due");
      lista.aggiungiNodoInTesta("Tre");
      System.out.println("La lista ha " + lista.lunghezza() + " elementi.");
      lista.mostraLista();

      if (lista.nellaLista("Tre"))
          System.out.println("Tre e' sulla lista.");
      else
          System.out.println("Tre NON e' sulla lista.");
      lista.eliminaNodoDiTesta();

      if (lista.nellaLista("Tre"))
          System.out.println("Tre e' sulla lista.");
      else
          System.out.println("Tre NON e' sulla lista.");
```

```
        lista.eliminaNodoDiTesta();
        lista.eliminaNodoDiTesta();
        System.out.println("Inizio della lista:");
        lista.mostraLista();
        System.out.println("Fine della lista.");
    }
}
```

Esempio di output

```
La lista ha 3 elementi.
Tre
Due
Uno
Tre e' sulla lista.
Tre NON e' sulla lista.
Inizio della lista:
Fine della lista.
```

NullPointerException

Senza alcun dubbio, a un certo punto dell'esecuzione di un programma ci si è imbattuti in un messaggio `NullPointerException`. Se non si è mai ricevuto tale messaggio, congratulazioni! Il messaggio `NullPointerException` indica che il codice ha tentato di utilizzare una variabile di un tipo classe per referenziare un oggetto, ma la variabile contiene `null`. Vale a dire che la variabile non contiene alcun riferimento a un oggetto. D'ora in avanti questo messaggio dovrebbe avere molto più senso. Nei nodi di una lista, si utilizza `null` per indicare che una variabile di istanza collegamento non contiene nessun riferimento. Di conseguenza, un valore `null` indica nessun riferimento a oggetto e questo è il motivo per cui l'eccezione si chiama `NullPointerException`.

`NullPointerException` è una delle eccezioni che non bisogna catturare in un blocco `catch` o dichiarare in una clausola `throws`. Significa solo che occorre correggere il codice.

Molti nodi non hanno nome

In una lista concatenata, la variabile `testa` contiene un riferimento al primo nodo. Di conseguenza, `testa` può essere usata come un nome per il primo nodo. Tuttavia, gli altri nodi nella lista concatenata non hanno variabili con nome che contengono il riferimento a ciascuno di loro, in pratica non hanno alcun nome. L'unico modo per assegnare loro un nome è tramite qualche riferimento indiretto, come `testa.getCollegamento()`, oppure utilizzando un'altra variabile di tipo `NodoLista`, come la variabile locale `posizione` nel metodo `mostraLista` (Listato 15.2).

15.1.3 Privacy leak

L'argomento trattato di seguito è importante, ma un po' delicato. Al fine di comprenderlo, sarà utile rivedere il Paragrafo 9.4 del Capitolo 9, dove si è detto che una violazione dell'*information hiding* accade quando un metodo restituisce un riferimento a una variabile di istanza privata di un tipo classe. La restrizione `private` sulla variabile di istanza potrebbe essere facilmente aggirata se si invoca sulla variabile di istanza un metodo *get*. Infatti, ottenere un riferimento a un tale oggetto potrebbe permettere al programmatore di cambiare la variabile di istanza privata dell'oggetto.

Si consideri il metodo `getCollegamento` nella classe `NodoLista` (Listato 15.1) che restituisce un valore di tipo `NodoLista`. Vale a dire, restituisce un riferimento a un oggetto `NodoLista`, ossia, un nodo. Inoltre, `NodoLista` ha metodi *set* pubblici che possono danneggiare i contenuti di un nodo. Così, `getCollegamento` può causare una *privacy leak*.

Al contrario, il metodo `getDati` della classe `NodoLista` non causa una *privacy leak*, ma solo perché la classe `String` non ha metodi *set*. La classe `String` è un caso speciale. Se i dati fossero stati di un altro tipo classe, `getDati` avrebbe potuto produrre una *privacy leak*.

Se la classe `NodoLista` viene usata solo nella definizione della classe `ListaConcatenataDiStringhe` e in altre classi analoghe, non c'è nessuna violazione dell'*information hiding*. Questo perché nessuno dei metodi pubblici nella classe `ListaConcatenataDiStringhe` restituisce un riferimento a un nodo. Sebbene il tipo di ritorno dal metodo `trova` è `NodoLista`, questo è un metodo privato. Se il metodo `trova` fosse stato pubblico, si sarebbe verificata una *privacy leak*. In sostanza, rendere privato il metodo `trova` non è semplicemente un fatto stilistico.

Sebbene non ci sia alcun problema nella definizione della classe `NodoLista` (quando questa è utilizzata in una classe definita come `ListaConcatenataDiStringhe`), non si è certi che sarà sempre utilizzata in questo contesto. Si può risolvere questo problema di *privacy leak* in diversi modi. Il modo più semplice consiste nel rendere la classe `NodoLista` una *inner class* privata della classe `ListaConcatenataDiStringhe`, come discusso nei prossimi due paragrafi: "Inner class" e "Classi nodo come inner class". Una soluzione semplice consiste nel collocare entrambe le classi `NodoLista` e `ListaConcatenataDiStringhe` in un unico package, cambiando la restrizione delle variabili di istanza da private a package (come brevemente discusso nel paragrafo "Modalità d'accesso protected" del Capitolo 10) e omettere il metodo *get* `getCollegamento`.

15.1.4 Inner class

Le ***inner class*** (letteralmente "classi interne") sono classi definite all'interno di altre classi. Sebbene una descrizione completa delle *inner class* non rientri negli scopi di questo testo, alcuni semplici utilizzi delle *inner class* possono essere facili da capire e utili. In questo paragrafo si descrivono le *inner class* in generale e si mostra come usarle. Nel prossimo paragrafo si fornisce un esempio di una *inner class* definita all'interno di una classe lista concatenata. La nuova definizione della classe basata sulle *inner class* sarà una soluzione al problema di *privacy leak* appena descritto.

Definire una *inner class* è molto facile: basta includere la definizione della *inner class* all'interno di un'altra classe, la **classe esterna** (*outer class*), nel seguente modo:

```
public void mostraLista() {
    NodoLista posizione = testa;
    while (posizione != null) {
        System.out.println(posizione.dati);
        posizione = posizione.collegamento;
    }
}

/**
Restituisce il numero di nodi che compongono la lista.
*/
public int lunghezza() {
    int conteggio = 0;
    NodoLista posizione = testa;
    while (posizione != null) {
        conteggio++;
        posizione = posizione.collegamento;
    }
    return conteggio;
}

/**
Aggiunge all'inizio della lista
un nodo contenente datiDaAggiungere.
*/
public void aggiungiNodoInTesta(String datiDaAggiungere) {
    testa = new NodoLista(datiDaAggiungere, testa);
}

/**
Elimina il primo nodo della lista.
*/
public void eliminaNodoDiTesta() {
    if (testa != null)
        testa = testa.collegamento;
    else {
        System.out.println("Si sta eliminando da una lista vuota.");
        System.exit(0);
    }
}

/**
Verifica se elemento è nella lista.
*/
public boolean nellaLista(String elemento) {
    return trova(elemento) != null;
}
```

Si noti che la classe esterna ha accesso diretto alle variabili di istanza `dati` e `collegamento` della classe interna (inner class).

```
    // Restituisce un riferimento al primo nodo
    // che contiene elemento. Se elemento
    // non è nella lista, restituisce null.
    private NodoLista trova(String elemento) {
        boolean trovato = false;
        NodoLista posizione = testa;
        while ((posizione != null) && !trovato) {
            String datiAllaPosizione = posizione.dati;
            if (datiAllaPosizione.equals(elemento))
                trovato = true;
            else
                posizione = posizione.collegamento;
        }
        return posizione;
    }

    private class NodoLista {
        private String dati;
        private NodoLista collegamento;

        public NodoLista() {
            collegamento = null;
            dati = null;
        }

        public NodoLista(String valoreDati, NodoLista valoreCollegamento) {
            dati = valoreDati;
            collegamento = valoreCollegamento;
        }
    }
}
```

Una inner class.

Fine della definizione della classe esterna.

LISTATO 15.5 Inserire in un array gli elementi della lista concatenata.

```
/**
 Restituisce un array degli elementi presenti nella lista.
*/
public String[] convertiInArray() {
    String[] unArray = new String[lunghezza()];
    NodoLista posizione = testa;
    int i = 0;
    while (posizione != null) {
        unArray[i] = posizione.dati;
        i++;
        posizione = posizione.collegamento;
    }
    return unArray;
}
```

Questo metodo può essere aggiunto alla classe lista concatenata del Listato 15.4.

Iteratori

Ogni variabile che consente di attraversare una collezione di elementi, come un array o una lista concatenata, muovendosi in maniera appropriata da elemento a elemento è chiamata iteratore. Con "in maniera appropriata", si intende che in un ciclo completo di iterazioni ogni elemento viene visitato esattamente una volta, che si possono leggere i dati di ogni elemento e, se gli elementi lo consentono, si possono cambiare i dati.

Un array che contiene i dati di una lista concatenata non è, però, sufficiente se si vuole che un iteratore, oltre a muoversi attraverso la lista concatenata, consenta anche di effettuare operazioni, come cambiare i dati contenuti in un nodo o persino inserire o cancellare un nodo. In ogni caso, un iteratore per un array suggerisce il modo in cui procedere qualora si voglia realizzare un iteratore per una lista concatenata che permetta di effettuare modifiche sugli elementi della lista stessa. Proprio come un indice specifica un elemento di un array, un riferimento a un nodo specifica un nodo. Di conseguenza, se si aggiunge una variabile di istanza, per esempio `corrente`, alla classe `ListaConcatenataDiStringheAutoContenuta` fornita nel Listato 15.4, si può utilizzare questa variabile di istanza come iteratore. Ciò è illustrato nel Listato 15.6, dove la classe è stata rinominata `ListaConcatenataDiStringheConIteratore`. Come si può notare, è stato aggiunto un insieme di metodi per gestire la variabile di istanza `corrente`. Infatti, questa variabile di istanza è l'iteratore, ma, poiché è stata definita `private`, occorrono dei metodi per gestirla. Sono stati inoltre aggiunti i metodi per inserire ed eliminare un nodo in qualsiasi punto della lista concatenata. L'iteratore rende più facile esprimere questi metodi per aggiungere e cancellare i nodi, poiché fornisce un modo per assegnare un nome a un nodo arbitrario.

MyLab

LISTATO 15.6 Una lista concatenata con iteratore.

```
/**
Lista concatenata con iteratore. Un nodo è il "nodo corrente".
Inizialmente, il nodo corrente è il primo nodo.
Può essere cambiato con il nodo successivo finché l'iterazione
non va oltre la fine della lista.
*/
public class ListaConcatenataDiStringheConIteratore {
    private NodoLista testa;
    private NodoLista corrente;
    private NodoLista precedente;

    public ListaConcatenataDiStringheConIteratore() {
        testa = null;
        corrente = null;
        precedente = null;
    }

    public void aggiungiNodoInTesta(String datiDaAggiungere) {
        testa = new NodoLista(datiDaAggiungere, testa);
```

```
        if ((corrente == testa.collegamento) && (corrente != null))
            //Se corrente è al vecchio nodo di testa
            precedente = testa;
    }

    /**
    Imposta l'iteratore all'inizio della lista.
    */
    public void reimpostaIterazione() {
        corrente = testa;
        precedente = null;
    }

    /**
    Restituisce vero se l'iterazione non è terminata.
    */
    public boolean altriElementi() {
        return corrente != null;
    }

    /**
    Sposta l'iteratore al nodo successivo.
    */
    public void vaiAlSuccessivo() {
        if (corrente != null) {
            precedente = corrente;
            corrente = corrente.collegamento;
        } else if (testa != null) {
            System.out.println("Si e' iterato troppe volte o" +
                               " l'iterazione non e' stata" +
                               " inizializzata.");
            System.exit(0);
        } else {
            System.out.println("Si sta iterando su una lista vuota.");
            System.exit(0);
        }
    }

    /**
    Restituisce i dati del nodo corrente.
    */
    public String getDatiDaNodoCorrente() {
        String risultato = null;
        if (corrente != null)
            risultato = corrente.dati;
```

```
        else {
            System.out.println("Si sta richiedendo i dati quando" +
                               " corrente non e' posizionato su nessun nodo.");
            System.exit(0);
        }
        return risultato;
    }

    /**
    Sostituisce i dati al nodo corrente.
    */
    public void impostaDatiANodoCorrente(String nuoviDati) {
        if (corrente != null) {
            corrente.dati = nuoviDati;
        } else {
            System.out.println("Si sta impostando i dati quando" +
                               " corrente non e' posizionato su nessun nodo.");
            System.exit(0);
        }
    }

    /**
    Inserisce un nuovo nodo che contiene nuoviDati dopo il nodo corrente.
    Il nodo corrente è uguale a quello prima dell'invocazione.
    Precondizione: La lista non è vuota; il nodo corrente non è oltre
    la fine della lista.
    */
    public void inserisciDopoNodoCorrente(String nuoviDati) {
        NodoLista nuovoNodo = new NodoLista();
        nuovoNodo.dati = nuoviDati;
        if (corrente != null) {
            nuovoNodo.collegamento = corrente.collegamento;
            corrente.collegamento = nuovoNodo;
        } else if (testa != null) {
            System.out.println("Si sta inserendo quando iteratore"
                               " ha visitato tutti i nodi" +
                               " o non e' stato inizializzato.");
            System.exit(0);
        } else {
            System.out.println("Si sta utilizzando" +
                               " inserisciDopoNodoCorrente con"
                               " una lista vuota.");
            System.exit(0);
        }
    }
```

```
/**
Elimina il nodo corrente. Dopo l'invocazione,
il nodo corrente è o il nodo successivo a quello eliminato
o null se non c'è nessun nodo successivo.
*/
public void eliminaNodoCorrente() {
    if ((corrente != null) && (precedente != null)) {
        precedente.collegamento = corrente.collegamento;
        corrente = corrente.collegamento;
    } else if ((corrente != null) && (precedente == null)) {
        //Al nodo di testa
        testa = corrente.collegamento;
        corrente = testa;
    } else {//corrente == null
        System.out.println("Si sta eliminando un nodo" +
                           " corrente non inizializzato o la" +
                           " lista e' vuota.");
        System.exit(0);
    }
}
```

`eliminaNodoDiTesta` non è più necessario, poiché si ha `eliminaNodoCorrente`, ma se si volesse mantenere `eliminaNodoDiTesta`, si dovrebbe ridefinirlo per tenere in considerazione `corrente` e `precedente`.

```
    <I metodi lunghezza, nellaLista e mostraLista e la inner class
    NodoLista sono gli stessi presenti nel Listato 15.4>
    <Il metodo convertiInArray è lo stesso presente nel Listato 15.5>
}
```

In aggiunta alle variabili di istanza `testa` e `corrente`, è stata introdotta una variabile di istanza chiamata `precedente`. L'idea è che, mentre il riferimento `corrente` scende lungo la lista concatenata, il riferimento `precedente` lo precede di un nodo. Questa struttura fornisce un modo per far riferimento al nodo precedente rispetto a quello `corrente`. Poiché i collegamenti nella lista concatenata si muovono tutti in un'unica direzione, occorre un nodo `precedente` per retrocedere di un nodo.

Il metodo `reimpostaIterazione` inizializza `corrente` all'inizio della lista concatenata, fornendogli un riferimento al primo nodo (`testa`), come segue:

```
corrente = testa;
```

Poiché la variabile di istanza `precedente` non ha nodi precedenti a cui riferirsi, il metodo `reimpostaIterazione` le assegna semplicemente il valore `null`.

Il metodo `vaiAlSuccessivo` sposta l'iteratore al nodo successivo, come segue:

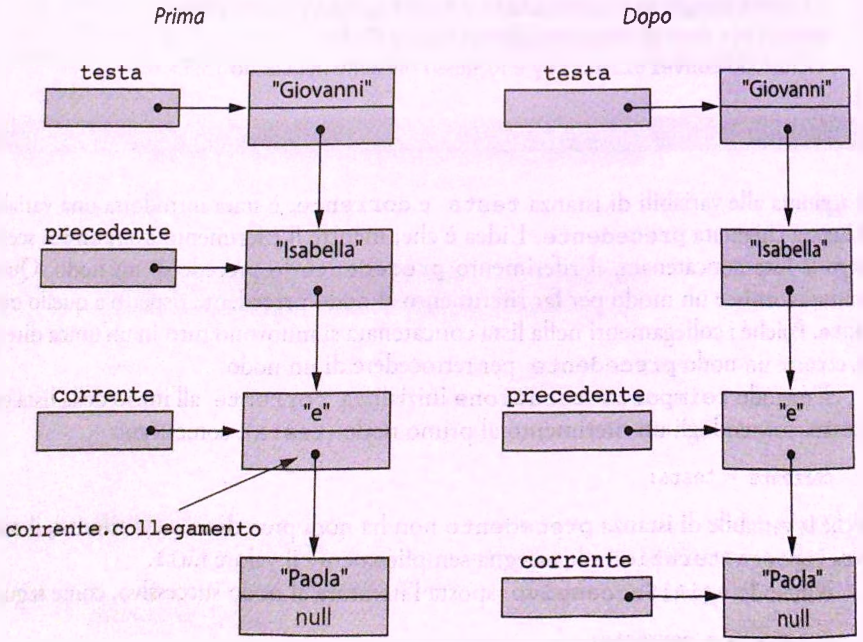
```
precedente = corrente;
corrente = corrente.collegamento;
```

Questo processo è illustrato nella Figura 15.4. Nel metodo `vaiAlSuccessivo`, le ultime due clausole dell'istruzione ramificata `if-else` producono un messaggio di errore quando il metodo `vaiAlSuccessivo` viene utilizzato in una situazione dove non avrebbe senso.

Il metodo `altriElementi` restituisce vero finché `corrente` è diverso da `null`, cioè finché `corrente` contiene un riferimento a qualche nodo. Questo risultato il più delle volte è sensato, ma perché il metodo restituisce vero quando `corrente` contiene il riferimento all'ultimo nodo? Quando `corrente` si riferisce all'ultimo nodo, il programma non è in grado di dire che ha raggiunto l'ultimo nodo finché non ha invocato ancora una volta `vaiAlSuccessivo`, a quel punto a `corrente` è assegnato `null`. Si osservi la Figura 15.4 o la definizione di `vaiAlSuccessivo` per vedere come funziona questo processo. Quando `corrente` è uguale a `null`, `altriElementi` restituisce falso, indicando che è stata attraversata l'intera lista.

Ora che la lista concatenata è dotata di un iteratore, il programma ha un modo per far riferimento a ogni nodo della lista concatenata. La variabile di istanza `corrente` può mantenere un riferimento a un nodo qualunque; quel nodo è conosciuto come il **nodo all'iteratore**. Il metodo `inserisciDopoNodoCorrente` inserisce un nuovo nodo dopo il nodo all'iteratore (`corrente`). Questo processo è illustrato nella Figura 15.5. Il metodo `eliminaNodoCorrente` elimina il nodo all'iteratore. Questo processo è illustrato nella Figura 15.6.

Gli altri metodi nella classe `ListaConcatenataDiStringheConIteratore` (Listato 15.6) sono abbastanza semplici e si lascia al lettore il compito di studiare il loro funzionamento.



`corrente = corrente.collegamento` fornisce a `corrente` un riferimento all'ultimo nodo.

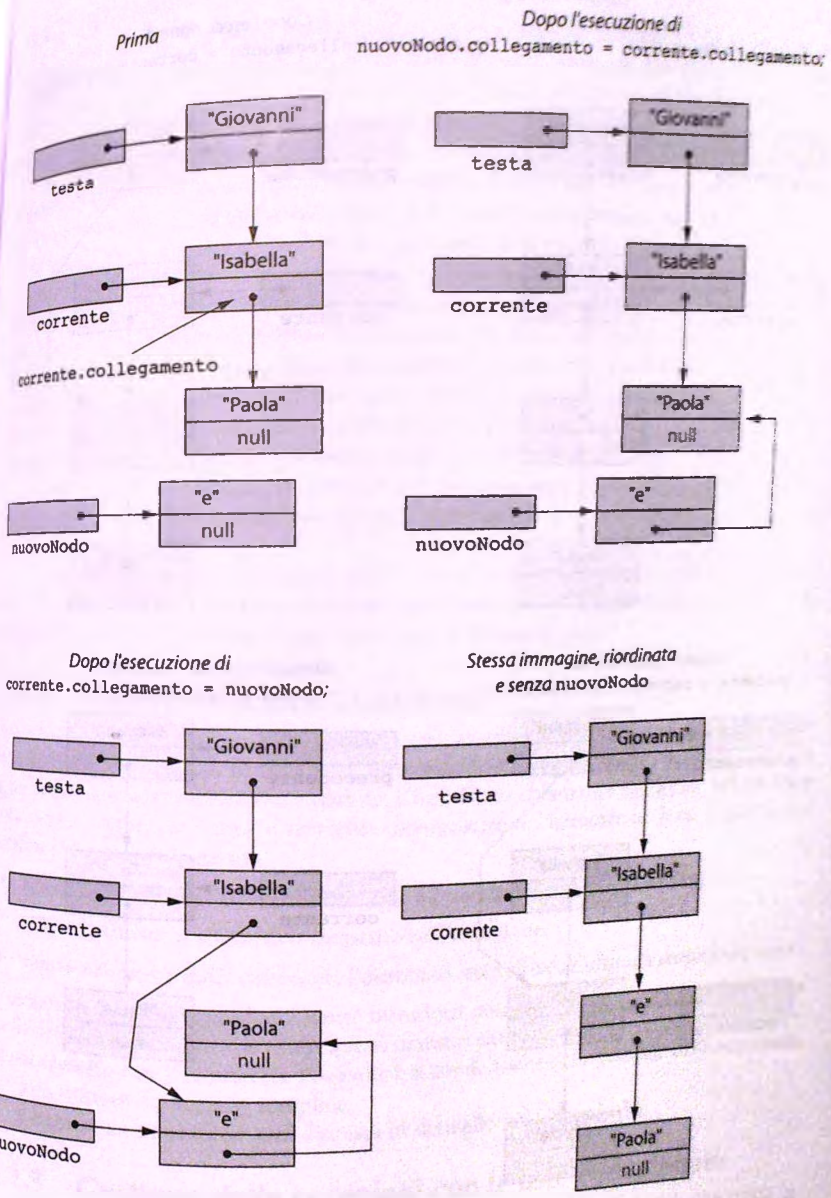


Figura 15.5 Aggiungere un nodo a una lista concatenata utilizzando `inserisciDopoNodoCorrente`.

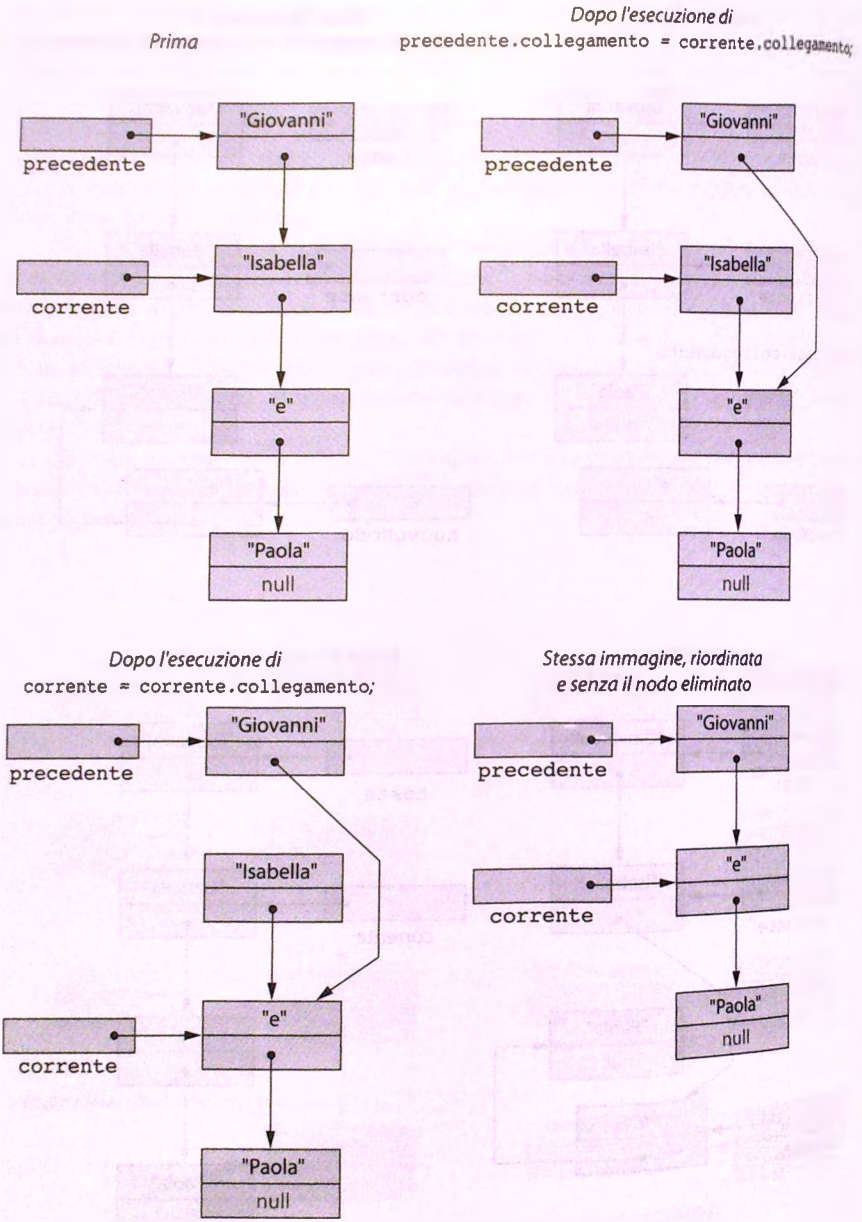


Figura 15.6 Eliminare un nodo.

15.1.7 Iteratori interni ed esterni (opzionale)

La classe `ListaConcatenataDiStringheConIteratore` (Listato 15.6) utilizza la variabile di istanza `corrente` di tipo `NodoLista` come iteratore per visitare uno dopo l'altro i nodi della lista concatenata. Un iteratore definito all'interno della classe della lista concatenata è conosciuto come un **iteratore interno**.

Se si inseriscono i valori di una lista concatenata in un array attraverso il metodo `convertiInArray`, si può utilizzare una variabile di tipo `int` come un iteratore dell'array. La variabile intera mantiene un indice dell'array e così specifica un elemento dell'array (e, quindi, un elemento della lista concatenata). Se la variabile intera si chiama `posizione` e l'array si chiama `a`, l'iteratore `posizione` specifica l'elemento `a[posizione]`. Per muoversi al prossimo elemento, si incrementa semplicemente di una unità il valore di `posizione`. Un iteratore definito esternamente alla lista concatenata, come la variabile intera `posizione`, è chiamato **iteratore esterno**. La cosa importante da notare è che la variabile intera `posizione`, che è l'iteratore, è esterna alla lista concatenata, non tanto il fatto che l'array è esterno alla lista concatenata. Per comprendere meglio questo punto, si noti che la variabile intera `posizione` è anche un iteratore esterno per l'array.

Si possono avere anche più iteratori esterni contemporaneamente, ognuno in una posizione differente della stessa lista. Questo non è possibile con gli iteratori interni e ciò è un evidente svantaggio.

È possibile definire un iteratore esterno che lavora direttamente sulla lista concatenata, piuttosto che su un array di dati della lista concatenata. A ogni modo, questa tecnica è leggermente più complicata e non verrà trattata in questo testo.

15.1.8 L'interfaccia Java `Iterator`

L'introduzione sugli iteratori li ha definiti come variabili che consentono di attraversare una collezione di elementi. In ogni caso, Java considera formalmente un iteratore come un oggetto, non semplicemente una variabile. L'interfaccia chiamata `Iterator` nel package `java.util` stabilisce come si dovrebbe comportare un iteratore in Java. L'interfaccia specifica i tre metodi seguenti.

- `hasNext` restituisce vero se l'iterazione ha un altro elemento da restituire.
- `next` restituisce il successivo elemento nell'iterazione.
- `remove` rimuove dalla collezione l'elemento restituito dall'ultima invocazione `next`.

Gli esempi di iteratori precedentemente introdotti non soddisfano questa interfaccia, ma è facile definire delle classi che usino questi iteratori per realizzare in maniera soddisfacente questa interfaccia. L'interfaccia `Iterator` si avvale anche della gestione delle eccezioni, ma il loro utilizzo è piuttosto semplice.

L'interfaccia `Iterator` sarà discussa in dettaglio nel Capitolo 16.

15.1.9 Gestione delle eccezioni con le liste concatenate

Prima di leggere questa parte del capitolo occorre aver letto il Capitolo 13, che tratta la gestione delle eccezioni.

Si consideri la classe `ListaConcatenataDiStringheConIteratore` nel Listato 15.6. I suoi metodi sono stati definiti in modo che ogniqualvolta qualcosa fallisce, inviino un messaggio d'errore sullo schermo e facciano terminare il programma. Tuttavia, potreb-

be non essere sempre necessario terminare il programma ogni volta che accade qualcosa di anomalo. Per consentire al programmatore di fornire un'azione alternativa specifica per queste situazioni anomale, ha più senso sollevare un'eccezione, in modo da lasciar decidere al programmatore come gestire la situazione. Il programmatore potrebbe certamente decidere di terminare il programma, ma potrebbe anche gestire la situazione in un altro modo. Per esempio, potrebbe riscrivere il metodo `vaiAlSuccessivo` come segue:

```
public void vaiAlSuccessivo() throws ListaConcatenataException {
    if (corrente != null) {
        precedente = corrente;
        corrente = corrente.collegamento;
    } else if (testa != null)
        throw new ListaConcatenataException("Iterato troppe volte" +
                                            " oppure iterazione" +
                                            " non inizializzata.");
    else
        throw new ListaConcatenataException("Si sta iterando una lista" +
                                            " vuota.");
}
```

In questa versione è stato sostituito ogni ramo che terminava il programma con un ramo che crea un'eccezione. La classe di eccezione `ListaConcatenataException` può essere molto semplice, come mostrato nel Listato 15.7.

MyLab

LISTATO 15.7 La classe `ListaConcatenataException`.

```
public class ListaConcatenataException extends Exception {
    public ListaConcatenataException() {
        super("Eccezione di lista concatenata");
    }

    public ListaConcatenataException(String messaggio) {
        super(messaggio);
    }
}
```

Durante un'iterazione si può lanciare un'eccezione per svariati motivi, per esempio, per un tentativo di leggere oltre la fine di una lista concatenata.

Si supponga che la versione di `ListaConcatenataDiStringheConIteratore` che solleva le eccezioni si chiami `ListaConcatenataDiStringheConIteratore2`. Il seguente codice rimuove da una lista concatenata tutti i nodi che contengono una stringa (`stringaCattiva`), sollevando un'eccezione se tenta di leggere oltre la fine della lista:

```
ListaConcatenataDiStringheConIteratore2 lista =
                                new ListaConcatenataDiStringheConIteratore2();
String stringaCattiva;
<Codice per riempire la lista concatenata e inizializzare la variabile stringaCattiva>
lista.reimpostaIterazione();
```

```
try {
    while (lista.lunghezza() >= 0) {
        if (stringaCattiva.equals(lista.getDatiDaNodoCorrente()))
            lista.eliminaNodoCorrente();
        else
            lista.vaiAlSuccessivo();
    }
} catch (ListaConcatenataException e) {
    if (e.getMessage().equals("Si sta iterando una lista vuota.")) {
        //Questo non dovrebbe mai succedere,
        //ma il blocco catch è obbligatorio.
        System.out.println("Errore Fatale.");
        System.exit(0);
    }
}
System.out.println("Lista pulita da cattive stringhe.");
```

Sebbene questo utilizzo delle eccezioni per verificare la fine di una lista possa sembrare insolito, in realtà è molto utilizzato anche nelle classi presenti nella Java Class Library. Per esempio, Java richiede qualcosa di simile quando controlla la fine di un file binario. Di certo, ci sono molti altri utilizzi per la classe `ListaConcatenataException`.

ESEMPIO DI PROGRAMMAZIONE
UNA LISTA CONCATENATA GENERICA

L'esempio proposto modifica la definizione della classe lista concatenata che appare nel Listato 15.4 perché utilizzi un tipo parametrico `E` (si veda il Capitolo 12) invece del tipo `String`, come tipo di dato memorizzato nella lista. Il Listato 15.8 mostra il risultato di questa revisione.

Si noti che l'intestazione del costruttore appare come dovrebbe essere in una classe non parametrica. Questa non include `<E>`, il tipo parametrico dentro le parentesi angolari, dopo il nome del costruttore. Ciò può apparire inaspettato. Anche se un costruttore (come qualsiasi altro metodo nella classe) può utilizzare (dentro il suo corpo) un tipo parametrico, la sua intestazione potrebbe non includere il tipo parametrico.

Nello stesso modo la *inner class* `NodoLista` non ha `<E>` dopo il suo nome nell'intestazione, ma `NodoLista` utilizza il tipo parametrico `E` all'interno della sua definizione. Tuttavia, ciò è corretto perché `NodoLista` è una *inner class* e può accedere al tipo parametrico della sua classe esterna.

La classe della lista concatenata nel Listato 15.4 può avere un metodo chiamato `convertiInArray` (mostrato nel Listato 15.5) che restituisce un array contenente gli stessi dati della lista concatenata. In ogni modo, la versione generica di una lista concatenata non può implementare questo stesso metodo. Se, infatti, si traducesse `convertiInArray` nel Listato 15.5 in modo da poterlo includere nella lista concatenata generica nel Listato 15.8, esso inizierebbe come segue:

```
public E[] convertiInArray(){
    E[] unArray = new E[lunghezza()];     //Illegale
    ...
```

L'espressione new E[lunghezza()] non è permessa. Ci sono situazioni in cui non si può includere il tipo parametrico e questa, sfortunatamente, è proprio una di queste. Di conseguenza, non si può definire il metodo convertiInArray nella lista concatenata generica. Al suo posto, nel Listato 15.8 viene definito il metodo convertiInArrayList, che restituisce un'istanza della classe ArrayList. Si noti che la sua definizione contiene un ciclo come quello presente in convertiInArray.

Il Listato 15.9 contiene un semplice esempio di come usare la lista concatenata parametrica.

MyLab

LISTATO 15.8 Una classe parametrica per liste concatenate.

```
import java.util.ArrayList;

public class ListaConcatenata<E> {
    private NodoLista testa;

    public ListaConcatenata() {
        testa = null;
    }

    <I metodi mostraLista, lunghezza e eliminaNodoDiTesta sono gli
    stessi come nel Listato 15.4>

    public void aggiungiNodoInTesta(E datiDaAggiungere) {
        testa = new NodoLista(datiDaAggiungere, testa);
    }

    public boolean nellaLista(E elemento) {
        return trova(elemento) != null;
    }

    private NodoLista trova(E elemento) {
        boolean trovato = false;
        NodoLista posizione = testa;
        while ((posizione != null) && !trovato) {
            E datiAllaPosizione = posizione.dati;
            if (datiAllaPosizione.equals(elemento))
                trovato = true;
            else
                posizione = posizione.collegamento;
        }
        return posizione;
    }

    public ArrayList<E> convertiInArrayList() {
        ArrayList<E> lista = new ArrayList<E>(lunghezza());
        NodoLista posizione = testa;
        while (posizione != null) {
            lista.add(posizione.dati);
            posizione = posizione.collegamento;
```

Le intestazioni dei costruttori non includono il tipo parametrico

```
        }
        return lista;
    }

    private class NodoLista {
        private E dati;
        private NodoLista collegamento;

        public NodoLista() {
            collegamento = null;
            dati = null;
        }

        public NodoLista(E nuoviDati, NodoLista valoreCollegamento) {
            dati = nuoviDati;
            collegamento = valoreCollegamento;
        }
    }
}
```

L'intestazione della inner class non ha il tipo parametrico

Tuttavia, il tipo parametrico è utilizzato dentro la definizione della inner class.

LISTATO 15.9 Utilizzo della lista concatenata parametrica.

```
import java.util.ArrayList;

public class ListaConcatenataDemo {

    public static void main(String[] args) {
        ListaConcatenata<String> listaDiStringhe =
                                        new ListaConcatenata<String>();
        listaDiStringhe.aggiungiNodoInTesta("Ciao");
        listaDiStringhe.aggiungiNodoInTesta("Arrivederci");
        listaDiStringhe.mostraLista();
        ListaConcatenata<Integer> listaDiNumeri =
                                        new ListaConcatenata<Integer>();
        for (int i = 0; i < 10; i++)
            listaDiNumeri.aggiungiNodoInTesta(i);
        listaDiNumeri.eliminaNodoDiTesta();
        ArrayList<Integer> lista = listaDiNumeri.convertiInArrayList();
        int dimensioneLista = lista.size();
        for (int posizione = 0; posizione < dimensioneLista; posizione++)
            System.out.print(lista.get(posizione) + " ");
        System.out.println();
    }
}
```

Il boxing automatico converte un `int` in un `Integer`.

Esempio di output

```
Arrivederci
Ciao
8 7 6 5 4 3 2 1 0
```

15.2 Varianti delle liste concatenate

In questo paragrafo verranno discusse alcune varianti delle liste concatenate, comprese due strutture dati note come pile (*stack*) e code (*queue*). Pile e code non richiedono necessariamente l'utilizzo delle liste concatenate, ma queste ultime costituiscono un modo comune per implementare le prime.

15.2.1 Liste concatenate doppie

Una lista concatenata ordinaria può essere percorsa in un'unica direzione (seguendo i collegamenti). Una **lista concatenata doppia** ha, per ogni nodo, un collegamento al nodo successivo e uno al nodo precedente. In alcuni casi, la disponibilità del collegamento al nodo precedente può semplificare la gestione della lista. Per esempio, non sarà più necessario utilizzare una variabile di istanza per tenere traccia del nodo dal quale si è arrivati a quello corrente. Una lista concatenata doppia può essere rappresentata come nella Figura 15.7.

La definizione di una classe nodo per una lista concatenata doppia potrebbe iniziare come segue:

```
public class NodoListaDoppia {
    public String dati;
    public NodoListaDoppia precedente;
    public NodoListaDoppia successivo;
    ...
```

Rispetto al caso della lista concatenata ordinaria, i costruttori e alcuni altri metodi richiederanno delle modifiche per la gestione del collegamento aggiuntivo. Le modifiche più rilevanti riguardano i metodi che aggiungono o eliminano dei nodi. Per maggiore chiarezza, si può aggiungere un costruttore che inizializzi entrambi i collegamenti:

```
public NodoListaDoppia(String nuoviDati, NodoListaDoppia nodoPrecedente,
                                         NodoListaDoppia nodoSuccessivo) {
    dati = nuoviDati;
    successivo = nodoSuccessivo;
    precedente = nodoPrecedente;
}
```

Per aggiungere un nuovo nodo all'inizio della lista è necessario modificare i collegamenti relativi a due nodi anziché uno solo. La procedura generale è illustrata nella Figura 15.8. Nel metodo `aggiungiAllInizio`, per prima cosa si crea un nuovo `NodoListaDoppia`. Dato che questo nodo verrà aggiunto all'inizio della lista, occorrerà impostare il collegamento al nodo precedente a `null` e quello al nodo successivo al nodo attualmente in testa alla lista:

```
NodoListaDoppia nuovaTesta = new NodoListaDoppia(nome, null, testa);
```

Successivamente, è necessario impostare il collegamento precedente del vecchio nodo di testa al nuovo nodo. Per fare questo, si può utilizzare l'istruzione `testa.precedente = nuovaTesta`, ma bisogna fare attenzione e assicurarsi che `testa` non sia `null` (cioè che la lista non sia vuota prima dell'aggiunta del nodo). Infine, si può impostare `testa` a `nuovaTesta`:

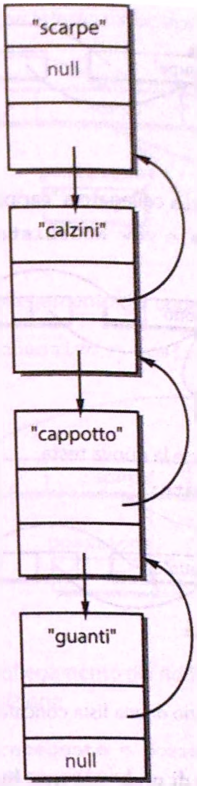


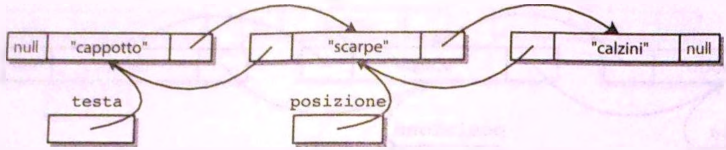
Figura 15.7 Una lista concatenata doppia.

```
if (testa != null) {
    testa.precedente = nuovaTesta;
}
testa = nuovaTesta;
```

Anche la cancellazione di un nodo da una lista concatenata doppia richiede l'aggiornamento dei riferimenti dei nodi da entrambi i lati di quello da eliminare. Grazie alla presenza del collegamento all'indietro, non è necessario utilizzare una variabile di istanza per tenere traccia del nodo precedente nella lista, come accadrebbe con una lista concatenata ordinaria. La procedura generale per l'eliminazione di un nodo referenziato dalla variabile `posizione` è mostrata nella Figura 15.9. Si noti che alcuni casi richiedono una gestione particolare, come quello dell'eliminazione di un nodo dall'inizio o dalla fine della lista.

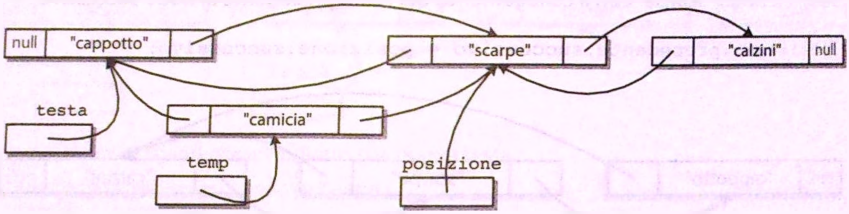
La procedura per l'inserimento di un nuovo nodo in una lista concatenata doppia è mostrata nella Figura 15.10. In questo caso, il nuovo nodo verrà inserito appena prima dell'elemento referenziato da `posizione`. Si noti che è anche necessario considerare come casi particolari della procedura di inserimento quelli nei quali si inserisce il nodo all'inizio o alla fine della lista. La Figura 15.10 mostra solo il caso generale di inserimento tra due nodi già esistenti.

1. Lista esistente con un iteratore che referenzia "scarpe"



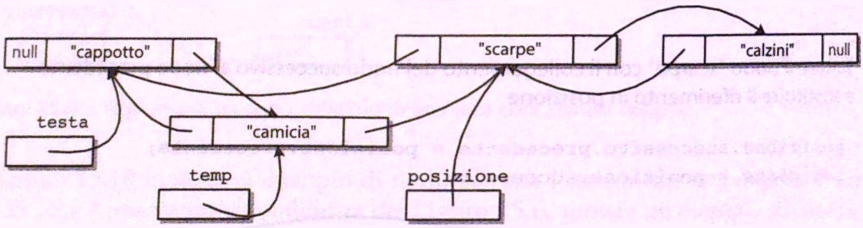
2. Creare un nuovo `NodoListaDoppia` con precedente collegato a "cappotto" e successivo a "scarpe"

```
NodoListaDoppia temp = new NodoListaDoppia(nuovoDato, posizione.precedente, posizione);
// nuovoDato = "camicia"
```



3. Impostare il collegamento successivo di "cappotto" al nuovo nodo "camicia"

```
posizione.precedente.successivo = temp;
```



4. Impostare il collegamento precedente di "scarpe" al nuovo nodo "camicia"

```
posizione.precedente = temp;
```

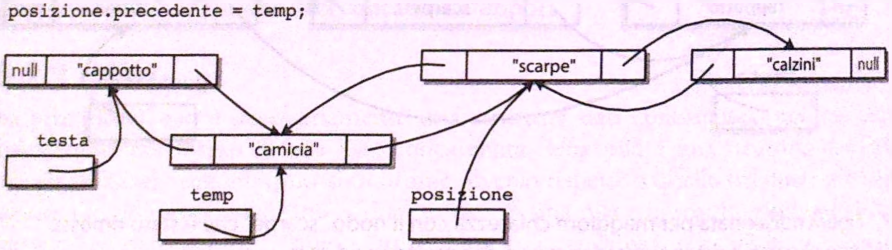


Figura 15.10 Inserire un nodo in una lista concatenata doppia.

Pile

Una pila è una struttura del tipo *last-in/first-out* (l'ultimo elemento inserito è il primo a essere rimosso). Cioè, gli elementi vengono estratti in ordine inverso a quello di inserimento.

LISTATO 15.10 Un nodo di una lista concatenata doppia.

MyLab

```
public class NodoListaDoppia {
   public String dati;
   public NodoListaDoppia precedente;
   public NodoListaDoppia successivo;

   public NodoListaDoppia(String nuoviDati,
                          NodoListaDoppia nodoPrecedente,
                          NodoListaDoppia nodoSuccessivo) {
      dati = nuoviDati;
      successivo = nodoSuccessivo;
      precedente = nodoPrecedente;
   }
}
```

LISTATO 15.11 Una lista concatenata doppia con un iteratore.

MyLab

```
/**
Lista concatenata doppia con iteratore. Un nodo è il "nodo corrente".
Inizialmente, il nodo corrente è il primo nodo.
*/
public class ListaConcatenataDoppiaDiStringheConIteratore {
   private NodoListaDoppia testa;
   private NodoListaDoppia corrente;

   public ListaConcatenataDoppiaDiStringheConIteratore() {
      testa = null;
      corrente = null;
   }

   public void aggiungiNodoInTesta(String datiDaAggiungere) {
      NodoListaDoppia nuovaTesta = new NodoListaDoppia(datiDaAggiungere,
                                                       null, testa);
      if (testa != null)
         testa.precedente = nuovaTesta;
      testa = nuovaTesta;
   }

   /**
   Imposta l'iteratore all'inizio della lista.
   */
   public void reimpostaIterazione() {
      corrente = testa;
   }

   /**
   Sposta l'iteratore al nodo successivo.
   */
```

```
    public void vaiAlSuccessivo() {
        if (corrente != null) {
            corrente = corrente.successivo;
        } else if (testa != null) {
            System.out.println("Si e' iterato troppe volte o" +
            " l'iterazione non e' stata" +
            " inizializzata.");
            System.exit(0);
        } else {
            System.out.println("Si sta iterando su una lista vuota.");
            System.exit(0);
        }
    }

    /**
    Inserisce un nuovo nodo che contiene nuoviDati dopo il nodo corrente.
    Il nodo corrente è uguale a quello prima dell'invocazione.
    Precondizione: la lista non è vuota; il nodo corrente non è oltre
    la fine della lista.
    */
    public void inserisciDopoNodoCorrente(String nuoviDati) {
        if (corrente != null) {
            NodoListaDoppia nuovoNodo = new NodoListaDoppia(nuoviDati,
                                corrente, corrente.successivo);
            corrente.successivo = nuovoNodo;

            if (corrente.successivo != null)
                corrente.successivo.precedente = nuovoNodo;
        } else if (testa != null) {
            System.out.println("Si sta inserendo quando iteratore" +
            " ha visitato tutti i nodi" +
            " o non e' stato inizializzato.");
            System.exit(0);
        } else {
            System.out.println("Si sta utilizzando" +
            " inserisciDopoNodoCorrente con" +
            " una lista vuota.");
            System.exit(0);
        }
    }

    /**
    Elimina il nodo corrente. Dopo l'invocazione,
    il nodo corrente è o il nodo successivo a quello eliminato
    o null se non c'è nessun nodo successivo.
    */
    public void eliminaNodoCorrente() {
        if (corrente != null) {
            if (corrente.successivo != null)
                corrente.successivo.precedente = corrente.precedente;
```

```
            else
                testa = corrente.successivo;
            corrente = corrente.successivo;
        } else { //corrente == null
            System.out.println("Si sta eliminando un nodo" +
            " corrente non inizializzato o la" +
            " lista e' vuota.");
            System.exit(0);
        }
    }

    <i metodi altriElementi, getDatiDaNodoCorrente e impostaDatiANodoCorrente
    sono gli stessi del Listato 15.6>
}
```

LISTATO 15.12 Utilizzo di una lista concatenata doppia con iteratore.

```
public class EsempioListaConcatenataDoppia {
    public static void main(String args[]) {
        ListaConcatenataDoppiaDiStringheConIteratore lista = new
                ListaConcatenataDoppiaDiStringheConIteratore();

        lista.aggiungiNodoInTesta("scarpe");
        lista.aggiungiNodoInTesta("aranciata");
        lista.aggiungiNodoInTesta("cappotto");

        System.out.println("La lista contiene:");
        lista.reimpostaIterazione();
        while (lista.altriElementi()) {
            System.out.println(lista.getDatiDaNodoCorrente());
            lista.vaiAlSuccessivo();
        }
        System.out.println();

        lista.reimpostaIterazione();
        lista.vaiAlSuccessivo();
        lista.vaiAlSuccessivo();
        System.out.println("Cancella " + lista.getDatiDaNodoCorrente());
        lista.eliminaNodoCorrente();

        System.out.println("La lista ora contiene:");
        lista.reimpostaIterazione();
        while (lista.altriElementi()) {
            System.out.println(lista.getDatiDaNodoCorrente());
            lista.vaiAlSuccessivo();
        }
```

```
        System.out.println();

        lista.reimpostaIterazione();
        System.out.println("Inserisci calzini dopo " +
                                lista.getDatiDaNodoCorrente());
        lista.inserisciDopoNodoCorrente("calzini");

        System.out.println("La lista ora contiene:");
        lista.reimpostaIterazione();
        while (lista.altriElementi()) {
            System.out.println(lista.getDatiDaNodoCorrente());
            lista.vaiAlSuccessivo();
        }
        System.out.println();
    }
}
```

Esempio di output

```
La lista contiene:
cappotto
aranciata
scarpe

Cancella scarpe
La lista ora contiene:
cappotto
aranciata

Inserisci calzini dopo cappotto
La lista ora contiene:
cappotto
calzini
aranciata
```

MyLab

LISTATO 15.13 Una classe `Pila`.

```
public class Pila {
    private NodoLista testa;

    public Pila() {
        testa = null;
    }

    /**
    Questo metodo sostituisce aggiungiNodoInTesta
    */
    public void push(String nuoviDati) {
        testa = new NodoLista(nuoviDati, testa);
    }
```

```
    /**
    Questo metodo sostituisce eliminaNodoDiTesta
    e restituisce la stringa in cima alla pila
    */
    public String pop() {
        String dati = "";

        if (testa != null) {
            dati = testa.getDati();
            testa = testa.getCollegamento();
        }
        return dati;
    }

    public boolean vuota() {
        return (testa == null);
    }
}
```

LISTATO 15.14 Esempio di utilizzo di una pila.

```
public class EsempioPila {
    public static void main(String args[]) {
        Pila pila = new Pila();
        pila.push("Alessandro");
        pila.push("Federico");
        pila.push("Marta");

        while (!pila.vuota()) {
            String s = pila.pop();
            System.out.println(s);
        }
    }
}
```

Gli elementi vengono rimossi dalla pila in ordine inverso rispetto a quello nel quale erano stati inseriti.

Esempio di output

```
Marta
Federico
Alessandra
```

15.2.3 Code

Una pila è una struttura dati nella quale l'ultimo elemento inserito è il primo a essere rimosso (*last-in/first-out*). Un'altra struttura dati di uso comune è la **coda** (*queue*), nella quale i dati sono gestiti in modo che il primo elemento inserito sia anche il primo a essere rimosso (*first-in/first-out*). Una coda è come una fila a uno sportello: i clienti arrivano alla fine della fila e vengono serviti quelli all'inizio. Anche una coda può essere implementata

per mezzo di una lista concatenata. Tuttavia, una coda richiede di tenere traccia sia della testa che della fine (cioè l'altro capo) della lista, poiché le azioni possono coinvolgere l'una o l'altra delle due posizioni. È più semplice rimuovere un nodo dalla testa che dalla fine di una lista concatenata. Pertanto, una semplice implementazione di una coda rimuoverà gli elementi dalla testa della lista e inserirà nuovi elementi alla fine.

La definizione di una semplice classe `Coda` basata su una lista concatenata è presentata nel Listato 15.15. Una breve dimostrazione del suo utilizzo è riportata nel Listato 15.16. In questo esempio la coda non è stata resa generica per semplicità, ma sostituire il tipo `String` con un tipo parametrico sarebbe molto facile.

Code

Una coda è una struttura del tipo *first-in/first-out* (il primo elemento inserito è il primo a essere rimosso). Cioè, gli elementi vengono estratti nello stesso ordine nel quale erano stati inseriti.

MyLab

LISTATO 15.15 Una classe Coda.

```
public class Coda {
    private NodoLista inizio;
    private NodoLista fine;

    public Coda() {
        inizio = null;
        fine = null;
    }

    /**
    Aggiunge una stringa alla fine della coda
    */
    public void aggiungi(String nuoviDati) {
        NodoLista nuovaFine = new NodoLista(nuoviDati, null);
        if (fine != null) {
            fine.setCollegamento(nuovaFine);
            fine = nuovaFine;
        } else {
            inizio = nuovaFine;
            fine = nuovaFine;
        }
    }

    /**
    Restituisce la stringa all'inizio della coda
    o null se la coda è vuota
    */
```

```
        else
            return null;
    }

    /**
    Elimina la stringa all'inizio della coda.
    Restituisce false se la coda è buota.
    */
    public boolean rimuovi() {
        if (inizio != null) {
            inizio = inizio.getCollegamento();
            return true;
        } else {
            return false;
        }
    }

    public boolean vuota() {
        return (inizio == null);
    }

    public void cancella() {
        inizio = null;
        fine = null;
    }
}
```

LISTATO 15.16 Esempio di utilizzo di una coda.

MyLab

```
public class EsempioCoda {
    public static void main(String args[]) {
        Coda coda = new Coda();

        coda.aggiungi("Alessandro");
        coda.aggiungi("Federico");
        coda.aggiungi("Marta");

        while (!coda.vuota()) {
            System.out.println(coda.prossimo());
            coda.rimuovi();
        }
        System.out.println("La coda è vuota.");
    }
}
```

Gli elementi vengono estratti dalla coda nello stesso ordine nel quale erano stati inseriti.

Esempio di output

```
Alessandro
Federico
Marta
La coda è vuota.
```

← Gli elementi vengono estratti dalla coda nello stesso ordine nel quale erano stati inseriti.

15.3 Tabelle di *hash*

Una tabella di ***hash***, o mappa di ***hash***, è una struttura dati che immagazzina e recupera dati in memoria in maniera efficiente. Esistono molti modi per costruire una tabella di *hash*; in questo paragrafo si utilizzerà la combinazione di un array e di una lista concatenata ordinaria. Mentre la ricerca di un elemento in una lista concatenata richiede in genere un tempo proporzionale alla lunghezza della lista, una tabella di *hash* può, potenzialmente, eseguire la ricerca in un numero fissato di operazioni, indipendentemente dalla quantità di dati in essa contenuti. Tuttavia, la particolare implementazione di tabella di *hash* che verrà presentata potrebbe richiedere comunque un tempo proporzionale alla quantità di dati, anche se solo in situazioni poco probabili.

Per immagazzinare un elemento in una tabella di *hash* gli si assegna una *chiave*. Nota la chiave, è possibile recuperare l'elemento. Idealmente, la chiave individua univocamente l'elemento associato. Se un elemento non fornisce intrinsecamente una chiave univoca, si può utilizzare una **funzione di *hash*** per generarne una. Nella maggior parte dei casi, una funzione di *hash* produce un numero.

Per esempio, si supponga di utilizzare una tabella di *hash* per immagazzinare le parole di un vocabolario. Una tabella di questo tipo potrebbe essere utile nella realizzazione di un correttore automatico: le parole che non sono presenti nel dizionario saranno state probabilmente scritte in modo scorretto. Si costruirà la tabella di *hash* mediante un singolo array di lunghezza fissa nel quale ogni elemento referenzia una lista concatenata. La chiave calcolata dalla funzione di *hash* associerà ogni parola all'indice corrispondente nell'array. I dati veri e propri saranno immagazzinati nella lista concatenata presente in quella posizione dell'array. La Figura 15.11 illustra l'idea nel caso di un array da 10 elementi. All'inizio, ogni elemento dell'array `arrayHash` contiene un riferimento a una lista concatenata vuota. Per prima cosa, si aggiunge la parola `"gatto"`, alla quale è stata associata la chiave, o **valore di *hash***, pari a 3 (si vedrà successivamente come è stato calcolato questo valore). Successivamente, si aggiungono `"cane"` e `"tartaruga"` con valori di *hash* rispettivamente 7 e 1. Ognuna di queste parole viene inserita in testa alla lista concatenata presente all'indice dell'array corrispondente al valore di *hash*. Infine, si aggiunge la parola `"uccello"`, anch'essa con valore di *hash* 3. Poiché all'indice 3 è già presente la parola `"gatto"`, si ottiene una cosiddetta **collisione**: sia `"uccello"` che `"gatto"` sono associate allo stesso indice dell'array. Quando si verifica una situazione di questo tipo, si effettua una **concatenazione**, cioè si aggiunge semplicemente un nuovo nodo alla lista concatenata già presente. In questo esempio, all'indice 3 sono ora presenti due nodi: `"uccello"` e `"gatto"`.

Per cercare un elemento da una tabella di *hash*, per prima cosa si calcola il suo valore di *hash*. Successivamente, si cerca, per mezzo di una ricerca sequenziale, l'elemento nella lista concatenata presente nella posizione dell'array di indice pari al valore di *hash* ottenuto. Se l'elemento non viene trovato in questa lista concatenata, allora non è presente

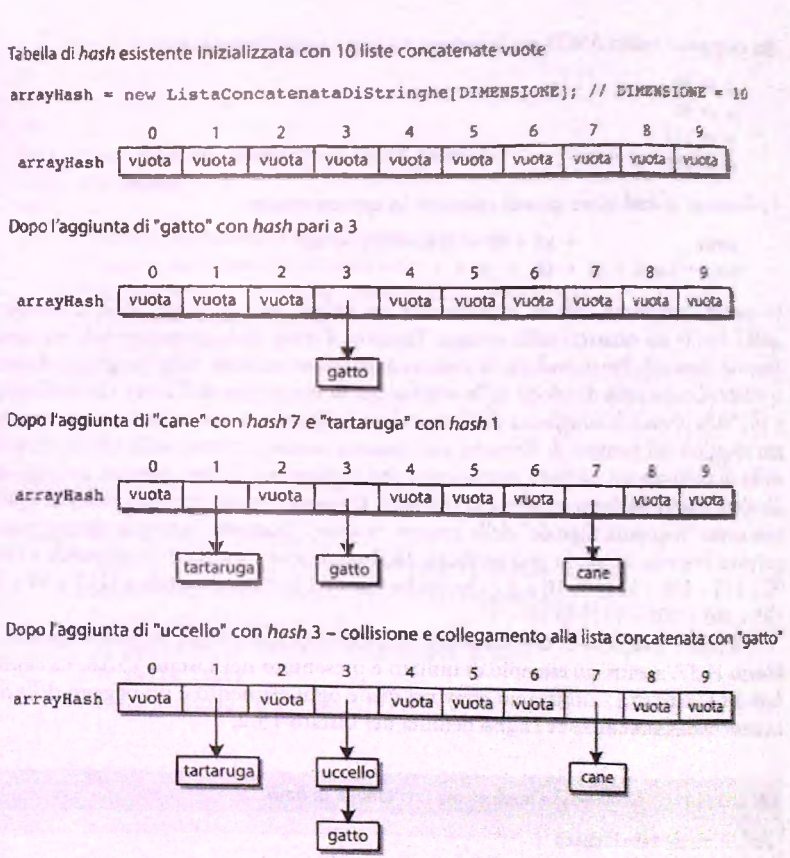


Figura 15.11 Costruzione di una tabella di *hash*.

nella tabella di *hash*. Se la dimensione della lista concatenata è piccola, questo processo sarà molto veloce.

15.3.1 Una funzione di *hash* per le stringhe

Un modo semplice per calcolare un valore di *hash* numerico per una stringa è quello di sommare i codici ASCII di ogni carattere della stringa e calcolare il resto della divisione della somma per la lunghezza dell'array. Di seguito è riportato un esempio di codice che implementa questa funzione:

```
private int calcolaHash(String s) {
    int hash = 0;
    for (int i = 0; i < s.length(); i++)
        hash += s.charAt(i);
    return hash % DIMENSIONE; // DIMENSIONE = 10 nell'esempio
}
```

Per esempio, i codici ASCII per la stringa "cane" sono i seguenti:

```
c -> 99
a -> 97
n -> 110
e -> 101
```

La funzione di *hash* viene quindi calcolata in questo modo:

```
somma              = 99 + 97 + 110 + 101 = 407
hash = somma % 10 = 407 % 10 = 7
```

In questo esempio, si calcola inizialmente un valore non limitato, cioè la somma dei codici ASCII dei caratteri nella stringa. Tuttavia, l'array può contenere solo un numero finito di elementi. Per ricondurre la somma a un valore minore della lunghezza dell'array, si calcola il resto della divisione della somma per la lunghezza dell'array, che nell'esempio è 10. Nella pratica, la lunghezza dell'array viene solitamente scelta pari a un numero primo maggiore del numero di elementi che saranno immagazzinati nella tabella di *hash* (la scelta di utilizzare un numero primo evita che compaiano fattori comuni nel resto della divisione che potrebbero produrre collisioni). Il valore 7 così ottenuto può essere utilizzato come "impronta digitale" della stringa "cane". Tuttavia, stringhe diverse possono generare lo stesso valore. Si può verificare facilmente che "gatto" corrisponde a (103 + 97 + 116 + 116 + 111) % 10 = 3 e che anche "uccello" corrisponde a (117 + 99 + 99 + 101 + 108 + 108 + 111) % 10 = 3.

Il codice completo di una classe che implementa una tabella di *hash* è riportato nel Listato 15.17, mentre un esempio di utilizzo è presentato nel Listato 15.18. La tabella di *hash* del Listato 15.17 utilizza un array nel quale ogni elemento è un oggetto della classe `ListaConcatenataDiStringhe` definita nel Listato 15.2.

MyLab

LISTATO 15.17 Una classe che implementa una tabella di *hash*.

```
public class TabellaHash {
    // Utilizza la classe per liste concatenate
    // ListaConcatenataDiStringhe del Listato 15.2

    private ListaConcatenataDiStringhe[] arrayHash;
    private static final int DIMENSIONE = 10;

    public TabellaHash() {
        arrayHash = new ListaConcatenataDiStringhe[DIMENSIONE];
        for (int i = 0; i < DIMENSIONE; i++)
            arrayHash[i] = new ListaConcatenataDiStringhe();
    }

    private int calcolaHash(String s) {
        int hash = 0;
        for (int i = 0; i < s.length(); i++) {
            hash += s.charAt(i);
```

```
        return hash % DIMENSIONE;
    }

    /**
    Restituisce true se obiettivo è nella tabella,
    false altrimenti.
    */
    public boolean contieneStringa(String obiettivo) {
        int hash = calcolaHash(obiettivo);
        ListaConcatenataDiStringhe lista = arrayHash[hash];
        if (lista.nellaLista(obiettivo))
            return true;
        return false;
    }

    /**
    Salva la stringa s nella tabella di hash
    */
    public void aggiungi(String s) {
        int hash = calcolaHash(s); // Calcola il valore di hash
        ListaConcatenataDiStringhe lista = arrayHash[hash];
        if (!lista.nellaLista(s)) {
            // Aggiunge la stringa solo se non è già
            // nella lista
            lista.aggiungiNodoInTesta(s);
        }
    }
}
```

LISTATO 15.18 Esempio di utilizzo di una tabella di *hash*. MyLab

```
public class EsempioTabellaHash {
    public static void main(String args[]) {
        TabellaHash h = new TabellaHash();

        System.out.println("Aggiunta di cane, gatto,
                            tartaruga, uccello");
        h.aggiungi("cane");
        h.aggiungi("gatto");
        h.aggiungi("tartaruga");
        h.aggiungi("uccello");

        System.out.println("Contiene cane? " +
              h.contieneStringa("cane"));
        System.out.println("Contiene gatto? " +
              h.contieneStringa("gatto"));
```

```
        System.out.println("Contiene tartaruga? " +
                h.contieneStringa("tartaruga"));
        System.out.println("Contiene uccello? " +
                h.contieneStringa("uccello"));
        System.out.println("Contiene pesce? " +
                h.contieneStringa("pesce"));
        System.out.println("Contiene mucca? " +
                h.contieneStringa("mucca"));
    }
}
```

Esempio di output

```
Aggiunta di cane, gatto, tartaruga, uccello
Contiene cane? true
Contiene gatto? true
Contiene tartaruga? true
Contiene uccello? true
Contiene pesce? false
Contiene mucca? false
```

15.3.2 Efficienza delle tabelle di *hash*

L'efficienza dell'implementazione di una tabella di *hash* presentata sopra dipende da vari fattori. Per prima cosa, è utile analizzare due casi estremi. Le prestazioni peggiori si avranno se tutti gli elementi inseriti vengono associati alla stessa chiave. In questo caso, tutti gli elementi saranno immagazzinati in un'unica lista e quindi l'operazione di ricerca richiederà, in generale, un tempo proporzionale al numero di elementi nella tabella. Fortunatamente, se gli elementi da inserire hanno una distribuzione in qualche modo casuale, sarà molto improbabile che vengano associati tutti alla stessa chiave. All'estremo opposto, le prestazioni migliori si avrebbero se ogni elemento venisse associato a una chiave diversa. In questo caso non si avrebbero collisioni e l'operazione di ricerca richiederebbe sempre lo stesso tempo, dato che l'elemento da trovare sarebbe sempre il primo nodo della lista concatenata.

È possibile ridurre la probabilità che si verifichino delle collisioni utilizzando una funzione di *hash* migliore. Per esempio, la semplice funzione di *hash* presentata precedentemente, che somma i codici dei caratteri di una stringa, non tiene in considerazione l'ordine di tali caratteri. Le parole "attore" e "teatro", per esempio, verrebbero associate alla stessa chiave. Una funzione di *hash* migliore per una stringa `s` può essere ottenuta moltiplicando il valore numerico di ogni lettera per un peso che cresce con la posizione della lettera nella stringa. Per esempio,

```
int hash = 0;
for (int i = 0; i < s.length(); i++) {
    hash = 31 * hash + s.charAt(i);
}
```

Un altro modo per ridurre la probabilità di collisioni è aumentare la dimensione della tabella di *hash*. Per esempio, inserendo 1000 elementi in una tabella da 10000 posti la

probabilità di collisione è molto più bassa che se la tabella avesse solo 1000 posti. Tuttavia, uno svantaggio dell'utilizzo di tabelle molto grandi è rappresentato dallo spreco di memoria: se in una tabella da 10000 posti si inseriscono solo 1000 elementi, almeno 9000 posizioni di memoria saranno inutilizzate. Questo esempio mostra che è sempre necessario stabilire un compromesso tra tempo di elaborazione e spazio di memoria occupato. In generale, è possibile migliorare le prestazioni di esecuzioni a patto di utilizzare più memoria e viceversa.

15.4 Insiemi

Un insieme è una collezioni di elementi dei quali si ignorano ordine e molteplicità. Molti problemi in informatica possono essere affrontati utilizzando strutture dati di tipi insieme. Un modo semplice di implementare un insieme consiste in una variante di una lista concatenata. In questa implementazione, gli elementi dell'insieme vengono immagazzinati in una lista concatenata ordinaria. Il Listato 15.19 presenta un'implementazione completa. La classe `Nodo` è una *inner class*, come spiegato nel Paragrafo 15.1.5. La lista concatenata è simile alla lista generica presentata nel Listato 15.8. In effetti, le operazioni `nellInsieme`, `mostraInsieme` e `numeroElementi` sono virtualmente identiche alle operazioni corrispondenti per una lista concatenata. Il metodo `aggiungi`, che sostituisce `aggiungiNodoInTesta`, deve essere modificato leggermente per impedire che vengano inseriti elementi duplicati. La Figura 15.12 mostra due esempi di insiemi realizzati mediante questa struttura dati. L'insieme rotondi contiene `"piselli"`, `"palla"` e `"torta"`, mentre l'insieme `verdi` contiene `"piselli"` ed `"erba"`. Poiché una lista concatenata contiene riferimenti agli elementi dell'insieme, è possibile includere lo stesso elemento in più insiemi referenziandolo da più liste concatenate. Per esempio, nella Figura 15.12 `"piselli"` appartiene a entrambi gli insiemi.

LISTATO 15.19 Una classe `Insieme<T>`.

MyLab

```
// Utilizza una lista concatenata come struttura dati
// interna per immagazzinare gli elementi di un insieme
public class Insieme<T> {
   private class Nodo<T> {
       private T dati;
       private Nodo<T> collegamento;

       public Nodo() {
           dati = null;
           collegamento = null;
       }

       public Nodo(T nuoviDati, Nodo<T> valoreCollegamento) {
           dati = nuoviDati;
           collegamento = valoreCollegamento;
       }
   } // Fine della inner class Nodo<T>
```

```
    private Nodo<T> testa;

    public Insieme() {
        testa = null;
    }

    /**
    Aggiunge un nuovo elemento all'insieme.
    Se l'elemento è già presente, restituisce false,
    altrimenti restituisce true.
    */
    public boolean aggiungi(T nuovoElemento) {
        if (!nellInsieme(nuovoElemento)) {
            testa = new Nodo<T>(nuovoElemento, testa);
            return true;
        }
        return false;
    }

    public boolean nellInsieme(T elemento) {
        Nodo<T> posizione = testa;
        T elementoAllaPosizione;

        while (posizione != null) {
            elementoAllaPosizione = posizione.dati;
            if (elementoAllaPosizione.equals(elemento))
                return true;
            posizione = posizione.collegamento;
        }
        return false; // l'elemento non è stato trovato
    }

    public void mostraInsieme() {
        Nodo<T> posizione = testa;
        while (posizione != null) {
            System.out.print(posizione.dati.toString() + " ");
            posizione = posizione.collegamento;
        }
        System.out.println();
    }

    /**
    Restituisce un nuovo insieme corrispondente all'unione
    dell'insieme con l'insieme specificato.
    */
    public Insieme<T> unione(Insieme<T> altroInsieme) {
        Insieme<T> insiemeUnione = new Insieme<T>();
```

```
        // Copia l'insieme corrente nell'unione
        Nodo<T> posizione = testa;
        while (posizione != null) {
            insiemeUnione.aggiungi(posizione.dati);
            posizione = posizione.collegamento;
        }

        // Copia l'altro insieme nell'unione.
        // Il metodo aggiungi elimina automaticamente
        // i duplicati.
        posizione = altroInsieme.testa;
        while (posizione != null) {
            insiemeUnione.aggiungi(posizione.dati);
            posizione = posizione.collegamento;
        }
        return insiemeUnione;
    }

    /**
    Restituisce un nuovo insieme corrispondente all'intersezione
    tra l'insieme e l'insieme specificato.
    */
    public Insieme<T> intersezione(Insieme<T> altroInsieme) {
        Insieme<T> insiemeIntersezione = new Insieme<T>();
        // Copia solo gli elementi presenti in entrambi
        // gli insiemi.
        Nodo<T> posizione = testa;
        while (posizione != null) {
            if (altroInsieme.nellInsieme(posizione.dati))
                insiemeIntersezione.aggiungi(posizione.dati);
            posizione = posizione.collegamento;
        }
        return insiemeIntersezione;
    }

    public int numeroElementi() {
        int conteggio = 0;
        Nodo<T> posizione = testa;
        while (posizione != null) {
            conteggio++;
            posizione = posizione.collegamento;
        }
        return conteggio;
    }
}
```

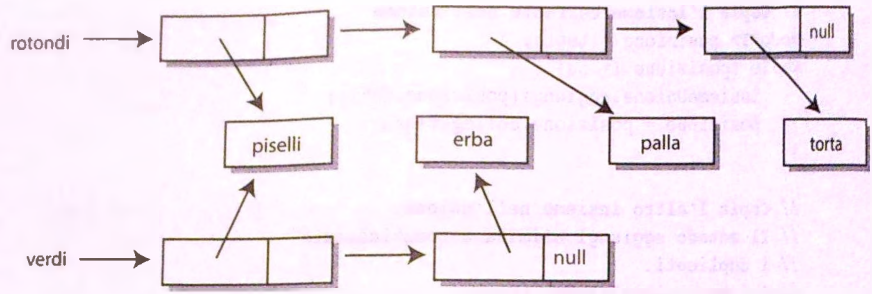


Figura 15.12 Insiemi realizzati mediante liste concatenate.

15.4.1 Operazioni di base sugli insiemi

Le operazioni di base che una classe che implementi un insieme dovrebbe consentire sono le seguenti:

- aggiunta di un elemento;
- verifica dell'appartenenza di un elemento all'insieme;
- unione;
- intersezione.

Sarebbe inoltre opportuno fornire un iteratore per estrarre gli elementi dall'insieme. Ciò è lasciato come esercizio al lettore (si veda il Progetto 10). Altre operazioni utili comprendono metodi per determinare la cardinalità dell'insieme e per rimuovere elementi.

Il metodo `aggiungi` del Listato 15.19 è simile al metodo `aggiungiNodoInTesta` delle liste concatenate. La variabile `testa` referenzia sempre il primo nodo della lista. Il metodo `nellInsieme` è identico al metodo `nellaLista` per le liste concatenate ordinarie: si scorrono semplicemente tutti gli elementi della lista cercando l'elemento da trovare.

Il metodo `unione` combina gli elementi dell'oggetto chiamante con quelli dell'insieme `altroInsieme` passato come argomento. Per ottenere l'unione, per prima cosa si crea un nuovo oggetto di tipo `Insieme<T>` vuoto. Successivamente, si itera su tutti gli elementi dell'oggetto chiamante e dell'argomento. Tutti gli elementi vengono aggiunti al nuovo insieme. Il metodo `aggiungi` garantisce che non vengano inseriti elementi duplicati, quindi non è necessario effettuare il controllo esplicitamente.

Il metodo `intersezione` crea anch'esso un nuovo insieme vuoto. In questo caso, il nuovo insieme viene popolato con gli elementi che appartengono sia all'oggetto chiamante che all'insieme `altroInsieme` passato come argomento. Per fare questo, si itera su tutti gli elementi dell'oggetto chiamante e si invoca per ognuno il metodo `nellInsieme` di `altroInsieme`. Se questo metodo restituisce `true`, l'elemento viene aggiunto all'insieme intersezione.

Un semplice esempio di utilizzo della classe è presentato nel Listato 15.20.

LISTATO 15.20 Esempio di utilizzo della classe `Insieme<T>`.

MyLab

```
public class EsempioInsieme {
   public static void main(String args[]) {
      // Oggetti rotondi
      Insieme<String> rotondi = new Insieme<String>();
      // Oggetti verdi
      Insieme<String> verdi = new Insieme<String>();

      // Aggiunta di elementi agli insiemi
      rotondi.aggiungi("piselli");
      rotondi.aggiungi("palla");
      rotondi.aggiungi("torta");
      rotondi.aggiungi("acini");

      verdi.aggiungi("piselli");
      verdi.aggiungi("tubo");
      verdi.aggiungi("erba");

      System.out.println("Contenuto dell'insieme rotondi:");
      rotondi.mostraInsieme();
      System.out.println();
      System.out.println("Contenuto dell'insieme verdi:");
      verdi.mostraInsieme();
      System.out.println();

      System.out.println("palla appartiene all'insieme rotondi? " +
                         rotondi.nellInsieme("palla"));
      System.out.println("palla appartiene all'insieme verdi? " +
                         verdi.nellInsieme("palla"));
      System.out.println("palla e piselli appartengono allo stesso insieme? " +
          ((rotondi.nellInsieme("palla") && rotondi.nellInsieme("piselli")) ||
          (verdi.nellInsieme("palla") && verdi.nellInsieme("piselli"))));

      System.out.println("torta e erba appartengono allo stesso insieme? " +
          ((rotondi.nellInsieme("torta") && rotondi.nellInsieme("erba")) ||
          (verdi.nellInsieme("torta") && verdi.nellInsieme("erba"))));

      System.out.print("Unione di verdi e rotondi: ");
      rotondi.unione(verdi).mostraInsieme();

      System.out.print("Intersezione di verdi e rotondi: ");
      rotondi.intersezione(verdi).mostraInsieme();
}
```

Esempio di output

```
Contenuto dell'insieme rotondi:
acini torta palla piselli

Contenuto dell'insieme verdi:
erba tubo piselli

palla appartiene all'insieme rotondi? true
palla appartiene all'insieme verdi? false
palla e piselli appartengono allo stesso insieme? true
torta e erba appartengono allo stesso insieme? false
Unione di verdi e rotondi: tubo erba piselli palla torta acini
Intersezione di verdi e rotondi: piselli
```

15.4.2 Efficienza degli insiemi realizzati mediante liste concatenate

L'efficienza dell'implementazione di insieme presentata può essere valutata in relazione alle operazioni di base. Il metodo `nellInsieme` scorre l'intero insieme cercando l'elemento da trovare, cosa che richiede un numero di operazioni proporzionale al numero di elementi nell'insieme. L'aggiunta di un elemento inserisce un nuovo nodo all'inizio della lista, il che richiede un numero costante di operazioni, indipendentemente dal numero di elementi nell'insieme. L'esecuzione del metodo `aggiungi` richiederà però un tempo proporzionale al numero di elementi, dato che il metodo deve verificare che l'elemento non sia già presente nell'insieme. Quando si invoca il metodo `unione` su due insiemi, si itera sugli elementi di entrambi gli insiemi aggiungendo ognuno di essi. Se i due insiemi contengono rispettivamente n ed m elementi, si avranno in totale $(n+m)$ chiamate al metodo `aggiungi`. Tuttavia, anche in questo caso ci sono delle chiamate al metodo `nellInsieme` nascoste in quelle al metodo `aggiungi`. Complessivamente, il tempo necessario per costruire l'insieme `unione` cresce approssimativamente come $(n+m)^2$, cioè come il quadrato della somma dei numeri n ed m di elementi dei due insiemi. Infine, il metodo `intersezione` applicato a due insiemi invoca il metodo `nellInsieme` del secondo insieme a ogni elemento del primo. Poiché l'esecuzione del metodo `nellInsieme` richiede un tempo proporzionale al numero di elementi del secondo insieme per ogni elemento del primo, il tempo complessivo necessario è proporzionale al prodotto dei numeri di elementi dei due insiemi. L'implementazione di insieme presentata sopra è quindi in generale poco efficiente. Un approccio diverso alla gestione degli insiemi, basato per esempio sull'utilizzo di tabelle di *hash* al posto delle liste concatenate, potrebbe consentire di ottenere un metodo `intersezione` che richieda solo un tempo proporzionale alla somma dei numeri di elementi dei due insiemi. L'implementazione basata sulle liste concatenate sarà comunque adeguata per applicazioni che utilizzino insiemi con pochi elementi o nelle quali si utilizzi poco l'operazione di intersezione e ha il valore aggiunto della semplicità del codice.

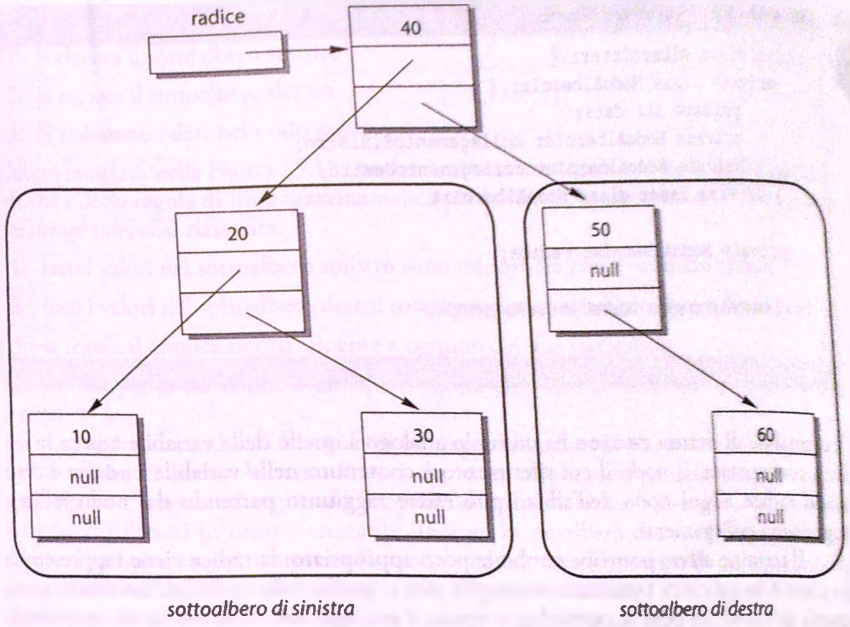


Figura 15.13 Un albero binario.

15.5 Alberi

Un albero è un esempio di struttura dati concatenata di tipo più complesso rispetto a quelli visti fino a questo punto. Si tratta di un tipo di struttura dati molto utilizzato, pertanto verranno ora presentate le tecniche generali per costruire e utilizzare gli alberi.

15.5.1 Proprietà degli alberi

Un albero è una struttura dati organizzata come mostrato nella Figura 15.13. In particolare, in un albero qualunque nodo può essere raggiunto partendo dal nodo radice (*root*) lungo un percorso costituito da una sequenza di collegamenti. Si noti che non sono ammessi percorsi chiusi (cicli) in un albero. Seguendo i collegamenti, si arriverà quindi necessariamente a una "fine". La definizione di una classe che implementa un albero di questo tipo è riportata nel Listato 15.21. Si noti che ogni nodo ha due collegamenti a altri nodi. Questo tipo di albero è detto **albero binario**, perché da ogni nodo partono esattamente due collegamenti. Sono possibili anche altri tipi di alberi con numeri diversi di collegamenti, ma quello binario è il tipo più comune.

MyLab

LISTATO 15.21 Un albero binario.

```
public class AlberoInteri {
    private class NodoAlberoInt {
        private int dati;
        private NodoAlberoInt collegamentoSinistro;
        private NodoAlberoInt collegamentoDestro;
    } // Fine inner class NodoAlberoInt

    private NodoAlberoInt radice;

    <I metodi non sono mostrati in questo esempio>
}
```

La variabile di istanza `radice` ha un ruolo analogo a quello della variabile `testa` in una lista concatenata. Il nodo il cui riferimento è contenuto nella variabile `radice` è detto **nodo radice**. Ogni nodo dell'albero può essere raggiunto partendo dal nodo radice e seguendo i collegamenti.

Il termine *albero* potrebbe sembrare poco appropriato: la radice viene rappresentata in cima e la struttura ramificata assomiglia più a quella delle radici di un albero che a quella dei rami. Se però si capovolge la figura, l'analogia con la struttura di un normale albero è evidente. I nodi alla fine dei rami, che hanno entrambe le variabili di istanza che rappresentano i collegamenti impostate a `null`, sono detti **nodi foglia**. In analogia a una lista concatenata vuota, un **albero vuoto** ha la variabile radice impostata a `null`.

Si noti che un albero ha una struttura ricorsiva. Ogni albero binario contiene, infatti, due sottoalberi che hanno come radici i nodi referenziati rispettivamente dalle variabili `collegamentoSinistro` e `collegamentoDestro` del nodo `radice`. Questi due sottoalberi sono evidenziati nella Figura 15.13. Questa naturale struttura ricorsiva rende gli alberi particolarmente indicati per l'utilizzo con algoritmi ricorsivi. Per esempio, si consideri il problema della ricerca in un albero, nella quale si visita ciascun nodo eseguendo qualche operazione sui dati del nodo (per esempio, si visualizzano i dati sullo schermo). Un approccio generale al problema potrebbe essere il seguente:

Elaborazione pre-ordine (*preorder*)

1. Si elaborano i dati nel nodo radice
2. Si elabora il sottoalbero sinistro
3. Si elabora il sottoalbero destro

Si possono anche considerare varianti a questo approccio. Due possibili alternative sono:

Elaborazione in-ordine (*inorder*)

1. Si elabora il sottoalbero sinistro
2. Si elaborano i dati nel nodo radice
3. Si elabora il sottoalbero destro

Elaborazione post-ordine (*postorder*)

1. Si elabora il sottoalbero sinistro
2. Si elabora il sottoalbero destro
3. Si elaborano i dati nel nodo radice

L'albero mostrato nella Figura 15.13 contiene numeri immagazzinati in un modo particolare che è detto **regola di immagazzinamento per la ricerca in alberi binari** (*binary search tree storage rule*) così riassunta:

1. tutti i valori nel sottoalbero sinistro sono minori del valore nel nodo radice;
2. tutti i valori nel sottoalbero destro sono maggiori o uguali al valore nel nodo radice;
3. la regola si applica ricorsivamente a ognuno dei due sottoalberi.

(Il caso base per la ricorsione è un albero vuoto, che viene considerato compatibile con la regola.)

Un albero che soddisfa la regola di immagazzinamento per la ricerca in alberi binari è detto **albero di ricerca binaria**. Si noti che se un albero soddisfa questa regola e se ne visualizzano gli elementi seguendo l'approccio dell'elaborazione in-ordine, gli elementi saranno visualizzati in ordine crescente. Inoltre, in un albero che soddisfa la regola gli elementi possono essere recuperati in modo molto veloce per mezzo di un algoritmo di ricerca binaria simile a quello presentato per gli array nel Listato 7.7. Il tema della ricerca e della gestione di un albero di ricerca binaria che massimizzino l'efficienza è molto ampio e va al di là degli scopi di questo testo. Tuttavia, verrà presentato un esempio di classe che implementa un albero che soddisfa la regola descritta sopra.

ESEMPIO DI PROGRAMMAZIONE
UNA CLASSE PER ALBERI DI RICERCA BINARIA

Il Listato 15.22 contiene la definizione di una classe che implementa un albero per la ricerca binaria che soddisfa la regola di immagazzinamento per la ricerca in alberi binari. Per semplicità, la classe presentata gestisce numeri interi. Il Listato 15.23 illustra un esempio di utilizzo della classe. Si noti che indipendentemente dall'ordine nel quale gli elementi sono inseriti, essi verranno stampati in ordine crescente.

I metodi di questa classe fanno un uso intensivo della natura ricorsiva degli alberi binari. Se `unNodo` è un riferimento a un qualunque nodo dell'albero (compreso, eventualmente, il nodo radice), il sottoalbero che ha `unNodo` come radice può essere scomposto in tre parti:

1. il singolo nodo `unNodo`;
2. il sottoalbero sinistro avente come radice `unNodo.collegamentoSinistro`;
3. il sottoalbero destro avente come radice `unNodo.collegamentoDestro`.

Anche i sottoalberi sinistro e destro soddisfano la regola di immagazzinamento, pertanto è naturale utilizzare la ricorsione per elaborare l'intero albero come segue:

1. si elabora il sottoalbero sinistro con radice `unNodo.collegamentoSinistro`;
2. si elabora il nodo singolo `unNodo`;
3. si elabora il sottoalbero destro con radice `unNodo.collegamentoDestro`.

Si noti che il nodo radice viene elaborato dopo il sottoalbero sinistro (elaborazione inordine). Ciò garantisce che i numeri nell'albero vengano stampati in ordine crescente. Il metodo `mostraSottoAlbero` sfrutta un'implementazione diretta di questa tecnica. Altri metodi sono leggermente più complicati perché richiedono l'elaborazione di uno solo dei due sottoalberi. Per esempio, si consideri il metodo `nelSottoAlbero`, che restituisce `true` o `false` a seconda che l'elemento passato come argomento sia o meno nel sottoalbero che ha `radiceSottoAlbero` come nodo radice. L'algoritmo per il metodo `nelSottoAlbero` si scrive, in pseudocodice, nel modo seguente:

```
if (Il nodo radice radiceSottoAlbero è vuoto)
    return false;
else if (Il nodo radiceSottoAlbero contiene l'elemento)
    return true;
else if (elemento < radiceSottoAlbero.dati)
    return (Il risultato della ricerca nell'albero avente come
                                radice radiceSottoAlbero.collegamentoSinistro)
else // elemento > radiceSottoAlbero.dati
    return (Il risultato della ricerca nell'albero avente come
                                radice radiceSottoAlbero.collegamentoDestro)
```

Il motivo grazie al quale questo algoritmo produce il risultato corretto è che l'albero soddisfa la regola di immagazzinamento per la ricerca in alberi binari, per cui si sa che se

```
elemento < radiceSottoAlbero.dati
```

allora `elemento`, se è presente nell'albero, deve trovarsi necessariamente nel sottoalbero sinistro, mentre se

```
elemento > radiceSottoAlbero.dati
```

allora `elemento`, se è nell'albero, può essere solo nel sottoalbero destro.
Il seguente metodo utilizza una tecnica molto simile a quella appena descritta:

```
private NodoAlberoInt inserisciInSottoAlbero(int elemento,
                                             NodoAlberoInt radiceSottoAlbero)
```

Tuttavia, qui compare anche un aspetto nuovo: si vuole che il metodo `inserisciInSottoAlbero` inserisca un nuovo nodo, avente come dato l'elemento specificato, nell'albero che ha come nodo radice `radiceSottoAlbero`. Ma in questo caso è necessario trattare `radiceSottoAlbero` come una variabile anziché limitarsi a leggere il valore che contiene. Per esempio, se `radiceSottoAlbero` contiene il valore `null`, occorrerà modificare tale valore in modo che la variabile referenzi un nuovo nodo che contenga l'elemento da inserire. Non è però possibile, in Java, modificare il valore di una variabile passata come argomento in modo che la modifica abbia effetto anche al di fuori del metodo stesso. Di conseguenza, è necessario agire in modo un po' diverso. Per poter cambiare il valore della variabile `radiceSottoAlbero`, si restituisce un riferimento al nuovo valore da assegnare e si invoca il metodo in questo modo:

```
radiceSottoAlbero = inserisciInSottoAlbero(elemento, radiceSottoAlbero);
```

Ciò spiega perché il metodo `inserisciInSottoAlbero` restituisca un riferimento a un nodo dell'albero, ma non è ancora ovvio perché il nodo restituito sia la radice del

sottoalbero (modificato) desiderato. Si noti che il metodo `inserisciInSottoAlbero` effettua una ricerca nell'albero esattamente come il metodo `nelSottoAlbero`, ma non si ferma una volta trovato l'elemento cercato; al contrario, prosegue finché non raggiunge un nodo foglia, cioè un nodo contenente `null`. Questo è il punto nel quale va inserito l'elemento e quindi `null` viene sostituito con un nuovo sottoalbero composto da un unico nodo che contiene il nuovo elemento. Potrebbe essere necessario studiare a fondo il metodo per convincersi che il suo funzionamento è corretto. In particolare, è importante assicurarsi di aver capito perché dopo un'aggiunta, come la seguente

```
radiceSottoAlbero = inserisciInSottoAlbero(elemento, radiceSottoAlbero);
```

l'albero con radice `radiceSottoAlbero` soddisferà ancora la regola di immagazzinamento per la ricerca in alberi binari.
Il resto della definizione della classe non presenta caratteristiche di particolare interesse.

LISTATO 15.22 Un albero di ricerca binaria per numeri interi.

```
public class AlberoInteri {
   private class NodoAlberoInt {
      private int dati;
      private NodoAlberoInt collegamentoSinistro;
      private NodoAlberoInt collegamentoDestro;

      public NodoAlberoInt(int nuoviDati, NodoAlberoInt
         nuovoCollegamentoSinistro, NodoAlberoInt nuovoCollegamentoDestro) {
         dati = nuoviDati;
         collegamentoSinistro = nuovoCollegamentoSinistro;
         collegamentoDestro = nuovoCollegamentoDestro;
      }
   } // Fine della inner class NodoAlberoInt

   private NodoAlberoInt radice;

   public AlberoInteri() {
      radice = null;
   }

   public void aggiungi(int elemento) {
      radice = inserisciInSottoAlbero(elemento, radice);
   }

   public boolean nellAlbero(int elemento) {
      return nelSottoAlbero(elemento, radice);
   }

   public void mostraAlbero() {
      mostraSottoAlbero(radice);
   }
```

Questa classe dovrebbe avere anche altri metodi. Quelli presentati sono soltanto un esempio.

```
/**
Restituisce il nodo radice dell'albero avente come radice radiceSottoAlbero
e nel quale è stato inserito un nodo con il nuovo elemento specificato.
*/
private NodoAlberoInt inserisciInSottoAlbero(int elemento,
                      NodoAlberoInt radiceSottoAlbero) {
    if (radiceSottoAlbero == null)
        return new NodoAlberoInt(elemento, null, null);
    else if (elemento < radiceSottoAlbero.dati) {
        radiceSottoAlbero.collegamentoSinistro =
                    inserisciInSottoAlbero(elemento,
                    radiceSottoAlbero.collegamentoSinistro);
        return radiceSottoAlbero;
    } else { // elemento >= radiceSottoAlbero.dati
        radiceSottoAlbero.collegamentoDestro =
                    inserisciInSottoAlbero(elemento,
                    radiceSottoAlbero.collegamentoDestro);
        return radiceSottoAlbero;
    }
}

private static boolean nelSottoAlbero(int elemento,
                      NodoAlberoInt radiceSottoAlbero) {
    if (radiceSottoAlbero == null)
        return false;
    else if (radiceSottoAlbero.dati == elemento)
        return true;
    else if (elemento < radiceSottoAlbero.dati)
        return nelSottoAlbero(elemento,
                      radiceSottoAlbero.collegamentoSinistro);
    else // elemento >= radiceSottoAlbero.dati
        return nelSottoAlbero(elemento,
                      radiceSottoAlbero.collegamentoDestro);
}

private static void mostraSottoAlbero(NodoAlberoInt radiceSottoAlbero) {
    if (radiceSottoAlbero != null) {
        mostraSottoAlbero(radiceSottoAlbero.collegamentoSinistro);
        System.out.print(radiceSottoAlbero.dati + " ");
        mostraSottoAlbero(radiceSottoAlbero.collegamentoDestro);
    }    // altrimenti non fare niente:
        // un albero vuoto non ha elementi da mostrare
}
}
```

LISTATO 15.23 Esempio di utilizzo dell'albero di ricerca binaria.

```
import java.util.Scanner;

public class EsempioAlberoRicercaBinaria {
   public static void main(String args[]) {
       Scanner tastiera = new Scanner(System.in);
       AlberoInteri albero = new AlberoInteri();

       System.out.println("Inserire una lista di interi non negativi");
       System.out.println("Inserire un intero negativo alla fine.");

       int prossimo = tastiera.nextInt();
       while (prossimo >= 0) {
           albero.aggiungi(prossimo);
           prossimo = tastiera.nextInt();
       }

       System.out.println("Gli elementi in ordine crescente sono:");
       albero.mostraAlbero();
   }
}
```

Esempio di output

```
Inserire una lista di interi non negativi
Inserire un intero negativo alla fine.
40
30
20
10
11
22
33
44
-1
Gli elementi in ordine crescente sono:
10 11 20 22 30 33 40 44
```

15.5.2 Efficienza degli alberi di ricerca binaria

Quando si effettua una ricerca in un albero che sia il più corto possibile (cioè nel quale le lunghezze dei percorsi dal nodo radice a un qualunque nodo foglia differiscano al più di una unità), il metodo di ricerca `nelSottoAlbero` e, di conseguenza, anche il metodo `nellAlbero`, sono all'incirca tanto efficienti quanto l'algoritmo di ricerca binaria in un array ordinato (Listato 7.7). Questo fatto non dovrebbe sorprendere, dato che i due

algoritmi sono molto simili. Nel caso peggiore, il tempo necessario per la ricerca risulterà proporzionale al logaritmo del numero di nodi nell'albero. Ciò significa che la ricerca in un albero binario corto è molto efficiente. Per ottenere la massima efficienza, non è necessario che l'albero sia esattamente il più corto possibile, a patto che non si discosti troppo da tale condizione. Mano a mano che l'albero diventa sempre meno corto e largo e sempre più lungo e sottile, l'efficienza si riduce sempre di più, finché nel caso estremo diventa pari a quella della ricerca in una lista concatenata con lo stesso numero di nodi dell'albero.

Le tecniche di gestione di un albero volte a mantenerlo corto e largo (il termine tecnico è *bilanciato*) quando vengono aggiunti nuovi elementi va al di là degli scopi di questo testo. Qui ci si limiterà a segnalare il fatto che se gli elementi da immagazzinare nell'albero si presentano in modo casuale, l'albero risulterà naturalmente sufficientemente bilanciato da presentare le proprietà di massima efficienza appena discusse.

15.6 Riepilogo

- Una lista concatenata è una struttura di dati costituita da oggetti, chiamati nodi, che contengono sia dati sia un riferimento a un altro nodo. In questo modo, i nodi si collegano fra loro a formare una lista.
- È possibile realizzare una struttura dati concatenata (per esempio una lista concatenata) auto-contenuta definendo la classe nodo come *inner class* della lista concatenata.
- Una variabile o un oggetto, che permette di visitare uno alla volta tutti gli elementi di una collezione (un array o una lista concatenata), è chiamato iteratore.
- I nodi di una lista concatenata doppia hanno due collegamenti: uno al nodo precedente e uno a quello successivo. Ciò rende leggermente più semplici alcune operazioni, come l'aggiunta e la rimozione di elementi.
- Una pila è una struttura dati nella quale gli elementi vengono rimossi in ordine inverso rispetto a quello nel quale sono stati inseriti.
- Una coda è una struttura dati nella quale gli elementi vengono rimossi nello stesso ordine nel quale sono stati inseriti.
- Una tabella di *hash* è una struttura dati utilizzata per inserire e recuperare elementi in modo efficiente. Una funzione di *hash* viene utilizzata per associare a ogni elemento un valore utilizzato per indicizzarlo.
- Le liste concatenate possono essere utilizzate per implementare gli insiemi, realizzando anche le operazioni di unione, intersezione e verifica di appartenenza.
- Un albero binario è una struttura dati ramificata composta da nodi, ognuno dei quali ha due collegamenti ad altri nodi. Un albero ha un nodo speciale detto radice. Ogni nodo in un albero può essere raggiunto a partire dal nodo radice seguendo dei collegamenti.
- Se i valori in un albero binario sono immagazzinati seguendo la regola di immagazzinamento per la ricerca in alberi binari, esistono algoritmi molto efficienti per recuperare i valori nell'albero.

15.7 Esercizi

1. Si cambi la classe `ListaConcatenataDiStringhe` nel Listato 15.2 in modo che si possano aggiungere e rimuovere elementi alla fine della lista.

2. Si supponga di voler creare una struttura dati per contenere numeri che possono essere visitati nell'ordine con cui sono stati aggiunti o in ordine numerico crescente. Sono necessari nodi con due riferimenti. Se si segue una direzione tra i due riferimenti, si ottengono gli elementi nell'ordine con cui sono stati aggiunti. Se si segue l'altra direzione, si visitano gli elementi in ordine numerico. Si crei una classe `NodoDuale` che supporti tale struttura di dati. Non si scriva la struttura dati di per sé.

3. Si disegni una figura di una struttura dati inizialmente vuota, come descritta nel precedente esercizio, poi si aggiungano i numeri 1, 8, 4 e 6, proprio in quest'ordine.

4. Si scriva un programma che utilizzi un iteratore per duplicare ogni elemento in un'istanza di `ListaConcatenataDiStringheConIteratore` nel Listato 15.6. Per esempio, se la lista contiene "a", "b" e "c", dopo l'esecuzione del programma, dovrà contenere "a", "a", "b", "b", "c" e "c".

5. Si scriva un programma che utilizzi un iteratore per spostare alla fine della lista il primo elemento di un'istanza di `ListaConcatenataDiStringheConIteratore` (Listato 15.6). Per esempio, se la lista contiene "a", "b", "c" e "d", dopo l'esecuzione del codice, dovrà contenere "b", "c", "d" e "a".

6. Si scriva un programma che usi un iteratore per scambiare gli elementi in ogni coppia di elementi all'interno di un'istanza di `ListaConcatenataDiStringheConIteratore` nel Listato 15.6. Per esempio, se la lista contiene `"a"`, `"b"`, `"c"`, `"d"`, `"e"` e `"f"`, dopo l'esecuzione del codice, dovrà contenere `"b"`, `"a"`, `"d"`, `"c"`, `"f"` e `"e"`. Si può supporre che la lista contenga sempre un numero pari di stringhe.

7. Si ridefinisca la classe `ListaConcatenataDiStringheConIteratore` nel Listato 15.6 in modo che implementi l'interfaccia `Iterator` di Java. Questa interfaccia dichiara i metodi `next`, `remove` e `hasNext` come segue:

```
/**
Restituisce l'elemento successivo nella lista.
Solleva una NoSuchElementException
se non c'è nessun elemento da restituire.
*/
public E next() throws NoSuchElementException

/**
Rimuove l'ultimo elemento che è stato restituito
più di recente dall'invocazione di next().
Solleva una IllegalStateException
se il metodo next non è stato ancora invocato
o se il metodo remove è stato già invocato
dopo l'ultima invocazione del metodo next.
*/
public void remove() throws IllegalStateException
```

```
/**
Restituisce vero se c'è almeno
un elemento per il metodo next da restituire.
Altrimenti, restituisce falso.
*/
public boolean hasNext()
```

Si noti il tipo di dato generico `E` nella specifica del metodo `next`. Se si seguono attentamente le istruzioni di questo esercizio, non c'è bisogno di conoscere che cosa sia un'interfaccia, anche se ciò aumenterebbe senz'altro il livello di padronanza dello strumento. Le interfacce sono trattate nel Capitolo 11.

Si inizi la definizione della classe con:

```
import java.util.Iterator;
public class ListaConcatenataDiStringheConIteratore2
                    implements Iterator<String> {
    private NodoLista testa;
    private NodoLista corrente;
    private NodoLista precedente;                  //segue corrente
    private NodoLista dueDietro;                   //segue precedente
    private boolean rimuoviDaSuccessivo;           //vero se il metodo
    //rimuovi è stato invocato dall'ultima invocazione del metodo next.
    //Altrimenti è falso se next non è stato invocato affatto

    public ListaConcatenataDiStringheConIteratore2() {
        testa = null;
        corrente = null;
        precedente = null;
        dueDietro = null;
        rimuoviDaSuccessivo = true;
    }
```

Il resto della definizione è come quella contenuta nel Listato 15.6, tranne per l'aggiunta delle definizioni dei metodi specificate da `Iterator`. Si effettui una piccola modifica al metodo `reimpostaIterazione` in modo che la nuova variabile di istanza `dueDietro` venga reimpostata e si omettano i metodi `eliminaNodoCorrente`, `vaiAlSuccessivo` e `altriElementi`, che diventerebbero ridondanti.

Suggerimenti

- Malgrado i dettagli richiesti, questo esercizio è piuttosto facile. Le tre definizioni di metodo che bisogna aggiungere sono facili da implementare utilizzando i metodi a disposizione.
- Si noti che il metodo `hasNext` e il metodo `altriElementi` del Listato 15.6 non sono esattamente la stessa cosa.

Le classi di eccezioni menzionate sono tutte predefinite e non si devono ridefinire. Queste particolari eccezioni non necessitano di essere catturate o dichiarate in una clausola `throws`. L'interfaccia `Iterator` indica che il metodo `remove` solleva anche una `UnsupportedOperationException` se il metodo `remove` non è supportato. Tuttavia, il metodo `remove` di questo esercizio non ha bisogno di sollevare questa eccezione.

8. Si scriva un programma che crea due istanze di una classe generica `ListaConcatenata` fornita nel Listato 15.8. La prima istanza è `nomiDiStadi` e conterrà elementi di tipo `String`. La seconda istanza è `incassiPartita` e conterrà elementi di tipo `Double`. All'interno del ciclo, si leggano i dati per le partite di baseball giocate durante una stagione. I dati per una partita comprendono un nome di stadio e l'importo incassato per quella partita. Si aggiungano i dati a `nomiDiStadi` e `incassiPartita`. Poiché in uno stadio potrebbe essere giocata più di una partita, `nomiDiStadi` potrebbe prevedere anche la possibilità di elementi duplicati. Dopo la lettura di tutti i dati per le partite, si legga il nome di uno stadio e si mostri l'importo incassato per tutte le partite giocate in quello stadio.

15.8 Progetti

1. Si definisca una variazione su `ListaConcatenataDiStringheAutoContenuta` nel Listato 15.4 che memorizzi gli oggetti di tipo `Specie`, piuttosto che quelli di tipo `String`. Si scriva un programma che utilizzi tale classe per creare una lista concatenata di oggetti `Specie`; si chieda all'utente di inserire un nome di `Specie` e quindi si cerchi nella lista concatenata e si mostri uno dei seguenti messaggi, a seconda della presenza del nome nella lista:

   `La Specie` *Nome_Specie* `è una delle`
   Numero_Di_Nomi_Di_Specie_Sulla_Lista `presenti nella lista.`
   `I dati per` *Nome_Specie* `sono i seguenti:`
   Dati_per_Nome_Specie

   oppure

   `La Specie` *Nome_Specie* `non è una specie presente nella lista.`

   L'utente può inserire più nomi di specie finché non indica una fine per il programma. La classe `Specie` è fornita nel Listato 8.16 del Capitolo 8.

2. Si definisca una variazione di `ListaConcatenataDiStringheAutoContenuta` dal Listato 15.4 che memorizzi oggetti di tipo `Dipendente`, piuttosto che oggetti di tipo `String`. Si scriva un programma che usi questa classe per creare una lista concatenata di oggetti `Impiegato`, si chieda all'utente di inserire il numero di codice fiscale di un impiegato e poi ricercare la lista concatenata e mostrare i dati per il corrispondente impiegato. Se tale impiegato non esiste, si mostri un messaggio appropriato. La classe `Dipendente` è descritta nel Progetto 7 del Capitolo 13. Se non si è ancora svolto tale progetto, è necessario definire una classe `Dipendente` come appena descritto.

3. Si scriva una definizione di classe parametrica per una lista concatenata doppia che abbia un parametro per il tipo di dato memorizzato in un nodo. Si renda la classe nodo una *inner class*. La scelta di quali metodi definire fa parte di questo progetto. Inoltre, si scriva un programma per verificare interamente questa definizione di classe.

MyLab

Video 15.1 Definire una lista concatenata circolare

4. Si crei un'applicazione che terrà traccia di diversi gruppi di stringhe. Ogni stringa apparterrà esattamente a un gruppo. Si dovrà essere in grado di verificare se due stringhe sono nello stesso gruppo e di effettuare un'unione di due gruppi.

Si usi una struttura concatenata per rappresentare un gruppo di stringhe. Ogni nodo della struttura contiene una stringa e un riferimento ad altri nodi nel gruppo. Per esempio, il gruppo {"a", "b", "d", "e"} è rappresentato dalla seguente struttura:

Una stringa in ogni gruppo ("d" in questo esempio) è in un nodo che ha un riferimento null, cioè non fa riferimento a nessun altro nodo della struttura. Questa è la **stringa rappresentativa** del gruppo.

Si crei la classe ContenitoreGruppi per rappresentare tutti i gruppi e per eseguire le operazioni su di essi. La classe dovrebbe avere una variabile di istanza privata elementi per i nodi che appartengono a tutti i gruppi. I nodi di ogni gruppo sono collegati come descritto precedentemente. Si renda elementi un'istanza di ArrayList il cui tipo base è NodoGruppo, una *inner class* privata di ContenitoreGruppi. NodoGruppo ha le seguenti variabili di istanza private:

- dati – una stringa;
- collegamento – un riferimento a un altro nodo nel gruppo o null.

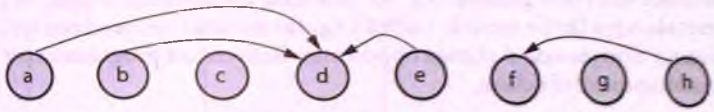
Definire i seguenti metodi nella classe ContenitoreGruppi.

- aggiungiElemento(s) – aggiunge una stringa s a un gruppo vuoto. Il metodo prima verifica che s non sia già presente in elementi (non esiste, cioè, un NodoGruppo che ha dati uguale a s). Se è già presente, il metodo non fa nulla, altrimenti crea un nuovo oggetto NodoGruppo che ha s come sua stringa e null come riferimento e aggiunge questo nuovo nodo a elementi. Il nuovo gruppo conterrà solo l'elemento s.
- getRappresentativa(s) – restituisce la stringa rappresentativa per il gruppo che contiene s. Per trovare la stringa rappresentativa, il metodo effettua una ricerca su elementi. Se non trova s, restituisce null. Se trova s, segue i riferimenti fino a trovare il riferimento null. La stringa contenuta in quel nodo è la stringa rappresentativa del gruppo.
- getTutteRappresentative – restituisce un'istanza di ArrayList che contiene le stringhe rappresentative di tutti i gruppi contenuti nell'istanza di ContenitoreGruppi.
- nelloStessoGruppo(s1, s2) – restituisce vero se la stringa rappresentativa del gruppo a cui appartiene s1 e la stringa rappresentativa del gruppo a cui appartiene s2 sono le stesse e diverse da null, in tal caso le stringhe s1 e s2 sono nello stesso gruppo.
- unione(s1, s2) – costituisce l'unione dei gruppi a cui appartengono s1 e s2. *Suggerimento.* Si trovino le stringhe rappresentative per s1 e s2. Se sono differenti e nessuna è null, si faccia in modo che il collegamento del nodo contenente la stringa rappresentativa di s1 si riferisca al nodo della stringa rappresentativa di s2.

Per esempio, si supponga di invocare `aggiungiElemento` con ognuna delle seguenti stringhe come argomento: `"a"`, `"b"`, `"c"`, `"d"`, `"e"`, `"f"`, `"g"` e `"h"`. Poi si formino i gruppi utilizzando queste operazioni di unione:

`unione("a", "d")`, `unione("b", "d")`, `unione("e", "b")`, `unione("h", "f")`.

Si avranno quattro gruppi, {`"a"`, `"b"`, `"d"`, `"e"`}, {`"c"`}, {`"f"`, `"h"`} e {`"g"`}, rappresentati dalla seguente struttura:



Le stringhe rappresentative per questi quattro gruppi sono, rispettivamente, `"d"`, `"c"`, `"f"` e `"g"`. Ora l'operazione `nelloStessoGruppo("a", "e")` potrebbe restituire vero poiché sia `getRappresentativa("a")` sia `getRappresentativa("e")` restituiscono `"d"`. Inoltre, `nelloStessoGruppo("a", "f")` restituirebbe falso, poiché `getRappresentativa("a")` restituisce `"d"` e `getRappresentativa("f")` restituisce `"f"`. L'operazione `unione("a", "f")` farebbe in modo che il nodo che contiene la stringa rappresentativa del gruppo al quale appartiene `"a"`, che è `"d"`, si riferisca al nodo che contiene la stringa rappresentativa del gruppo al quale appartiene `"f"`, che è `"f"`. Questo riferimento sarebbe rappresentato da una freccia da `"d"` a `"f"` nel diagramma precedente. L'applicazione realizzata dovrebbe creare un'istanza di `ContenitoreGrouppi` e consentire all'utente di aggiungere un numero arbitrario di stringhe, ognuna nel suo stesso gruppo; dovrebbe eseguire un numero arbitrario di operazioni `unione` per formare più gruppi; infine, dovrebbe implementare le altre operazioni.

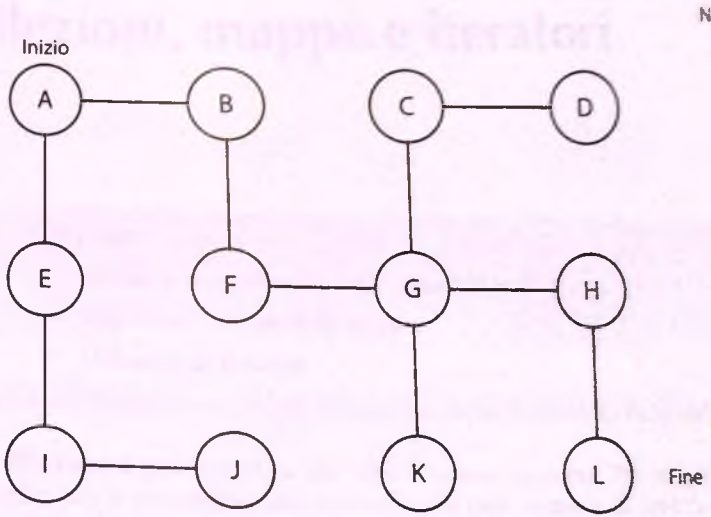
5. Si supponga di voler effettuare uno studio per contare il numero di uccelli di ogni specie in un'area. Si crei una classe `StudioUccelli` basata su una delle classi di liste concatenate presentate in questo capitolo (la lista concatenata che si utilizza influenzerà ciò che può fare la nuova classe, pertanto si pensi bene alla scelta da fare). Si modifichi la *inner class* per i nodi per aggiungere spazio a un contatore. `StudioUccelli` dovrebbe avere le seguenti operazioni.

   - `aggiungi(uccello)` – aggiunge la specie `uccello` alla fine della lista, se non già presente e imposta il suo contatore a 1; altrimenti, aggiunge 1 al contatore per `uccello`.
   - `getContatore(uccello)` – restituisce il contatore associato alla specie `uccello`. Se `uccello` non è nella lista restituisce 0.
   - `getReport` – mostra il nome e il contatore per ogni specie di uccello presente nella lista.

   Si scriva un programma che utilizza `StudioUccelli` per registrare i dati da un recente studio sugli uccelli. Si usi un ciclo per leggere i vari nomi di uccelli finché non si inserisce `fatto`. Al termine si mostri un report.

6. Anche se il tipo `long` può immagazzinare numeri interi grandi, non può gestire numeri estremamente grandi, come per esempio un numero con 200 cifre. Si crei una classe `NumeroEnorme` che utilizzi una lista concatenata di cifre per rappresentare numeri interi di lunghezza arbitraria. La classe dovrà avere un metodo per l'aggiunta di una nuova cifra che sarà la più significativa, così che si possano ottenere numeri sempre più grandi. Si aggiungano anche dei metodi per azzerare il numero e per restituire il valore del numero sotto forma di stringa, oltre ai costruttori e ai metodi *get* e *set* appropriati. Si scriva poi un programma per verificare il funzionamento della classe.
7. Si definisca una classe parametrica per una lista concatenata doppia. Si includano un metodo `equals`, un metodo `toString`, un metodo per produrre un iteratore e qualunque altro metodo si ritenga opportuno. Scrivere un programma per verificare il funzionamento della classe.
8. Si scriva un metodo `aggiungiInOrdine` per la lista concatenata parametrica del Listato 15.8 che aggiunga un nodo nel punto corretto in modo che la lista rimanga ordinata. Si noti che ciò richiede che il parametro `E` implementi l'interfaccia `Comparable`. Si scriva un programma per verificare il funzionamento del metodo.
9. Si aggiungano un metodo `rimuovi` e un iteratore alla classe `Insieme` del Listato 15.19. Si scriva un programma per verificarne il funzionamento.
10. La tabella di *hash* del Listato 15.17 contiene stringhe alle quali vengono associati numeri interi come valori di *hash*. Si modifichi il programma in modo che la tabella possa contenere oggetti della classe `Dipendente` definita nel Progetto 7 del Capitolo 13. Si utilizzi la variabile di istanza nome della classe `Dipendente` come argomento per la funzione di *hash*. La modifica richiederà il cambiamento della classe utilizzata come lista concatenata, dato che la lista utilizzata nel Listato 15.17 gestisce solamente stringhe. Per maggiore generalità, si modifichi la tabella di *hash* in modo che utilizzi la classe parametrica definita nel Listato 15.8. Occorrerà inoltre aggiungere un metodo *get* che restituisca l'oggetto `Dipendente` corrispondente a un nome specificato. Si verifichi il funzionamento della classe aggiungendo e estraendo vari nomi, considerando anche il caso in cui più nomi vengano associati alla stessa chiave.
11. Si modifichi la classe `Insieme<T>` del Listato 15.19 in modo che internamente utilizzi una tabella di *hash* al posto di una lista concatenata. Le intestazioni dei metodi pubblici dovranno rimanere invariate, così che un programma come quello del Listato 15.20 continui a funzionare senza richiedere modifiche. Si aggiunga inoltre un costruttore che consenta a chi utilizza la nuova classe di specificare la dimensione della tabella di *hash*. Successivamente, si implementi l'insieme utilizzando sia una tabella di *hash* che una lista concatenata, in modo che le operazioni di estrazione di un elemento sfruttino la tabella di *hash* e quelle che richiedono un'iterazione sugli elementi utilizzino la lista concatenata.

12. La struttura mostrata nella figura che segue è detta *grafo*. I cerchi sono detti *nodi*, mentre le linee sono dette archi. Ogni arco connette due nodi. Un grafo come questo può essere interpretato come un labirinto composto da stanze e passaggi tra una stanza e l'altra. I nodi rappresentano le stanze e gli archi i passaggi tra le stanze. Si noti che in questo esempio da ogni nodo partono al più quattro archi.



Si scriva un programma che implementi il labirinto di esempio utilizzando riferimenti a istanze di una classe `Nodo`. Ogni nodo del grafo corrisponderà a un'istanza di `Nodo`. Gli archi corrisponderanno a collegamenti tra un nodo e un altro e possono essere rappresentati nella classe `Nodo` da variabili di istanza che referenziano altri oggetti di tipo `Nodo`. All'inizio, l'utente parte dal nodo `A` e il suo scopo è raggiungere il nodo finale `L`. Il programma dovrà mostrare a ogni passo le possibili mosse in termini di direzione nord, sud, est o ovest. Un esempio di esecuzione è il seguente:

```
Ti trovi nella stanza A. Puoi andare a sud o a est.
E
Ti trovi nella stanza B. Puoi andare a sud o a ovest.
S
Ti trovi nella stanza F. Puoi andare a nord o a est.
E
```

Capitolo 16

Collezioni, mappe e iteratori

OBIETTIVI

- Fornire una panoramica delle collezioni Java.
- Descrivere l'utilizzo delle mappe.
- Utilizzare gli iteratori.

Una **collezione** è una struttura dati che contiene elementi. Per esempio, un oggetto `ArrayList<T>` è una collezione. Java offre un gran numero di interfacce e classi che coprono in modo esteso il tema delle collezioni. Un **iteratore** è un oggetto che scorre tutti gli elementi di una collezione. In questo capitolo saranno discusse collezioni, mappe e iteratori.

Prerequisiti

Per la comprensione degli argomenti trattati in questo capitolo, è necessaria la conoscenza del contenuto dei Capitoli da 1 a 6 e da 8 a 13. I Paragrafi 16.1.3 e 16.2.1 richiedono anche di aver compreso quanto presentato nel Paragrafo 15.3.

16.1 Le collezioni

Una collezione Java è una classe che contiene oggetti. Questo concetto è reso più preciso dall'interfaccia `Collection<T>`. Una collezione Java è una qualunque classe che implementi l'interfaccia `Collection<T>`. Come si vedrà, molte di queste classi possono essere utilizzate come strutture dati predefinite simili a quelle definite nel Capitolo 15. Un esempio di collezione Java visto nel Capitolo 12 è la classe `ArrayList<T>`. L'interfaccia `Collection<T>` consente di scrivere codice che può essere applicato a tutte le collezioni Java, così che non è necessario riscrivere il codice per ogni specifico tipo di collezione. Esistono anche altre interfacce e classi astratte che sono in un senso o nell'altro ottenute dall'interfaccia `Collection<T>`, alcune delle quali sono mostrate nella Figura 16.1. In questo paragrafo verrà fornita una panoramica sull'insieme delle collezioni Java. L'argomento è così vasto da non poter essere trattato in maniera esaustiva in questo testo. Di conseguenza, questo capitolo fornirà solo una trattazione intoduttiva all'argomento.

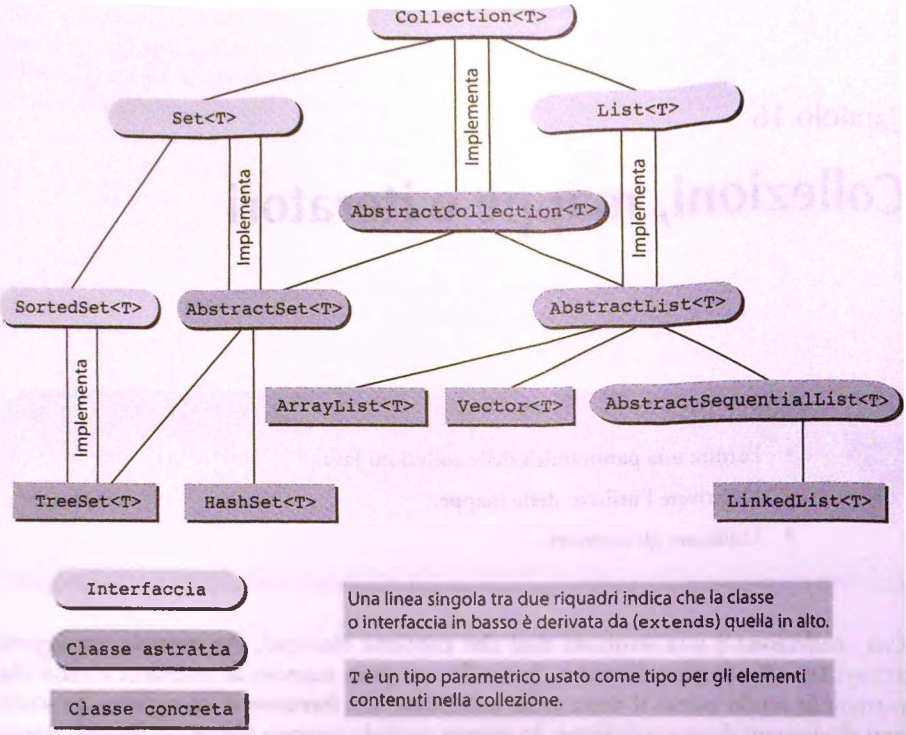


Figura 16.1 Panoramica della libreria delle collezioni.

Le collezioni sono utilizzate con gli *iteratori*, discussi nel Paragrafo 16.3. Collezioni e iteratori sono trattati in due paragrafi diversi per maggior chiarezza, ma si tratta in realtà di due concetti strettamente legati e in pratica vengono di solito utilizzati insieme.

Prima di discutere l'interfaccia `Collection<T>`, è necessario un breve approfondimento sul modo di specificare il tipo dei parametri.

16.1.1 Wildcard

Le classi e le interfacce della libreria delle collezioni Java utilizzano una modalità di specifica del tipo di parametro che non è stata ancora incontrata in precedenza. Per esempio, tale modalità consente di specificare cose come "L'argomento deve essere un `ArrayList<T>`, ma può avere qualunque tipo base". Più in generale, queste nuove modalità di specifica del tipo di parametri usano i tipi generici ma senza specificare completamente il tipo di oggetto da sostituire al tipo di parametro. Poiché queste nuove modalità di specifica coprono un'ampia gamma di tipi di argomenti, sono note come **wildcard.**

Il wildcard più semplice da capire è `<?>`, che indica che si può usare qualunque tipo al posto del tipo di parametro. Per esempio,

```
public void metodoDiEsempio(String arg1, ArrayList<?> arg2)
```

viene invocato passando due argomenti: il primo deve essere di tipo `String`, mentre il secondo può essere un `ArrayList<T>` con qualunque tipo base.

Si noti che `ArrayList<?>` è diverso da `ArrayList<Object>`. Per esempio, se la specifica di tipo è `ArrayList<?>`, allora si può passare un argomento di tipo `ArrayList<String>` (così come anche di altri tipi); al contrario, non si può passare un argomento di tipo `ArrayList<String>` se la specifica di tipo è `ArrayList<Object>`.

È possibile limitare un *wildcard* specificando che il tipo da usare al suo posto deve essere un antenato o un discendente di qualche classe o interfaccia. Per esempio, `<? extends Animale>` indica che l'argomento può essere un'istanza di una qualunque classe derivata dalla classe `Animale`. Per esempio,

```
public void altroMetodo(String arg1, ArrayList<? extends Animale> arg2)
```

viene invocato passando due argomenti: il primo deve essere di tipo `String`, mentre il secondo può essere di qualunque tipo `ArrayList<T>` a patto che il tipo base di `ArrayList<T>` sia derivato dalla classe `Animale`.

Per specificare che il tipo da sostituire al *wildcard* sia un antenato di qualche classe o interfaccia, si usa `super` al posto di `extends`. Per esempio, `ArrayList<? super Animale>` indica un `ArrayList<T>` il cui tipo base può essere un qualunque antenato della classe `Animale`.

16.1.2 La libreria delle collezioni (Collection Framework)

`Collection<T>` costituisce l'interfaccia più generica della libreria Java che contiene le classi per le collezioni. Questa interfaccia descrive le operazioni di base che devono essere implementate da tutte le classi di tipo collezione. Un riassunto di tali operazioni (le intestazioni dei metodi) è riportato nella Figura 16.2. Poiché un'interfaccia è un tipo, si possono definire metodi con parametri di tipo `Collection<T>`. Il parametro può essere sostituito con un argomento che sia un'istanza di una qualunque classe di tipo collezione (cioè qualunque classe che implementi l'interfaccia `Collection<T>`). Ciò costituisce uno strumento molto potente, come si vedrà. È già stata descritta una classe che implementa l'interfaccia `Collection<T>`: la classe `ArrayList<T>`. In aggiunta ai metodi presentati nel Capitolo 12 per la classe `ArrayList<T>`, questa classe implementa anche tutti i metodi mostrati nella Figura 16.2. Ci sono molte classi predefinite che implementano l'interfaccia `Collection<T>` ed è possibile definirne di nuove. Un metodo scritto per lavorare su un parametro di tipo `Collection<T>` funzionerà anche con tutte queste classi. Inoltre, i metodi dell'interfaccia `Collection<T>` garantiscono la possibilità di utilizzare contemporaneamente diverse classi di tipo collezione. Per esempio, si consideri il metodo

```
public boolean containsAll(Collection<?> collezioneDiObiettivi)
```

Si può utilizzare questo metodo con due oggetti `ArrayList<T>` (uno come oggetto chiamante il metodo e uno come argomento) per verificare se il primo contiene tutti gli elementi del secondo. I due oggetti `ArrayList<T>` non devono nemmeno essere dello stesso tipo base. Inoltre, si può utilizzare il metodo con un oggetto di tipo `ArrayList<T>` e un'istanza di qualunque altra classe che implementi `Collection<T>` per confrontare gli elementi dei due diversi tipi di oggetti `Collection<T>`.

L'interfaccia `Collection<T>` è compresa nel package `java.util`.

COSTRUTTORI

Nonostante non sia espressamente richiesto dall'interfaccia, qualunque classe che implementi l'interfaccia `Collection<T>` dovrebbe avere almeno due costruttori: un costruttore senza argomenti che crea un oggetto di tipo `Collection<T>` vuoto e un costruttore a un parametro di tipo `Collection<? extends T>` che crea un oggetto di tipo `Collection<T>` contenente gli stessi elementi dell'argomento.

METODI

`boolean isEmpty()`

Restituisce `true` se l'oggetto chiamante è vuoto, altrimenti restituisce `false`.

`public boolean contains(Object obiettivo)`

Restituisce `true` se l'oggetto chiamante contiene almeno un'istanza di `obiettivo`. Utilizza `obiettivo.equals` per determinare se `obiettivo` è contenuto nell'oggetto chiamante.

`public boolean containsAll(Collection<?> collezioneDiObiettivi)`

Restituisce `true` se l'oggetto chiamante contiene tutti gli elementi in `collezioneDiObiettivi`. Per ogni `elemento` in `collezioneDiObiettivi`, il metodo usa `elemento.equals` per determinare se `elemento` è contenuto nell'oggetto chiamante.

`public boolean equals(Object altro)`

Questo è il metodo `equals` per la collezione, non per gli elementi in essa contenuti. Sovrascrive il metodo `equals` ereditato. Nonostante non ci siano vincoli espliciti sul metodo `equals` per una collezione, dovrebbe essere definito come descritto nel Capitolo 10 ed essere inoltre compatibile con la nozione intuitiva di uguaglianza tra due collezioni.

`public int size()`

Restituisce il numero di elementi contenuti nell'oggetto chiamante. Se quest'ultimo contiene più di `Integer.MAX_VALUE` elementi, restituisce `Integer.MAX_VALUE`.

`Iterator<T> iterator()`

Restituisce un iteratore per l'oggetto chiamante (gli iteratori per le collezioni sono trattati nel Paragrafo 16.3).

`public Object[] toArray()`

Restituisce un array contenente tutti gli elementi dell'oggetto chiamante. Se l'oggetto chiamante garantisce l'ordinamento degli elementi restituiti dal suo iteratore, questo metodo deve restituire gli elementi nello stesso ordine.

L'array restituito dovrebbe essere un nuovo array, in modo che l'oggetto chiamante non abbia riferimenti all'array restituito (si potrebbe anche richiedere che gli elementi dell'array siano cloni degli elementi della collezione. Tuttavia, apparentemente ciò non è richiesto dall'interfaccia, dal momento che le classi della libreria, come `Vector<T>`, restituiscono array che contengono riferimenti agli elementi della collezione).

`public <E> E[] toArray(E[] a)`

Si noti che il tipo di parametro `E` è diverso da `T`. Quindi `E` può essere qualunque tipo, non è necessario che coincida con il tipo `T` in `Collection<T>`. Per esempio, `E` potrebbe essere un antenato di `T`.

Restituisce un array contenente tutti gli elementi dell'oggetto chiamante. L'argomento `a` è utilizzato essenzialmente per specificare il tipo di array da restituire. In dettaglio: il tipo di array restituito è lo stesso di `a`. Se `a` può contenere tutti gli elementi dell'oggetto chiamante, `a` è utilizzato per contenere gli elementi dell'array restituito; altrimenti viene creato un nuovo array con lo stesso tipo di `a`.

Se `a` ha più elementi dell'oggetto chiamante, gli elementi in eccesso alla fine di `a` vengono impostati a `null`.

Se l'oggetto chiamante garantisce l'ordine degli elementi ritornati dal suo iteratore, il metodo deve restituire gli elementi nello stesso ordine (gli iteratori per le collezioni sono trattati nel Paragrafo 16.3).

Figura 16.2 Intestazione dei metodi dell'interfaccia `Collection<T>`. (*segue*)

`public int hashCode()` Restituisce il codice *hash* dell'oggetto chiamante.
METODI OPZIONALI I seguenti metodi sono opzionali, il che significa che devono comunque essere implementati, ma la loro implementazione può limitarsi a generare una `UnsupportedOperationException` se, per qualche motivo, non si è interessati a definire un'implementazione vera e propria. Una `UnsupportedOperationException` è una `RuntimeException` e quindi non è necessario gestirla o dichiararla in una clausola `throws`.
`public boolean add(T elemento)` (opzionale) Garantisce che l'oggetto chiamante contenga l'`elemento` specificato. Restituisce `true` se l'oggetto chiamante è stato modificato dalla chiamata. Restituisce `false` se l'oggetto chiamante non ammette elementi duplicati e contiene già `elemento`; inoltre, restituisce `false` anche se l'oggetto chiamante non viene modificato per qualunque altro motivo.
`public boolean addAll(Collection<? extends T> collezioneDaAggiungere)` (opzionale) Garantisce che l'oggetto chiamante contenga tutti gli elementi in `collezioneDaAggiungere`. Restituisce `true` se l'oggetto chiamante è stato modificato dalla chiamata, altrimenti restituisce `false`.
`public boolean remove(Object elemento)` (opzionale) Rimuove una singola istanza dell'`elemento` specificato dall'oggetto chiamante. Restituisce `true` se l'oggetto chiamante conteneva l'`elemento`, altrimenti restituisce `false`.
`public boolean removeAll(Collection<?> collezioneDaRimuovere)` (opzionale) Rimuove dall'oggetto chiamante tutti gli elementi che sono contenuti anche in `collezioneDaRimuovere`. Restituisce `true` se l'oggetto chiamante è stato modificato, altrimenti restituisce `false`.
`public void clear()` (opzionale) Rimuove tutti gli elementi dall'oggetto chiamante.
`public boolean retainAll(Collection<?> conservaElementi)` (opzionale) Mantiene nell'oggetto chiamante tutti gli elementi contenuti anche nella collezione `conservaElementi`. In altre parole, rimuove dall'oggetto chiamante tutti gli oggetti non contenuti nella collezione `conservaElementi`. Restituisce `true` se l'oggetto chiamante è stato modificato dalla chiamata, altrimenti restituisce `false`.

Figura 16.2 Intestazione dei metodi dell'interfaccia `Collection<T>`.

Package

Tutte le classi e le interfacce di tipo collezione discusse in questo capitolo fanno parte del package `java.util`.

Le relazioni tra alcune delle classi e delle interfacce che implementano o estendono l'interfaccia `Collection<T>` sono riportate nella Figura 16.1. Le principali interfacce che estendono `Collection<T>` sono due: `Set<T>` e `List<T>`. Le classi che implementano `Set<T>` non ammettono elementi ripetuti. Le classi che implementano `List<T>` ordinano i loro elementi in una lista, così che esista un elemento di indice 0 (zero), uno di indi-

ce 1, uno di indice 2 e così via. Una classe che implementa `List<T>` ammette elementi ripetuti. La classe `ArrayList<T>` implementa l'interfaccia `List<T>`.

L'interfaccia `Set<T>` ha le stesse intestazioni dei metodi dell'interfaccia `Collection<T>`, ma in alcuni casi la semantica (cioè il significato) è diversa. Per esempio, la semantica dell'aggiunta di nuovi elementi all'insieme non consente i duplicati. I metodi per l'aggiunta di elementi sono descritti nella Figura 16.3.

L'interfaccia `Set<T>` fa parte del package `java.util`. L'interfaccia `Set<T>` estende l'interfaccia `Collection<T>` e ha le stesse dichiarazioni di metodi presentate nella Figura 16.2. Tuttavia, la semantica dei metodi di aggiunta cambia come descritto di seguito.
`public boolean add(T elemento)` (opzionale) Se elemento non è già contenuto nell'oggetto chiamante, è aggiunto e viene restituito `true`. Altrimenti l'oggetto chiamante rimane invariato e viene restituito `false`.
`public boolean addAll(Collection<? extends T> collezioneDaAggiungere)` (opzionale) Garantisce che, dopo la chiamata, l'oggetto chiamante contenga tutti gli elementi in `collezioneDaAggiungere`. Restituisce `true` se l'oggetto chiamante viene modificato dalla chiamata, altrimenti restituisce `false`. Quindi se `collezioneDaAggiungere` è un `Set<T>` l'oggetto chiamante viene trasformato nella sua unione con `collezioneDaAggiungere`.

Figura 16.3 Aggiunta di elementi nell'interfaccia `Set<T>`.

L'interfaccia `List<T>` ha più intestazioni di metodi dell'interfaccia `Collection<T>` e alcuni dei metodi ereditati da quest'ultima hanno una semantica diversa. Per esempio, la semantica dell'aggiunta di nuovi elementi all'insieme ammette i duplicati e devono essere stabilite delle regole per determinare quale elemento debba essere rimosso, qualora esistano duplicati. Questi metodi e quelli di nuova definizione sono presentati nella Figura 16.4.

L'interfaccia `List<T>` fa parte del package `java.util`. L'interfaccia `List<T>` estende l'interfaccia `Collection<T>`.
AGGIUNTA E RIMOZIONE DI ELEMENTI
`public boolean add(T elemento)` (opzionale) Aggiunge `elemento` alla fine della lista di elementi dell'oggetto chiamante. Normalmente restituisce `true`. Restituisce `false` se l'operazione è fallita, ma in tal caso deve essersi verificato qualcosa di grave e si otterrà probabilmente anche un errore durante l'esecuzione.
`public boolean addAll(Collection<? extends T> collezioneDaAggiungere)` (opzionale) Aggiunge alla fine della lista di elementi dell'oggetto chiamante tutti gli elementi in `collezioneDaAggiungere`. Gli elementi vengono aggiunti nell'ordine nel quale vengono forniti da un iteratore per `collezioneDaAggiungere`.
`public boolean remove(Object elemento)` (opzionale) Elimina dalla lista di elementi dell'oggetto chiamante la prima occorrenza di `elemento`, se presente. Restituisce `true` se l'oggetto chiamante conteneva l'`elemento`, altrimenti restituisce `false`.

Figura 16.4 Selezione di metodi dell'interfaccia `List<T>`. (*segue*)

`public boolean removeAll(Collection<?> collezioneDaRimuovere)` (opzionale)
Rimuove dall'oggetto chiamante tutti gli elementi contenuti anche in `collezioneDaRimuovere`. Restituisce `true` se l'oggetto chiamante è stato modificato, altrimenti restituisce `false`.

NUOVE INTESTAZIONI DI METODI

I metodi seguenti appartengono all'interfaccia `List<T>`, ma non all'interfaccia `Collection<T>`. Quelli opzionali sono indicati.

`public void add(int indice, T nuovoElemento)` (opzionale)
Inserisce `nuovoElemento` alla posizione `indice` nella lista di elementi dell'oggetto chiamante. L'elemento che si trovava alla posizione `indice` e tutti i successivi vengono spostati di una posizione.

`public boolean addAll(int indice, Collection<? extends T> collezioneDaAggiungere)` (opzionale)
Inserisce tutti gli elementi di `collezioneDaAggiungere` nella lista di elementi dell'oggetto chiamante a partire dalla posizione `indice`. L'elemento originariamente alla posizione `indice` e i successivi vengono spostati più avanti. Gli elementi sono aggiunti nello stesso ordine in cui sono forniti da un iteratore di `collezioneDaAggiungere`.

`public T get(int indice)`
Restituisce l'oggetto alla posizione `indice`.

`public T set(int indice, T nuovoElemento)` (opzionale)
Imposta l'elemento alla posizione `indice` a `nuovoElemento`. Viene restituito l'elemento che si trovava originariamente in quella posizione.

`public T remove(int indice)` (opzionale)
Rimuove l'elemento alla posizione `indice` dell'oggetto chiamante. Sposta gli elementi successivi a sinistra di una posizione (sottrae 1 ai loro indici). Restituisce l'elemento rimosso.

`public int indexOf(Object obiettivo)`
Restituisce l'indice del primo elemento uguale a `obiettivo`. Utilizza il metodo `equals` dell'oggetto `obiettivo` per verificare l'uguaglianza. Restituisce `-1` se `obiettivo` non viene trovato.

`public int lastIndexOf(Object obiettivo)`
Restituisce l'indice dell'ultimo elemento uguale a `obiettivo`. Utilizza il metodo `equals` dell'oggetto `obiettivo` per verificare l'uguaglianza. Restituisce `-1` se `obiettivo` non viene trovato.

`public List<T> subList(int daIndice, int aIndice)`
Restituisce una *vista* degli elementi alle posizioni comprese tra `daIndice` a `aIndice` dell'oggetto chiamante; l'oggetto alla posizione `daIndice` è incluso, quello alla posizione `aIndice` (se presente) è escluso. La *vista* è composta di riferimenti all'oggetto chiamante; le modifiche alla *vista* quindi modificano potenzialmente l'oggetto chiamante. L'oggetto restituito è di tipo `List<T>`, ma non è necessario che sia dello stesso tipo dell'oggetto chiamante. Se `daIndice` coincide con `aIndice`, viene restituito un oggetto `List<T>` vuoto.

`ListIterator<T> listIterator()`
Restituisce un iteratore per l'oggetto chiamante (gli iteratori per le collezioni sono trattati nel Paragrafo 16.3).

`ListIterator<T> listIterator(int indice)`
Restituisce un iteratore per l'oggetto chiamante che parte da `indice`. Il primo elemento restituito dall'iteratore è quello alla posizione `indice` (gli iteratori per le collezioni sono trattati nel Paragrafo 16.3).

Figura 16.4 Selezione di metodi dell'interfaccia `List<T>`.

Interfacce di tipo collezione

Le interfacce principali per le classi di tipo collezione sono `Collection<T>`, `Set<T>` e `List<T>`. Sia `Set<T>` che `List<T>` sono derivate da `Collection<T>`. L'interfaccia `Set<T>` è progettata per collezioni che non ammettono elementi ripetuti e non impongono un ordine ai propri elementi. L'interfaccia `List<T>` è progettata per collezioni che ammettono elementi ripetuti e impongono un ordine ai propri elementi.

Cicli *for-each*

I cicli *for-each* possono essere utilizzati con tutte le collezioni discusse in questo capitolo.

Metodi opzionali

Qual è l'utilità della dichiarazione di un metodo opzionale in un'interfaccia? L'utilità di un'interfaccia è quella di specificare quali metodi possano essere utilizzati su un oggetto del tipo dell'interfaccia, in modo da poter scrivere codice valido per oggetti arbitrari di quel tipo. L'idea alla base dei metodi opzionali è che normalmente questi vengano implementati, anche se in situazioni particolari il programmatore potrebbe lasciarli come "non supportati" (l'alternativa sarebbe avere due interfacce, una con i metodi opzionali e una senza. Diversamente dal solito, i progettisti del linguaggio Java hanno optato per un numero ridotto di interfacce). Ma questo non è tutto.

I metodi opzionali non sono a rigore veramente opzionali. Come ogni altro metodo in un'interfaccia, devono avere un corpo così che la loro dichiarazione possa essere convertita in una definizione completa. Cosa c'è quindi di opzionale? Il termine "opzionale" è riferito alla semantica del metodo: se il metodo è opzionale, se ne può dare un'implementazione banale senza sottrarsi alla responsabilità di seguire la (più libera) semantica per l'interfaccia.

Per evitare che i metodi opzionali producano comportamenti imprevedibili, la semantica dell'interfaccia prescrive che se non si fornisce un'implementazione "vera" di un metodo opzionale, bisognerebbe assicurarsi che il metodo generi una `UnsupportedOperationException`. Per esempio, il metodo `add` dell'interfaccia `Collection<T>` è opzionale e quindi può essere implementato come segue (sempre che ci sia una buona ragione per farlo):

```
public boolean add(T elemento) {
    throw new UnsupportedOperationException();
}
```

La classe `UnsupportedOperationException` è derivata dalla classe `RuntimeException`, quindi un'eccezione di tipo `UnsupportedOperationException` è un'eccezione non controllata, il che significa che non è necessario gestirla in un blocco `catch` o dichiararla con una clausola `throws`.

L'idea è quella di scrivere e utilizzare il codice che implementa un'interfaccia con metodi opzionali in modo che le `UnsupportedOperationException` vengano generate solo nella fase di debug. Queste regole sui metodi opzionali fanno parte della semantica dell'interfaccia e come tutti gli altri aspetti di essa dipendono interamente dalla buona volontà e dalla responsabilità del programmatore che definisce la classe che implementa l'interfaccia.

Metodi opzionali

Quando un'interfaccia indica un metodo come "opzionale", il metodo deve comunque essere implementato quando si definisce una classe che implementa l'interfaccia. Tuttavia, se non si vuole fornire un'implementazione "vera" del metodo, è sufficiente che quest'ultimo generi una `UnsupportedOperationException`.

Lavorare con tutte le eccezioni

Molti dei metodi delle interfacce `Collection<T>`, `Set<T>` e `List<T>` prevedono la generazione di eccezioni. Tutte le eccezioni sono, però, non controllate, per cui non è necessario gestirle in un blocco `catch` o dichiararle con clausole `throws`. Queste eccezioni sono intese prevalentemente come strumento di debug. Se si utilizza una classe collezione già esistente, possono essere interpretate come messaggi d'errore durante l'esecuzione. Se invece si sta definendo una nuova classe derivando un'altra classe di tipo collezione, la maggior parte della generazione delle eccezioni sarà ereditata e non ci sarà bisogno di preoccuparsene troppo. Se però si definisce una classe collezione da zero e si vuole che implementi una delle interfacce di tipo collezione, sarà necessario generare le eccezioni previste dall'interfaccia.

A parte un unico caso, tutte le eccezioni nominate in questo capitolo fanno parte del package `java.lang` e quindi non necessitano di un'istruzione di `import`. L'unico caso a parte è costituito dalla `NoSuchElementException`, utilizzata in relazione agli iteratori nel Paragrafo 16.3. Questa eccezione si trova nel package `java.util`, quindi è necessario importare questo package se il codice utilizza la classe `NoSuchElementException`.

16.1.3 Classi concrete di tipo collezione

Le classi astratte `AbstractSet<T>` e `AbstractList<T>` sono fornite per semplificare l'implementazione delle interfacce, rispettivamente, `Set<T>` e `List<T>` e contengono quasi esclusivamente i metodi richiesti dalle interfacce che implementano. Nonostante siano pochi i metodi astratti contenuti in queste due classi, gli altri metodi (quelli non astratti) hanno implementazioni pressoché inutili, che devono essere ridefinite. Quando si definisce una classe derivata da `AbstractSet<T>` o `AbstractList<T>`, è quindi necessario implementare non solo i metodi astratti, ma anche tutti gli altri metodi che si vogliono utilizzare. Di solito ha più senso utilizzare (o definire classi derivate da)

classi `HashSet<T>`, `ArrayList<T>` o `Vector<T>`, che sono a loro volta derivate da `AbstractSet<T>` e `AbstractList<T>` e costituiscono implementazioni complete delle interfacce `Set<T>` e `List<T>`.

La classe astratta `AbstractCollection<T>` è uno scheletro di classe che implementa l'interfaccia `Collection<T>`. Nonostante sia del tutto possibile farlo, sarà raramente necessario (se mai lo sarà) definire una classe derivata da `AbstractCollection<T>`. Piuttosto, normalmente si definiscono classi derivate da una delle classi discendenti di `AbstractCollection<T>`.

Se si vuole una classe che implementi l'interfaccia `Set<T>` e non sono necessari altri metodi oltre quelli di tale interfaccia, si può utilizzare la classe concreta `HashSet<T>`. Quindi, nonostante tutto quello che è stato detto finora, se tutto ciò di cui si ha bisogno è una classe di tipo collezione che non ammetta elementi ripetuti, si può utilizzare la classe `HashSet<T>` senza doversi preoccupare di tutte le altre classi e interfacce mostrate nella Figura 16.1. La parola *"hash"* si riferisce al fatto che la classe `HashSet<T>` è implementata utilizzando una **tabella *hash***. Una tabella *hash* (o *mappa hash*) è una struttura dati che contiene oggetti. Un oggetto è memorizzato in una tabella *hash* associandogli una **chiave** (*key*) univoca. La classe `HashSet<T>` implementa naturalmente tutti i metodi dell'interfaccia `Set<T>` e non aggiunge altri metodi, a parte i costruttori. Un riassunto dei costruttori e di altri metodi della classe `HashSet<T>` è presentato nella Figura 16.5. Se si vuole definire una propria classe che implementi l'interfaccia `Set<T>`, converrà probabilmente utilizzare come classe base la `HashSet<T>` anziché la `AbstractSet<T>`.

La classe `HashSet<T>` fa parte del package `java.util`. La classe `HashSet<T>` estende la classe `AbstractSet<T>` e implementa l'interfaccia `Set<T>`. La classe `HashSet<T>` implementa tutti i metodi dell'interfaccia `Set<T>` (Figura 16.3). Gli unici altri metodi nella classe `HashSet<T>` sono i costruttori, tre dei quali riportati di seguito. Tutte le eccezioni citate sono del tipo non controllato, quindi non è necessario gestirle in un blocco `catch` o dichiararle in una clausola `throws`. Tutte le eccezioni menzionate fanno parte del package `java.lang` e quindi non richiedono l'importazione di alcun package.
`public HashSet()` Crea un nuovo `HashSet` vuoto
`public HashSet(Collection<? extends T> c)` Crea un nuovo insieme contenente tutti gli elementi di `c`. Se `c` è `null`, genera un'eccezione `NullPointerException`.
`public HashSet(int dimensioneIniziale)` Crea un nuovo insieme vuoto della capacità specificata. Se `dimensioneIniziale` è minore di 0, genera un'eccezione `IllegalArgumentException`.
I metodi sono gli stessi già descritti per l'interfaccia `Set<T>` (Figura 16.3).

Figura 16.5 Metodi della classe `HashSet<T>`.

È importante tenere presente che se si utilizza la classe `HashSet<T>` con una classe propria al posto del parametro `T`, la classe deve ridefinire i due metodi seguenti:

```
public int hashCode();
public boolean equals(Object obj);
```

Il metodo `hashCode()` deve restituire una chiave numerica che idealmente dovrebbe rappresentare un identificativo univoco per un oggetto della classe. Si veda a proposito il Paragrafo 15.3 che tratta i codici *hash*. Ridefinire il metodo `equals()` per ogni nuova classe è sempre una buona idea, ma in questo caso è indispensabile. Java, infatti, utilizzerà il codice *hash* per indicizzare un oggetto e successivamente il metodo `equals()` per controllare se esiste già nell'insieme un oggetto uguale. Anche se due oggetti diversi avessero lo stesso codice *hash*, saranno comunque indicizzati correttamente se il metodo `equals()` indica che sono diversi. In ogni caso, il fatto di avere codici *hash* coincidenti provocherà una cosiddetta collisione che ridurrà le prestazioni.

Il Listato 16.1 mostra un semplice programma che utilizza la classe `HashSet<T>`. Le operazioni di unione e intersezione possono essere realizzate utilizzando rispettivamente i metodi `addAll` e `retainAll`. Per stampare gli elementi in un oggetto `HashSet<T>` viene definito il metodo `stampaInsieme`. Questo metodo utilizza gli iteratori, che saranno trattati solo nel Paragrafo 16.3, quindi per ora si possono trascurare i dettagli del funzionamento di `stampaInsieme`.

In generale, è sempre preferibile utilizzare le classi predefinite di tipo collezione, salvo che queste non forniscano tutte le funzionalità necessarie. Per esempio, si supponga di volere che ogni elemento aggiunto a un insieme contenga un riferimento all'insieme stesso. Ciò potrebbe essere utile per esempio per determinare se due elementi appartengono allo stesso insieme: basterebbe confrontare i riferimenti agli insiemi che contengono i due elementi. Senza questi riferimenti, sarebbe necessario invocare il metodo `contains` di ogni insieme, per verificare se i due elementi appartengono allo stesso insieme. Se questa fosse una caratteristica fondamentale del programma, si potrebbe voler sviluppare una classe personalizzata, invece di utilizzare le classi collezione predefinite. Se le classi predefinite fossero sufficienti, si otterrebbe un codice più breve, che è di solito più semplice da sviluppare e gestire. Inoltre, le classi collezione come `HashSet<T>` sono state progettate con particolare attenzione all'efficienza e alla scalabilità.

LISTATO 16.1 Esempio d'uso della classe `HashSet<T>`.

MyLab

```
import java.util.HashSet;
import java.util.Iterator;

public class HashSetDemo {
    public static void stampaInsieme(HashSet<String> insieme) {
        Iterator<String> i = insieme.iterator();
        while (i.hasNext())
            System.out.print(i.next() + " ");
        System.out.println();
    }

    public static void main(String[] args) {
        HashSet<String> rotondi = new HashSet<String>();
        HashSet<String> verdi = new HashSet<String>();

        // Aggiunta di dati ai due insiemi
        rotondi.add("piselli");
        rotondi.add("palla");
        rotondi.add("torta");
        rotondi.add("acini");
```

Il metodo `stampaInsieme` usa un iteratore per stampare il contenuto di un oggetto `HashSet<T>`. Gli iteratori sono descritti nel Paragrafo 16.3.

```
        verdi.add("piselli");
        verdi.add("acini");
        verdi.add("tubo da giardino");
        verdi.add("erba");

        System.out.println("Contenuto dell'insieme rotondi: ");
        stampaInsieme(rotondi);
        System.out.println("\nContenuto dell'insieme verdi: ");
        stampaInsieme(verdi);

        System.out.println("\npalla e' nell'insieme rotondi? " +
        rotondi.contains("palla"));
        System.out.println("palla e' nell'insieme verdi? " +

        verdi.contains("palla"));

        System.out.println("\npalla e piselli nello stesso insieme? "
           + ((rotondi.contains("palla") &&
              (rotondi.contains("piselli"))) ||
              (verdi.contains("palla") &&
              (verdi.contains("piselli")))));
        System.out.println("torta e erba nello stesso insieme? " +
             ((rotondi.contains("torta") &&
             (rotondi.contains("erba"))) ||
             (verdi.contains("torta") &&
             (verdi.contains("erba")))));

        // Per ottenere l'unione dei due insiemi si usa
        // il metodo addAll
        HashSet<String> unione = new HashSet<String>(rotondi);
        unione.addAll(verdi);
        System.out.println("\nUnione di verdi e rotondi:");
        stampaInsieme(unione);

        // Per ottenere l'intersezione dei due insiemi si usa
        // il metodo retainAll
        HashSet<String> intersezione = new HashSet<String>(rotondi);
        intersezione.retainAll(verdi);
        System.out.println("\nIntersezione di verdi e rotondi:");
        stampaInsieme(intersezione);
        System.out.println();
    }
}
```

Esempio di output

```
Contenuto dell'insieme rotondi:
palla piselli acini torta
```

```
Contenuto dell'insieme verdi:
erba piselli acini tubo da giardino

palla e' nell'insieme rotondi? true
palla e' nell'insieme verdi? false

palla e piselli nello stesso insieme? true
torta e erba nello stesso insieme? false

Unione di verdi e rotondi:
palla erba piselli acini tubo da giardino torta

Intersezione di verdi e rotondi:
piselli acini
```

Se serve una classe che implementi l'interfaccia `List<T>` e non occorrono metodi aggiuntivi, si possono utilizzare le classi `ArrayList<T>` o `Vector<T>`. Quindi, riassumendo, se tutto ciò di cui si ha bisogno è una collezione che permetta elementi ripetuti o che ordini gli elementi come in una lista (cioè come in un array) o che abbia entrambe le proprietà, basta usare la classe `ArrayList<T>` o la classe `Vector<T>` senza preoccuparsi di tutte le altre classi e interfacce della Figura 16.2. Le classi `ArrayList<T>` e `Vector<T>` implementano tutti i metodi dell'interfaccia `List<T>`. I metodi della classe `ArrayList<T>` sono stati presentati nel Capitolo 12 e sono riportati in forma più completa nella Figura 16.6, che contiene anche i metodi della classe `Vector<T>`. Se si vuole definire una nuova classe che implementi l'interfaccia `List<T>`, è solitamente preferibile usare la classe `ArrayList<T>` o `Vector<T>` come classe base al posto della classe `AbstractList<T>`.

La classe astratta `AbstractSequentialList<T>` è derivata dalla classe `AbstractList<T>`. Nonostante ridefinisca alcuni dei metodi ereditati dalla classe `AbstractList<T>`, non ne aggiunge di completamente nuovi. Lo scopo della classe `AbstractSequentialList<T>` è quello di fornire un'implementazione efficiente degli spostamenti sequenziali lungo la lista al costo di una poco efficiente implementazione dell'accesso diretto agli elementi (cioè un'implementazione poco efficiente del metodo `get`). La classe `LinkedList<T>` è una classe concreta derivata dalla classe `AbstractSequentialList<T>` (l'implementazione della classe `LinkedList<T>` è simile a quella delle classi per le liste concatenate discusse nel Capitolo 15). Se occorre una `List<T>` con accesso diretto efficiente agli elementi (cioè una lista con un'implementazione efficiente del metodo `get`), si utilizzi la classe `ArrayList<T>` o `Vector<T>` oppure una nuova classe derivata da una di queste due. Se non è necessario l'accesso diretto efficiente, ma si devono poter spostare sequenzialmente gli elementi lungo la lista in modo efficiente, si utilizzi la classe `LinkedList<T>` o una classe da essa derivata.

L'interfaccia `SortedSet<T>` e la classe concreta `TreeSet<T>` sono state progettate per essere utilizzate in implementazioni dell'interfaccia `Set<T>` che forniscano una versione efficiente dell'estrazione di elementi (quindi un'implementazione efficiente dei metodi `contains` e simili). Qui non saranno discusse l'interfaccia `SortedSet<T>` o la classe `TreeSet<T>`, ma si dovrebbe tenere presente la loro esistenza per sapere come trovare informazioni su di esse nella documentazione di Java in caso di necessità.

Le classi `ArrayList<T>` e `Vector<T>` e le interfacce `Iterator<T>` e `ListIterator<T>` fanno parte del package `java.util`.
Tutte le eccezioni citate sono del tipo non controllato, quindi non è necessario gestirle in un blocco `catch` o dichiararle con una clausola `throws` (se le eccezioni non sono ancora state affrontate, possono essere considerate errori a tempo di esecuzione).
L'eccezione `NoSuchElementException` fa parte del package `java.util`, quindi il codice che fa riferimento a questa classe deve importare quel package. Tutte le altre eccezioni menzionate, facendo parte del package `java.lang`, non richiedono package aggiuntivi.

COSTRUTTORI

`public ArrayList(int capacitaIniziale)`
Crea un oggetto `ArrayList<T>` vuoto della `capacitaIniziale` specificata. Quando è necessario aumentare la capacità dell'`ArrayList<T>`, questa viene raddoppiata.

`public ArrayList()`
Crea un oggetto `ArrayList<T>` vuoto con capacità iniziale uguale a `10`. Quando è necessario aumentare la capacità dell'`ArrayList<T>`, questa viene raddoppiata.

`public ArrayList(Collection<? extends T> c)`
Crea un oggetto `ArrayList<T>` contenente tutti gli elementi della collezione `c`, mantenendone l'ordine. Gli elementi nel nuovo oggetto `ArrayList<T>` avranno quindi lo stesso indice che avevano in `c`.
Il nuovo oggetto `ArrayList<T>` è solo una copia superficiale della collezione passata come argomento, poiché contiene riferimenti agli elementi di `c` (e non riferimenti a copie degli elementi di `c`).

`public Vector(int capacitaIniziale)`
Crea un vettore vuoto della `capacitaIniziale` specificata. Quando è necessario aumentare la capacità, questa viene raddoppiata.

`public Vector()`
Crea un vettore vuoto con capacità iniziale uguale a `10`. Quando è necessario aumentare la capacità, questa viene raddoppiata.

`public Vector(Collection<? extends T> c)`
Crea un vettore contenente tutti gli elementi della collezione `c`, mantenendone l'ordine. Gli elementi del nuovo vettore hanno quindi lo stesso indice che avevano in `c`.
Il nuovo vettore è solo una copia superficiale della collezione specificata, poiché contiene riferimenti agli elementi di `c` e non riferimenti a copie degli elementi di `c`.

`public Vector(int capacitaIniziale, int incrementoCapacita)`
Crea un vettore vuoto della `capacitaIniziale` e con l'`incrementoCapacita` specificati. Quando è necessario aumentare la capacità del vettore, verrà riservato spazio per `incrementoCapacita` nuovi elementi (la classe `ArrayList<T>` non ha un costruttore corrispondente a questo).

METODI TIPO ARRAY COMUNI AD `ArrayList<T>` E `Vector<T>`

`public T set(int indice, T nuovoElemento)`
Imposta a `nuovoElemento` l'elemento alla posizione `indice`. Viene restituito l'elemento che si trovava precedentemente in quella posizione. Facendo un'analogia con un array `a`, ciò corrisponde a impostare l'elemento `a[indice]` al valore `nuovoElemento`. L'indice deve essere maggiore o uguale a 0 e strettamente minore delle dimensioni correnti della lista.

Figura 16.6 Metodi delle classi `ArrayList<T>` e `Vector<T>`. *(segue)*

`public T get(int indice)`
Restituisce l'elemento alla posizione specificata. Corrisponde a restituire `a[indice]` dato un array `a`. L'indice deve essere maggiore o uguale a 0 e strettamente minore delle dimensioni correnti dell'oggetto chiamante.

METODI PER L'AGGIUNTA DI ELEMENTI AD `ArrayList<T>` E `Vector<T>`

`public boolean add(T nuovoElemento)`
Inserisce `nuovoElemento` alla fine della lista di elementi dell'oggetto chiamante, incrementando di 1 la dimensione della lista. Se necessario viene aumentata la capacità dell'oggetto chiamante. Restituisce `true` se l'aggiunta è stata completata correttamente. Questo metodo viene spesso utilizzato come se fosse un metodo `void`.

`public boolean add(int indice, T nuovoElemento)`
Inserisce `nuovoElemento` alla posizione specificata dell'oggetto chiamante e incrementa di 1 la dimensione di quest'ultimo. Gli elementi con indice maggiore o uguale a `indice` vengono spostati in avanti di una posizione.
L'`indice` deve essere maggiore o uguale a 0 e minore o uguale alle dimensioni dell'oggetto chiamante prima di questa aggiunta.
Si noti che è possibile utilizzare questo metodo per aggiungere un elemento dopo l'ultimo. Se necessario viene incrementata la capacità dell'oggetto chiamante.

`public boolean addAll(Collection<? extends T> c)`
Aggiunge tutti gli elementi contenuti in `c` dopo gli elementi dell'oggetto chiamante secondo l'ordine dettato da un iteratore di `c`. Il comportamento di questo metodo non è garantito se la collezione `c` coincide con l'oggetto chiamante o con qualunque collezione che include direttamente o indirettamente l'oggetto chiamante.

`public boolean addAll(int indice, Collection<? extends T> c)`
Inserisce nell'oggetto chiamante tutti gli elementi contenuti in `c`, cominciando dalla posizione `indice`. Gli elementi vengono inseriti nell'ordine determinato da un iteratore di `c`. Gli elementi che si trovavano originariamente nella posizione `indice` e in quelle successive vengono spostati a indici più grandi.

METODI PER LA RIMOZIONE DI ELEMENTI DA `ArrayList<T>` E `Vector<T>`

`public T remove(int indice)`
Elimina l'elemento alla posizione specificata e lo restituisce. La dimensione dell'oggetto chiamante viene ridotta di un'unità, mentre la sua capacità rimane invariata. L'indice di ognuno degli elementi dell'oggetto chiamante con indice maggiore di `indice` viene decrementato di un'unità.
Il valore di `indice` deve essere maggiore o uguale a 0 e minore della dimensione dell'oggetto chiamante prima della rimozione.

`public boolean remove(Object elemento)`
Rimuove dall'oggetto chiamante la prima occorrenza di `elemento`. Se `elemento` era contenuto nell'oggetto, l'indice di ognuno degli elementi successivi viene decrementato di 1. Restituisce `true` se l'elemento è stato trovato (e rimosso), altrimenti restituisce `false`. Se l'elemento è stato rimosso, la dimensione dell'oggetto chiamante viene ridotta di 1, mentre la capacità rimane invariata.

`protected void removeRange(int daIndice, int aIndice)`
Rimuove tutti gli elementi con indice maggiore o uguale a `daIndice` e strettamente minore di `aIndice`.
Si noti che questo metodo è `protected` e non `public`.

`public void clear()`
Rimuove dall'oggetto chiamante tutti gli elementi e imposta a 0 la dimensione.

Figura 16.6 Metodi delle classi `ArrayList<T>` e `Vector<T>`. *(segue)*

METODI DI RICERCA PER `ArrayList<T>` E `Vector<T>`
`public boolean isEmpty()` Restituisce `true` se l'oggetto chiamante è vuoto (cioè se ha dimensione 0), altrimenti restituisce `false`.
`public boolean contains(Object obiettivo)` Restituisce `true` se `obiettivo` è un elemento dell'oggetto chiamante, altrimenti restituisce `false`. Per verificare l'uguaglianza utilizza il metodo `equals` dell'oggetto `obiettivo`.
`public int indexOf(Object obiettivo)` Restituisce l'indice del primo elemento uguale a `obiettivo`. Per verificare l'uguaglianza usa il metodo `equals` dell'oggetto `obiettivo`. Se `obiettivo` non viene trovato, restituisce `-1`.
`public int lastIndexOf(Object obiettivo)` Restituisce l'indice dell'ultimo elemento uguale a `obiettivo`. Per verificare l'uguaglianza, usa il metodo `equals` dell'oggetto `obiettivo`. Se `obiettivo` non viene trovato, restituisce `-1`.
ITERATORI PER `ArrayList<T>` E `Vector<T>`
`public Iterator<T> iterator()` Restituisce un iteratore per l'oggetto chiamante. Gli iteratori per le collezioni vengono presentati nel Paragrafo 16.3.
`public ListIterator<T> listIterator()` Restituisce un `ListIterator<T>` per l'oggetto chiamante. Il `ListIterator<T>` è trattato nel Paragrafo 16.3.
`ListIterator<T> listIterator(int indice)` Restituisce un iteratore di lista per l'oggetto chiamante che parte dalla posizione `indice`. Il primo elemento restituito dall'iteratore è quello alla posizione `indice` (gli iteratori per le collezioni sono presentati nel Paragrafo 16.3).
CONVERSIONE DI `ArrayList<T>` E `Vector<T>` IN ARRAY
`public Object[] toArray()` Restituisce un array contenente tutti gli elementi dell'oggetto chiamante, nelle stesse posizioni.
`public <E> E[] toArray(E[] a)` Si noti che il tipo di parametro `E` è diverso da `T`. Quindi `E` può essere qualunque tipo di oggetto, non è necessario che coincida con il tipo `T` della collezione. Per esempio, `E` potrebbe essere un antenato di `T`. Restituisce un array contenente tutti gli elementi dell'oggetto chiamante, nelle stesse posizioni. L'argomento `a` serve principalmente per specificare il tipo di array da restituire. In dettaglio: il tipo dell'array restituito è lo stesso di `a`. Se `a` può contenere la collezione, viene utilizzato per contenere gli elementi dell'array restituito, altrimenti viene creato un nuovo array dello stesso tipo di `a`. Se `a` ha più elementi dell'oggetto chiamante, gli elementi in più sono impostati a `null`.
GESTIONE DELLA MEMORIA PER `ArrayList<T>` E `Vector<T>`
`public int size()` Restituisce il numero di elementi contenuti nell'oggetto chiamante.
`public int capacity()` Restituisce l'attuale capacità dell'oggetto chiamante.
`public void ensureCapacity(int nuovaCapacita)` Incrementa la capacità dell'oggetto chiamante per garantire che possa contenere un numero di elementi pari almeno a `nuovaCapacita`. L'uso di questo metodo può in alcuni casi aumentare l'efficienza.

Figura 16.6 Metodi delle classi `ArrayList<T>` e `Vector<T>`. *(segue)*

`public void trimToSize()` Riduce la capacità dell'oggetto chiamante, in modo che coincida con la sua dimensione attuale. È utilizzato per risparmiare spazio in memoria.
COPIA PER `ArrayList<T>` E `Vector<T>`
`public Object clone()` Restituisce una copia superficiale dell'oggetto chiamante. La copia superficiale viene trattata nel riquadro "Copia in profondità (*deep copy*) e copia superficiale (*shallow copy*)".

Figura 16.6 Metodi delle classi `ArrayList<T>` e `Vector<T>`.

Copia in profondità (*deep copy*) e copia superficiale (*shallow copy*)

La **copia in profondità** (*deep copy*) di un oggetto è una copia che non ha alcun riferimento (*reference*) in comune con l'oggetto originale tranne che per gli oggetti immutabili. Gli oggetti immutabili sono oggetti che non espongono metodi di *set* per i propri attributi. Come riportato nel Capitolo 9, la classe `String` è definita in modo da generare oggetti immutabili. Ogni copia che non sia una copia in profondità è definita **copia superficiale** (*shallow copy*).

Si consideri per esempio il metodo definito nel Listato 9.15 e qui di seguito riportato:

```
public Animale getPrimo(){
    return primo;
}
```

Tale metodo restituisce una copia superficiale dell'animale memorizzato nella variabile d'istanza `primo`.

Il metodo `getPrimo` definito nel seguente modo restituisce al contrario una copia in profondità dell'animale memorizzato nella variabile d'istanza `primo`:

```
public Animale getPrimo(){
    return new Animale(primo.getNome(), primo.getEta(), primo.getPeso());
}
```

Se si utilizza il secondo metodo (quello che restituisce una copia in profondità), qualsiasi modifica apportata all'oggetto di tipo `Animale` restituito dal metodo non si ripercuote sull'oggetto di tipo `Animale` memorizzato nella variabile d'istanza `primo`.

16.1.4 Differenze tra `ArrayList<T>` e `Vector<T>`

`ArrayList<T>` e `Vector<T>` possono essere considerate equivalenti per molti utilizzi. Le differenze tra le due classi sono limitate. I metodi presenti in entrambe sono riportati nella Figura 16.6. La classe `Vector<T>` ha anche altri metodi che non sono presenti nella classe `ArrayList<T>` e che non sono illustrati nella Figura 16.6. La maggior parte di questi metodi, tuttavia, corrisponde principalmente a nomi alternativi per metodi presenti sia in `ArrayList<T>`, sia in `Vector<T>`. Nessun metodo di `Vector<T>` fa niente che non possa essere fatto facilmente anche con un `ArrayList<T>`. La classe

`ArrayList<T>` è considerata più efficiente della classe `Vector<T>`. La differenza principale tra le due classi è che `ArrayList<T>` è più recente di `Vector<T>` ed è stata creata come parte della libreria Java delle collezioni, mentre `Vector<T>` è una classe più vecchia che è stata adattata con l'aggiunta di nuovi metodi per renderla compatibile con la libreria di collezioni. Si consiglia di utilizzare sempre `ArrayList<T>` al posto di `Vector<T>`. Tuttavia, poiché molto del codice già esistente utilizza `Vector<T>`, è necessario avere familiarità anche con questa classe[1].

16.1.5 Versione non parametrica della libreria delle collezioni

Prima della versione 5.0, Java non prevedeva i tipi parametrici. La libreria delle collezioni consisteva pertanto di classi e interfacce ordinarie, come `Collection`, `List`, `ArrayList` e così via, tutte senza tipi parametrici. Nonostante questa vecchia libreria sia stata soppiantata dalla nuova versione basata sui tipi generici, le classi e le interfacce della vecchia versione, prive di parametri, sono ancora presenti nelle librerie standard e in molto del codice già esistente. Le vecchie classi e interfacce prive di tipi parametrici non sono più necessarie e a volte possono essere più difficili da utilizzare e meno versatili delle nuove versioni basate sui tipi generici. Si dovrebbe quindi evitare di utilizzare le vecchie classi e interfacce, tuttavia capiterà spesso di trovarle utilizzate nel vecchio codice. In tal caso, non si sbaglierà troppo interpretando `Collection` come `Collection<Object>`, `ArrayList` come `ArrayList<Object>` e così via, anche se ciò, a rigore, non è del tutto corretto. Per esempio, le classi `ArrayList` e `ArrayList<Object>` non sono la stessa cosa, anche se sono molto simili.

Dimenticarsi il <T>

Se si omette il `<T>` o un nome di classe corrispondente, usando per esempio `ArrayList` al posto di `ArrayList<String>`, si potrebbe ottenere un errore in fase di compilazione. In tal caso, l'errore apparirà decisamente strano. Il problema è che `ArrayList` e altri nomi di classi e interfacce senza il `<T>` hanno in realtà un significato (si veda il paragrafo "Versione non parametrica della libreria delle collezioni"). L'unica difesa contro questa possibilità di errore è fare molta attenzione e cercare un `<T>` o un `<Nome_Classe>` mancante in caso in cui si ricevesse un errore particolarmente strano.

A volte gli avvertimenti (*warning*) del compilatore possono essere d'aiuto per correggere questi errori. Se si ottiene un avvertimento riferito a una conversione di tipo da un nome di classe senza `<T>` a uno con `<T>` o con un `<Nome_Classe>`, è opportuno cercare un `<T>` o un `<Nome_Classe>` mancante.

In conclusione, va detto che a volte il codice verrà compilato e addirittura funzionerà correttamente anche se è stato dimenticato il `<T>` nel nome di una classe della libreria delle collezioni.

[1] In realtà, la differenza principale tra le classi `Vector<T>` e `ArrayList<T>` è che gli oggetti di tipo `Vector<T>` sono *sincronizzati*, mentre quelli di tipo `ArrayList<T>` non lo sono. Tuttavia, la sincronizzazione è un tema che non sarà trattato in queste pagine e che non è rilevante per le modalità di programmazione che verranno discusse.

16.2 Mappe

La libreria Java delle **mappe** è simile a quella delle collezioni, a parte il fatto che lavora su collezioni di coppie ordinate. Gli oggetti della libreria delle mappe possono implementare funzioni e relazioni matematiche e quindi possono essere utilizzati per costruire classi per la gestione di basi di dati. Si pensi a una coppia come quella composta da una chiave `K` (da cercare) e da un valore associato `V`. Per esempio, la chiave potrebbe essere la matricola di uno studente e il valore potrebbe essere un oggetto contenente le informazioni sullo studente (come nome, relatore, indirizzo o numero di telefono) associato a quel codice. Le classi e interfacce di questa libreria più comunemente utilizzate sono mostrate nella Figura 16.7. In questo capitolo ci si concentrerà sull'interfaccia `Map<K, V>` e sulle classi `AbstractMap<K, V>` e `HashMap<K, V>`.

Dato che l'interfaccia di tipo mappa associa una chiave a un valore, ora è necessario specificare due tipi di parametro al posto del singolo tipo usato per le collezioni. L'interfaccia `Map<K, V>` specifica le operazioni di base che tutte le classi mappa dovrebbero implementare. Un riassunto di queste operazioni è presentato nella Figura 16.8. Si noti che ci sono molte similitudini con l'interfaccia `Collection<T>`.

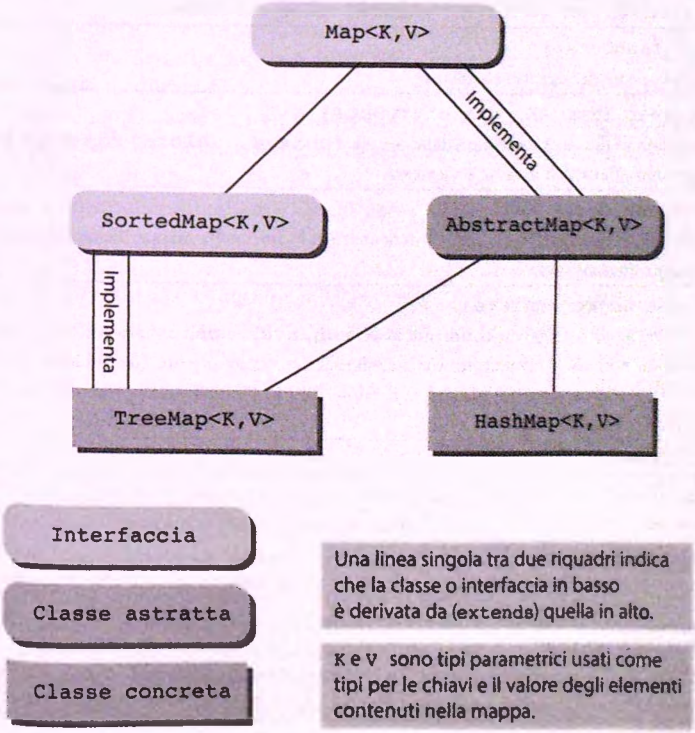


Figura 16.7 Panoramica della libreria delle mappe.

L'interfaccia `Map<K, V>` fa parte del package `java.util`.

COSTRUTTORI

Nonostante non sia strettamente richiesto dall'interfaccia, qualunque classe che implementi `Map<K, V>` dovrebbe avere almeno due costruttori: uno senza parametri, che crei un oggetto `Map<K, V>` vuoto, e uno con un parametro di tipo `Map<K, V>` che crei un oggetto con gli stessi elementi di quello passato come argomento.

METODI

`boolean isEmpty()`
Restituisce `true` se l'oggetto chiamante è vuoto, altrimenti restituisce `false`.

`public boolean containsValue(Object valore)`
Restituisce `true` se l'oggetto chiamante contiene almeno una chiave associata a un'istanza di `valore`.

`public boolean containsKey(Object chiave)`
Restituisce `true` se l'oggetto chiamante contiene `chiave` come una delle sue chiavi.

`public boolean equals(Object altra)`
Questo è il metodo `equals` per la mappa, non quello per gli elementi della mappa. Ridefinisce il metodo `equals` ereditato.

`public int size()`
Restituisce il numero di coppie `(chiave, valore)` nell'oggetto chiamante.

`public int hashCode()`
Restituisce il codice *hash* dell'oggetto chiamante.

`public Set<Map.Entry<K, V>> entrySet()`
Restituisce una vista di tipo `Set` costituita dalle coppie `(chiave, valore)` della mappa. Le modifiche alla mappa sono riflesse nell'insieme e viceversa.

`public Collection<V> values()`
Restituisce una vista di tipo `Collection` contenente tutti i valori nella mappa. Le modifiche alla mappa sono riflesse nella collezione e viceversa.

`public V get(Object chiave)`
Restituisce il valore al quale l'oggetto chiamante associa `chiave`. Se `chiave` non è nella mappa, viene restituito `null`. Si noti che un valore `null` non significa necessariamente che la chiave non sia nella mappa, perché è possibile associare a una chiave il valore `null`. Per distinguere tra i due casi può essere utilizzato il metodo `containsKey`.

METODI OPZIONALI

I metodi seguenti sono opzionali, il che significa che devono comunque essere implementati, ma l'implementazione può limitarsi a generare una `UnsupportedOperationException` se, per qualche motivo, non si vuole fornire un'implementazione "vera" dei metodi. Una `UnsupportedOperationException` è una `RuntimeException`, quindi non è necessario gestirla o dichiararla in una clausola `throws`.

`public V put(K chiave, V valore)` (opzionale)
Associa `chiave` a `valore` nella mappa. Se `chiave` era già associata a un valore, viene sostituito e restituito il vecchio valore, altrimenti viene restituito `null`.

Figura 16.8 Intestazione dei metodi dell'interfaccia `Map<K, V>`. (*segue*)

`public void putAll(Map<? extends K, ? extends V> mappaDaAggiungere)` (opzionale) Aggiunge alla mappa chiamante tutte le associazioni di `mappaDaAggiungere`.
`public V remove(Object chiave)` (opzionale) Rimuove l'associazione per la `chiave` specificata. Se la `chiave` non era nella mappa, viene restituito `null`, altrimenti viene restituito il valore originariamente associato alla chiave.

Figura 16.8 Intestazione dei metodi dell'interfaccia `Map<K, V>`.

16.2.1 Classi mappa concrete

La classe astratta `AbstractMap<K, V>` è comoda se si desidera implementare l'interfaccia `Map<K, V>`, esattamente come la classe `AbstractSet<T>` lo era per l'interfaccia `Set<T>`. Definendo una classe derivata da `AbstractMap<K, V>`, sarà necessario definire non solo i metodi astratti, ma anche tutti gli altri metodi che si intendono utilizzare. Di solito, ha più senso usare (o definire classi derivate da) le classi `HashMap<K, V>` e `TreeMap<K, V>`, che sono derivate da `AbstractMap<K, V>` e costituiscono implementazioni complete dell'interfaccia `Map<K, V>`. Tuttavia, se si vuole implementare una mappa con strutture dati personalizzate, è appropriato derivare classi da `AbstractMap<K, V>`.

In questo capitolo ci si concentrerà solo sulla classe `HashMap<K, V>`, che è un'implementazione concreta dell'interfaccia `Map<K, V>`. Internamente, la classe utilizza una tabella *hash*. Si noti che questa classe non fornisce alcuna garanzia sull'ordine degli elementi contenuti nella mappa. Se è richiesta una struttura ordinata, bisognerebbe utilizzare la classe `TreeMap<K, V>` (che internamente organizza i propri elementi in una struttura ad albero) o la classe `LinkedHashMap<K, V>`, che usa una lista a doppio concatenamento per mantenere l'ordine in un oggetto `HashMap<K, V>`. La classe `LinkedHashMap<K, V>` è derivata dalla `HashMap<K, V>`.

È utile conoscere come operano le tabelle di *hash* per ottimizzare un programma che utilizza una `HashMap`. Quando nel Paragrafo 15.3 è stata presentata una tabella di *hash*, è stato utilizzato un array di lunghezza fissa nel quale ogni elemento referenziava una lista concatenata. Una funzione di *hash* associava a un valore, come una stringa, un indice nell'array. Se la dimensione di questo array è molto più piccola rispetto al numero di elementi aggiunti, il numero di collisioni aumenta, deteriorando le prestazioni del sistema. Al contrario, se la dimensione dell'array è eccessiva rispetto al numero di elementi aggiunti, si ha uno spreco di memoria. Andrebbe ricercato un simile compromesso tra dimensione della tabella e numero di elementi inseriti anche quando si utilizza la classe `HashMap<K, V>`. Uno dei costruttori della classe `HashMap<K, V>` permette di specificare una **capacità iniziale** e un **fattore di carico**. La capacità iniziale determina quanti "blocchi" esistono nella tabella *hash*. Il fattore di carico è un numero compreso tra 0 e 1 che specifica una percentuale tale per cui se il numero di elementi aggiunti alla tabella *hash* supera questo fattore di carico, la capacità della tabella viene incrementata automaticamente. Il fattore di carico di default è 0.75 e la capacità iniziale di default è 16. Ciò significa che la capacità verrà aumentata (all'incirca del doppio) ogni 12 elementi aggiunti alla mappa. Questo processo è detto **rehashing** e può richiedere un tempo non trascurabile per mappe con molti elementi. Nonostante la capacità venga incrementata automaticamente se necessario, un programma verrà eseguito in modo più efficiente se la capacità iniziale è impostata al numero di elementi che ci si aspetta verranno aggiunti alla mappa.

La classe `HashMap<K, V>` implementa ovviamente tutti i metodi dell'interfaccia `Map<K, V>`, senza aggiungerne di nuovi a parte i costruttori e un'implementazione del metodo `clone()`. Un riassunto riguardante i costruttori e il metodo `clone()` della classe `HashMap<K, V>` è riportato nella Figura 16.9.

Come nel caso della classe `HashSet<T>`, se si utilizza una classe propria come tipo parametrico `K` in `HashMap<K, V>`, la classe deve ridefinire i due metodi seguenti:

```
public int hashCode();
public boolean equals(Object obj);
```

Questi metodi sono necessari per indicizzare e confrontare le chiavi. Si veda la discussione del Paragrafo 16.1 per maggiori dettagli.

Nel Listato 16.2 è riportato un programma che mostra l'utilizzo della classe `HashMap<K, V>`. In questo esempio, il programma usa la variabile d'istanza nome come chiave da associare a un oggetto `Studente` definito come nel Listato 10.2. Vari oggetti di tipo `Studente` vengono creati e aggiunti alla mappa utilizzando il nome come chiave. L'utente ha la possibilità di inserire nomi finché non inserisce una riga vuota. I nomi presenti nella mappa vengono estratti e le relative informazioni vengono poi visualizzate.

La classe `HashMap<K, V>` fa parte del package `java.util`. La classe `HashMap<K, V>` estende la classe `AbstractMap<K, V>` e implementa l'interfaccia `Map<K, V>`. La classe `HashMap<K, V>` implementa tutti i metodi dell'interfaccia `Map<K, V>` (Figura 16.8). Gli unici metodi in più della classe `HashMap<K, V>` sono i costruttori. Tutte le eccezioni citate sono del tipo non controllato, quindi non è necessario gestirle in un blocco `catch` o la dichiarazione in una clausola `throws`. Tutte le eccezioni citate, inoltre, appartengono al package `java.lang` e pertanto non richiedono l'importazione di package aggiuntivi.
`public HashMap()` Crea una nuova mappa vuota con capacità iniziale 16 e fattore di carico 0.75.
`public HashMap(int capacitaIniziale)` Crea una nuova mappa vuota con capacità iniziale specificata e fattore di carico 0.75. Genera una `IllegalArgumentException` se `capacitaIniziale` è negativa.
`public HashMap(int capacitaIniziale, float fattoreDiCarico)` Crea una nuova mappa vuota con capacità iniziale e fattore di carico specificati. Genera una `IllegalArgumentException` se `capacitaIniziale` è negativa o `fattoreDiCarico` è non positivo.
`public HashMap(Map<? extends K, ? extends V> m)` Crea una nuova mappa contenente le stesse associazioni della mappa `m`. La capacità iniziale è impostata alle dimensioni di `m` e il fattore di carico a 0.75. Genera una `NullPointerException` se `m` è `null`.
`public Object clone()` Crea una copia superficiale di questa istanza e la restituisce. Chiavi e valori non sono clonati.
Tutti gli altri metodi sono quelli già descritti per l'interfaccia `Map<K, V>` (Figura 16.8).

Figura 16.9 Metodi della classe `HashMap<K, V>`.

LISTATO 16.2 Esempio d'uso della classe `HashMap<T>`.

MyLab

```
// Questa classe usa la classe Studente definita nel Capitolo 10.
import java.util.HashMap;
import java.util.Scanner;

public class HashMapDemo {
   public static void main(String[] args) {
        // Crea una HashMap con capacità iniziale 10 e fattore di
        // carico di default
        HashMap<String, Studente> studenti =
        new HashMap<String, Studente>(10);

        // Aggiunge alla mappa alcuni studenti utilizzando i loro
        // nomi come chiave
        studenti.put("Alessio",
             new Studente("Alessio", 701554));
        studenti.put("Giulio",
             new Studente("Giulio", 701784));
        studenti.put("Lorenzo",
             new Studente("Lorenzo ", 702185));
        studenti.put("Marco ",
             new Studente("Marco", 701812));
        studenti.put("Simone",
             new Studente("Simone ", 701156));

        System.out.print("Aggiunti Alessio, Giulio, Lorenzo, Marco, ");
        System.out.println("e Simone alla mappa.");

        // Richiesta di un nome all'utente.
        // Se il nome è nella mappa, lo si stampa.
        Scanner tastiera = new Scanner(System.in);
        String nome = "";
        do {
             System.out.print("\nInserire un nome da cercare. ");
             System.out.println("Premere Invio per uscire.");
             nome = tastiera.nextLine();
             if (studenti.containsKey(nome)) {
                  Studente i = studenti.get(nome);
                  System.out.println("Nome trovato: ");
                  i.scriviOutput();
             } else if (!nome.equals("")) {
                  System.out.println("Nome non trovato.");
             }
        } while (!nome.equals(""));
   }
}
```

Esempio di output

```
Aggiunti Alessio, Giulio, Lorenzo, Marco e Simone alla mappa.
```

```
Inserire un nome da cercare. Premere Invio per uscire.
Giulio
Nome trovato:
Nome: Giulio
Matricola: 701784

Inserire un nome da cercare. Premere Invio per uscire.
Alessio
Nome trovato:
Nome: Alessio
Matricola: 701554

Inserire un nome da cercare. Premere Invio per uscire.
Lorenzo
Nome trovato:
Nome: Lorenzo
Matricola: 702185

Inserire un nome da cercare. Premere Invio per uscire.
Francesco
Nome non trovato.

Inserire un nome da cercare. Premere Invio per uscire.
```

16.3 Iteratori

Un iteratore è un oggetto utilizzato insieme a una collezione per fornire un accesso sequenziale agli elementi della collezione stessa. In questo paragrafo verranno discussi gli iteratori in generale e in particolare la loro applicazione alla libreria delle collezioni.

16.3.1 Il concetto di iteratore

Prima di introdurre l'interfaccia Java `Iterator`, verrà presentata l'idea intuitiva di iteratore. Un iteratore è qualcosa che permette di esaminare ed eventualmente modificare gli elementi di una collezione operando in un qualche tipo di ordine sequenziale. Un iteratore impone quindi un ordinamento agli elementi della collezione anche quando questa, come nel caso della classe `HashSet<T>`, non impone alcun ordine ai propri elementi.

Un esempio di entità che non è un oggetto (e quindi, a rigore, non è un iteratore Java), ma che soddisfa l'idea intuitiva di iteratore è una variabile `i` di tipo `int` utilizzata con un array `a`. Questo iteratore `i` può essere fatto partire dal primo elemento dell'array in questo modo:

```
i = 0;
```

L'iteratore può fornire l'elemento corrente: è semplicemente `a[i]`. L'iteratore può inoltre passare all'elemento successivo e restituirlo come segue:

```
i++;
"Restituisce a[i]"
```

Il concetto di iteratore è semplice, ma sufficientemente potente da essere utilizzato spesso.

16.3.2 L'interfaccia `Iterator<T>`

Java formalizza il concetto di iteratore per mezzo dell'interfaccia generica `Iterator<T>`. Qualunque oggetto di qualunque classe implementi l'interfaccia `Iterator<T>` è un `Iterator<T>`. Quindi, l'indice di un array non è un `Iterator<T>` Java.

Un oggetto `Iterator<T>` non ha senso da solo, ma deve essere associato a una collezione. Com'è realizzata l'associazione? In Java, qualunque classe implementi l'interfaccia `Collection<T>` deve offrire il metodo `iterator()` che restituisce un `Iterator<T>`. Per esempio, sia `c` un'istanza della classe `HashSet<T>`, con qualche classe sostituita al posto di `T`. Per rendere più concreto l'esempio, sia `String` la classe sostituita a `T`, così che `c` è un'istanza della classe `Collection<String>`. Si può ottenere un iteratore per `c` come segue:

```
Iterator<String> iteratorePerC = c.iterator();
```

Potrebbe non essere noto di quale classe sia effettivamente l'istanza `iteratorePerC`, ma sicuramente rispetterà l'interfaccia `Iterator<String>` e quindi esporrà i metodi dell'interfaccia `Iterator<T>`. Questi metodi sono presentati nella Figura 16.10.

Il Listato 16.3 illustra un semplice esempio di utilizzo degli iteratori con un oggetto `HashSet<T>`. Un oggetto `HashSet<T>` non impone alcun ordine ai propri elementi, ma l'iteratore ne stabilisce uno, cioè quello in cui gli elementi vengono prodotti dal suo metodo `next()`. Non ci sono vincoli su questo ordinamento. Eseguendo due volte il programma nel Listato 16.3, l'ordine in cui sono stampati gli elementi sarà quasi certamente lo stesso nei due casi. Tuttavia, non sarebbe un errore se gli elementi fossero restituiti in ordine diverso ogni volta che viene eseguito il programma.

L'interfaccia `Iterator<T>` fa parte del package `java.util`. Tutte le eccezioni citate sono del tipo non controllato, quindi non è necessario gestirle in un blocco `catch` o dichiararle in una clausola `throws`. L'eccezione `NoSuchElementException` appartiene al package `java.util`, che deve essere quindi importato se il codice utilizza questa classe. Tutte le altre eccezioni appartengono al package `java.lang` e quindi non richiedono l'importazione di package aggiuntivi.
`public T next()` Restituisce l'elemento successivo della collezione che ha prodotto l'iteratore. Genera un'eccezione `NoSuchElementException` se non esiste l'elemento successivo.
`public boolean hasNext()` Restituisce `true` se il metodo `next()` non ha ancora restituito tutti gli elementi della collezione, altrimenti restituisce `false`.
`public void remove()` (opzionale) Rimuove dalla collezione l'ultimo elemento restituito da `next`. Questo metodo può essere chiamato solo una volta per ogni chiamata a `next`. Se la collezione è stata modificata senza utilizzare `remove`, il comportamento dell'iteratore non è specificato (e quindi dovrebbe essere considerato imprevedibile). Genera una `IllegalStateException` se il metodo `next` non è ancora stato chiamato o se il metodo `remove` è già stato chiamato dopo l'ultima chiamata di `next`. Genera una `UnsupportedOperationException` se l'operazione di rimozione non è supportata da questo `Iterator<T>`.

Figura 16.10 Metodi dell'interfaccia `Iterator<T>`.

Se la collezione utilizzata con un `Iterator<T>` impone un ordine ai propri elementi, come nel caso di `ArrayList<T>`, allora l'`Iterator<T>` restituirà gli elementi in quell'ordine. Se è necessario ordinare gli elementi di un oggetto `HashSet<T>`, si può utilizzare la classe `LinkedHashSet<T>`, che internamente utilizza una lista a doppio concatenamento per immagazzinare gli elementi nell'ordine con il quale vengono aggiunti.

MyLab

LISTATO 16.3 Un iteratore.

```
import java.util.HashSet;
import java.util.Iterator;

public class IteratoreHashSetDemo {
    public static void main(String[] args) {
        HashSet<String> s = new HashSet<String>();

        s.add("salute");
        s.add("amore");
        s.add("denaro");

        System.out.println("L'insieme contiene:");

        Iterator<String> i = s.iterator();
        while (i.hasNext())
            System.out.println(i.next());

        i.remove();

        System.out.println();
        System.out.println("Ora l'insieme contiene:");

        i = s.iterator();
        while (i.hasNext())
            System.out.println(i.next());

        System.out.println("Fine del programma.");
    }
}
```

Non è possibile "azzerare" un iteratore "all'inizio". Per eseguire un'altra iterazione bisogna creare un altro iteratore.

Esempio di output

```
L'insieme contiene:
salute
denaro
amore

Ora l'insieme contiene:
salute
denaro
Fine del programma.
```

L'oggetto `HashSet<T>` non ordina gli elementi che contiene, ma l'iteratore stabilisce un ordine per gli elementi.

Iteratori

Un iteratore è qualcosa che consente di esaminare ed eventualmente modificare gli elementi di una collezione in un qualche ordine sequenziale. Java formalizza questo concetto tramite le due interfacce `Iterator<T>` e `ListIterator<T>` (discussa di seguito).

I cicli *for-each* usati come iteratori

Un ciclo *for-each* non è, strettamente parlando, un iteratore (perché, tra le altre cose, non è un oggetto), ma svolge lo stesso compito di un iteratore, ovvero permette di passare in rassegna gli elementi di una collezione. Lavorando con le collezioni, si può spesso utilizzare un ciclo *for-each* al posto di un iteratore e il ciclo è di solito più semplice da utilizzare rispetto all'iteratore. Per esempio, nel Listato 16.4 il programma del Listato 16.3 è stato riscritto per utilizzare i cicli *for-each* al posto degli iteratori. Si noti che si è reso necessario un po' di lavoro aggiuntivo con la variabile `ultimo` per simulare il metodo `i.remove()`. A volte è più indicato un iteratore, in altri casi un ciclo *for-each*. Molti potrebbero sostenere che il codice sarebbe migliore se il primo ciclo basato su iteratore nel Listato 16.3 non fosse sostituito con un ciclo *for-each*.

LISTATO 16.4 I cicli *for-each* come iteratori

MyLab

```
import java.util.HashSet;

public class ForEachDemo {
   public static void main(String[] args) {
        HashSet<String> s = new HashSet<String>();

        s.add("salute");
        s.add("amore");
        s.add("denaro");

        System.out.println("L'insieme contiene:");

        String ultimo = null;
        for (String e : s) {
            ultimo = e;
            System.out.println(e);
        }

        s.remove(ultimo);

        System.out.println();
        System.out.println("Ora l'insieme contiene:");
```

```
        for (String e : s)
            System.out.println(e);

        System.out.println("Fine del programma.");
    }
}
```

L'output è lo stesso del Listato 16.3.

16.3.3 Iteratori di lista

La libreria delle collezioni contiene due interfacce di tipo iteratore: l'interfaccia `Iterator<T>`, già analizzata e che può essere utilizzata con qualunque classe implementi l'interfaccia `Collection<T>`, e l'interfaccia `ListIterator<T>`, progettata per lavorare con le collezioni che implementano l'interfaccia `List<T>`. L'interfaccia `ListIterator<T>` estende `Iterator<T>`. Un `ListIterator<T>` ha tutti i metodi di un `Iterator<T>`, più altri che abilitano nuove funzionalità: un `ListIterator<T>` può muoversi lungo la lista degli elementi della collezione in entrambe le direzioni e offre metodi, come `set` e `add`, che possono essere utilizzati per modificare gli elementi della collezione. I metodi dell'interfaccia `ListIterator<T>` sono riportati nella Figura 16.11.

L'interfaccia `ListIterator<T>` fa parte del package `java.util`. Il concetto di posizione del cursore è descritto nel testo e nella Figura 16.12. Tutte le eccezioni citate sono del tipo non controllato, quindi non è necessario gestirle in un blocco `catch` o dichiararle in una clausola `throws`. L'eccezione `NoSuchElementException` appartiene al package `java.util`, che deve essere quindi importato se il codice utilizza questa classe. Tutte le altre eccezioni appartengono al package `java.lang` e quindi non richiedono l'importazione di package aggiuntivi.
`public T next()` Restituisce l'elemento successivo della lista che ha prodotto l'iteratore. Più precisamente, restituisce l'elemento immediatamente successivo alla posizione del cursore. Genera un'eccezione `NoSuchElementException` se non esiste l'elemento successivo.
`public T previous()` Restituisce l'elemento precedente della lista che ha prodotto l'iteratore. Più precisamente, restituisce l'elemento immediatamente precedente la posizione del cursore. Se l'elemento precedente non esiste, genera un'eccezione `NoSuchElementException`.
`public boolean hasNext()` Restituisce `true` se esiste un elemento che possa essere restituito da `next()`, altrimenti restituisce `false`.
`public boolean hasPrevious()` Restituisce `true` se esiste un elemento che possa essere restituito da `previous()`, altrimenti restituisce `false`.
`public int nextIndex()` Restituisce l'indice dell'elemento che sarebbe restituito da una chiamata a `next()`; se il cursore è alla fine della lista restituisce la dimensione della lista.

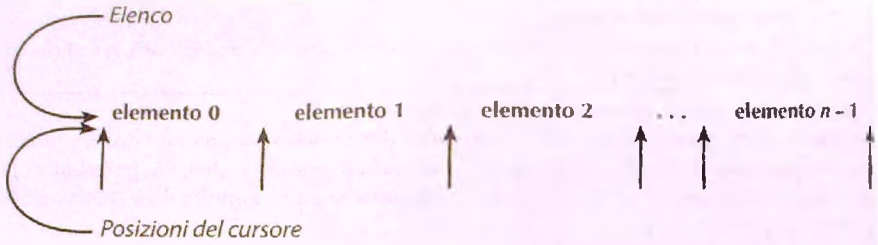
Figura 16.11 Metodi dell'interfaccia `ListIterator<T>`. *(segue)*

`public int previousIndex()` Restituisce l'indice dell'elemento che sarebbe restituito da una chiamata a `previous()`; se il cursore è all'inizio della lista, restituisce −1.
`public void add(T nuovoElemento)` *(opzionale)* Inserisce `nuovoElemento` nella posizione del cursore dell'iteratore (cioè prima del valore, se esistente, che sarebbe restituito da `next()` e dopo quello, se esistente, che sarebbe restituito da `previous()`). Non può essere utilizzato se è già stata effettuata una chiamata ad `add` o `remove` dopo l'ultima chiamata a `next()` o `previous()`. Genera una `IllegalStateException` se non sono stati chiamati né il metodo `next`, né il metodo `previous` o se dopo l'ultima invocazione di `next` o `previous` è già stato chiamato uno dei metodi `add` o `remove`.
`public void remove()` *(opzionale)* Rimuove dalla collezione l'ultimo elemento restituito da `next` o `previous`. Questo metodo può essere chiamato solo una volta per ogni chiamata a `next()` o a `previous()`. Non può essere utilizzato se dopo l'ultima chiamata a `next()` o `previous()` è già stata effettuata una chiamata ad `add` o `remove`. Genera una `IllegalStateException` se non sono stati chiamati né il metodo `next`, né il metodo `previous` o se dopo l'ultima invocazione di `next` o `previous` è già stato chiamato uno dei metodi `add` o `remove`.
`public void set(T nuovoElemento)` *(opzionale)* Sostituisce con `nuovoElemento` l'ultimo elemento restituito da `next()` o `previous()`. Non può essere utilizzato se dopo l'ultima chiamata a `next()` o `previous()` è stata effettuata una chiamata ad `add` o a `remove`. Genera una `IllegalStateException` se non sono stati richiamati né il metodo `next`, né il metodo `previous` o se dopo l'ultima invocazione di `next` o `previous` è stato chiamato uno dei metodi `add` o `remove`.

Figura 16.11 Metodi dell'interfaccia `ListIterator<T>`.

La libreria delle mappe non supporta direttamente le interfacce per gli iteratori, ma si possono utilizzare i metodi `keySet()`, `values()` e `entrySet()` delle mappe, che restituiscono collezioni iterabili contenenti rispettivamente le chiavi, i valori e le coppie (chiave, valore) di una mappa.

L'idea intuitiva di successivo e precedente è chiara, ma deve essere resa più precisa per poter comprendere il funzionamento dei metodi `next()` e `previous()` dell'interfaccia `ListIterator<T>`. Ogni `ListIterator<T>` ha un indicatore di posizione nella lista noto come **cursore**. Se la lista contiene n elementi, questi sono numerati da 0 a n-1, ma ci sono n+1 posizioni possibili per il cursore, come indicato nella Figura 16.12. Quando viene chiamato `next()`, viene restituito l'elemento immediatamente successivo alla posizione del cursore e quest'ultimo è spostato alla posizione successiva. Quando viene chiamato `previous()`, viene restituito l'elemento immediatamente precedente alla posizione del cursore e questo è spostato all'indietro nella posizione precedente.



La posizione iniziale di default per il cursore è quella più a sinistra.

Figura 16.12 Le posizioni del cursore per l'interfaccia `ListIterator<T>`.

L'interfaccia `ListIterator<T>`

L'interfaccia `ListIterator<T>` estende `Iterator<T>` e si differenzia da questa per due capacità aggiuntive: un `ListIterator<T>` può muoversi lungo la lista degli elementi della collezione in entrambe le direzioni e ha metodi, come `set` e `add`, che possono modificare gli elementi della collezione.

`next` può restituire un riferimento a un oggetto

Se `i` è un iteratore, `i.next()` restituisce un elemento della collezione che ha creato `i`, ma "restituisce un elemento" può avere due significati:

1. `i.next()` potrebbe restituire una copia dell'elemento della collezione;
2. `i.next()` potrebbe restituire un riferimento all'elemento della collezione.

Nel caso (1), una modifica al risultato di `i.next()` *non* modificherà l'elemento della collezione, a patto che la copia sia in profondità. Nel caso (2), invece, una modifica al risultato di `i.next()` *cambierà* l'elemento della collezione. Le API per `Iterator<T>` e `ListIterator<T>` non specificano quale delle due politiche debba essere seguita, ma gli iteratori delle classi predefinite, come `ArrayList<T>` e `HashSet<T>`, restituiscono dei riferimenti. Quindi è possibile modificare gli elementi della collezione tramite metodi che modifichino il risultato di `i.next()`. Ciò è illustrato nel Listato 16.5. I commenti fatti a proposito di `i.next()` si applicano anche a `i.previous()`.

Il fatto che `next` e `previous` restituiscano dei riferimenti non costituisce necessariamente un problema. Significa che è necessario fare attenzione, ma anche che è possibile scorrere gli elementi della collezione ed effettuare delle operazioni che modifichino gli elementi. Per esempio, se gli elementi della collezione contengono registrazioni di qualche tipo, si possono usare metodi di modifica degli elementi per aggiornare le registrazioni.

Analizzando le API delle interfacce `Iterator<T>` e `ListIterator<T>`, si trova scritto che un `ListIterator<T>` può modificare la collezione, mentre probabilmente un semplice `Iterator<T>` non può farlo. Questi commenti non si riferiscono al fatto che `i.next()` restituisca o meno un riferimento, ma semplicemente al fatto che un `ListIterator<T>` ha un metodo `set` che manca all'interfaccia `Iterator<T>`. Non si confonda questo aspetto con la questione discussa qui.

LISTATO 16.5 Un iteratore restituisce un riferimento. MyLab

```
import java.util.ArrayList;
import java.util.Iterator;

public class RiferimentiIteratoriDemo {
   public static void main(String[] args) {
        ArrayList<Animale> cani = new ArrayList<Animale>();

        cani.add(new Animale("Fido", 3, 9));
        cani.add(new Animale("Fuffy", 4, 8));
        cani.add(new Animale("Pluto", 2, 7));

        System.out.println("La lista contiene:");

        Iterator<Animale> i = cani.iterator();
        while (i.hasNext())
              i.next().scriviOutput();

        i = cani.iterator();
        Animale c = null; //Per non avere problemi di compilazione.
        System.out.println("Modifica dei riferimenti.");

        while (i.hasNext()) {
              c = i.next();
              c.setPeso(10);
        }

        System.out.println("La lista ora contiene:");
        i = cani.iterator();
        while (i.hasNext())
              i.next().scriviOutput();
   }
}
```

Esempio di output

```
La lista contiene:
Nome: Fido
Eta: 3 anni
Peso: 9.0 Kg
```

```
Nome: Fuffy
Eta: 4 anni
Peso: 8.0 Kg
Nome: Pluto
Eta: 2 anni
Peso: 7.0 Kg
Modifica dei riferimenti.
La lista ora contiene:
Nome: Fido
Eta: 3 anni
Peso: 10.0 Kg
Nome: Fuffy
Eta: 4 anni
Peso: 10.0 Kg
Nome: Pluto
Eta: 2 anni
Peso: 10.0 Kg
```

Definire nuove classi di tipo iteratore

Raramente è necessario definire nuove classi che implementino `Iterator<T>` o `ListIterator<T>`. Il modo più comune e più semplice per definire una nuova collezione è estendendo una delle classi predefinite, come `ArrayList<T>` o `HashSet<T>`. Facendo questo, si ottengono automaticamente il metodo `iterator()` ed eventualmente il metodo `listIterator()`, cosa che risolve il problema della gestione degli iteratori. Tuttavia, se dovesse essere necessario definire una classe di tipo collezione in qualche altro modo, il modo migliore di definire la classe (o le classi) per gli iteratori è definirla come *inner class* della collezione.

16.4 Riepilogo

- Le interfacce principali di tipo collezione sono tre: `Collection<T>`, `Set<T>` e `List<T>`. Le interfacce `Set<T>` e `List<T>` estendono `Collection<T>`. Le classi di libreria solitamente impiegate per usare e implementare queste interfacce sono `HashSet<T>`, che implementa `Set<T>`, e `ArrayList<T>`, che implementa `List<T>`.
- Un `Set<T>` non ammette elementi ripetuti e non ordina i propri elementi. Una `List<T>` ammette elementi ripetuti e ordina i propri elementi.
- L'interfaccia `Map<K, V>` è usata per immagazzinare le associazioni tra una chiave `K` e un valore `V`. Di solito è usata per mantenere in memoria basi di dati. La classe `HashMap<K, V>` è una classe standard di libreria che implementa una mappa.

- Un iteratore è qualcosa che consente di esaminare ed eventualmente modificare gli elementi di una collezione in un qualche ordine sequenziale. Java formalizza questo concetto con due interfacce: `Iterator<T>` e `ListIterator<T>`.
- Un `Iterator<T>` scandisce gli elementi della collezione in una sola direzione, dall'inizio alla fine. Un `ListIterator<T>` può muoversi all'interno della lista in entrambe le direzioni: avanti e indietro.

16.5 Esercizi

1. Riscrivere il Progetto di programmazione 4 del Capitolo 6, questa volta per ordinare in ordine alfabetico un vettore di stringhe.
2. Il Crivello di Eratostene è un antico algoritmo per generare i numeri primi. Si consideri la lista dei numeri da 2 a 10:

   ```
   2 3 4 5 6 7 8 9 10
   ```

   L'algoritmo parte dal primo numero primo della lista, 2, e scorre la lista eliminando tutti i suoi multipli (in questo caso 4, 6, 8 e 10), così che la lista diventi:

   ```
   2 3 5 7 9
   ```

   Si ripete il procedimento con il secondo numero primo della lista, 3, eliminando dalla lista tutti i suoi multipli rimasti (in questo caso, solo il numero 9). Si ottiene:

   ```
   2 3 5 7
   ```

   Si ripete nuovamente il procedimento partendo dal numero primo successivo, ma in questo caso non vengono eliminati altri elementi, perché non ci sono multipli di 5 e di 7 (un'implementazione più efficiente dell'algoritmo terminerebbe senza dover esaminare 5 e 7). I numeri rimasti sono tutti i numeri primi della lista originale.

   Si implementi questo algoritmo usando un `ArrayList` di `Integer` inizializzato con i valori da 2 a 100. Il programma può scorrere l'`ArrayList` numericamente dall'indice 0 all'indice `size()-1` per estrarre il numero primo corrente, ma dovrebbe usare un `Iterator` per analizzare la parte restante della lista ed eliminare i multipli. Si può usare il metodo `listIterator` per ottenere l'iteratore che parte all'indice specificato dell'`ArrayList`. Si mostrino a video tutti i numeri primi rimasti.

16.6 Progetti

1. Il paradosso dei compleanni consiste nel fatto che esiste una probabilità sorprendentemente elevata che due o più persone nella stessa stanza abbiano lo stesso compleanno, nel senso che lo festeggiano lo stesso giorno (trascurando gli anni bisestili), ignorando quindi l'anno e l'ora di nascita. Si scriva un programma che approssimi la probabilità che due o più persone nella stessa stanza abbia lo stesso compleanno, per un numero di persone nella stanza da 2 a 50.

Il programma dovrebbe utilizzare una simulazione per approssimare la risposta. Eseguendo molte prove (per esempio, 5000), si assegnino i compleanni (cioè numeri tra 1 e 365) in modo casuale a ogni persona nella stanza. Si usi un `HashSet<T>` per immagazzinare i compleanni. Ogni volta che viene generato in modo casuale un compleanno, si usi il metodo `contains` dell'`HashSet` per verificare se c'è già qualcuno nella stanza con quel compleanno. In tal caso, si incrementi un contatore per tenere traccia del numero di casi in cui due o più persone hanno lo stesso compleanno e poi si passi alla prova successiva. Alla fine delle prove, si divida il contatore per il numero di prove, per ottenere una stima della probabilità che due o più persone abbiano lo stesso compleanno, fissato il numero di persone nella stanza.

L'output dovrebbe essere simile a quello riportato di seguito. Non sarà esattamente lo stesso a causa della generazione casuale dei numeri:

```
Per 2 persone, la probabilità che due compleanni coincidano è 0.002
Per 3 persone, la probabilità che due compleanni coincidano è 0.0082
Per 4 persone, la probabilità che due compleanni coincidano è 0.0163
...
Per 49 persone, la probabilità che due compleanni coincidano è 0.9654
Per 50 persone, la probabilità che due compleanni coincidano è 0.969
```

2. In un antico regno, la bella principessa Eva aveva molti pretendenti. Stabilì una procedura per decidere quale avrebbe sposato. Per prima cosa, i pretendenti sarebbero stati allineati e a ciascuno sarebbe stato assegnato un numero. Il primo pretendente avrebbe ricevuto il numero 1, il secondo il 2 e così via fino all'ultimo, con il numero *n*. Partendo dal primo, avrebbe contato tre (come il numero delle lettere del suo nome) pretendenti lungo la fila e il terzo pretendente sarebbe stato eliminato e tolto dalla fila. Eva avrebbe quindi continuato contando ogni volta tre pretendenti ed eliminandone uno ogni tre. Una volta arrivata alla fine della lista, avrebbe invertito la direzione e continuato tornando all'inizio. Allo stesso modo, una volta raggiunta la prima persona della lista, avrebbe invertito nuovamente la direzione.

   Per esempio, se ci fossero stati cinque pretendenti, il processo di eliminazione si sarebbe svolto come segue:

```
12345  lista iniziale dei pretendenti, si inizia a contare da 1
1245   pretendente numero 3 eliminato, si continua dal numero 4,
       rimbalzando alla fine della lista di nuovo sul numero 4
125    pretendente numero 4 eliminato, si riparte dal 2, rimbalzando
       all'inizio della lista di nuovo sul numero 2
15     pretendente numero 2 eliminato, si continua a contare dal 5
1      pretendente numero 5 eliminato, il numero 1 è il fortunato
       vincitore
```

   Scrivere un programma che usi un `ArrayList` o un `Vector` per determinare in quale posizione ci si dovrebbe trovare per sposare la principessa se ci sono *n* pretendenti. Il programma dovrebbe utilizzare l'interfaccia `ListIterator` per scorrere la lista di pretendenti ed eliminarli. Si verifichi con attenzione che l'iteratore punti all'oggetto giusto quando si inverte la direzione all'inizio o alla fine della lista. Il pretendente all'inizio o alla fine della lista va contato solo una volta in corrispondenza dell'inversione.

3. *Sui siti di "social networking" le persone si collegano ai propri amici per formare una rete di contatti. Scrivere un programma che usi delle* `HashMap` *per immagazzinare i dati di una rete di questo tipo. Il programma dovrebbe leggere un file che specifica le connessioni della rete per diversi nomi utente. Il file dovrebbe avere il seguente formato per la specifica di un collegamento:*

```
nomeutente_sorgente nomeutente_amico
```

Ci deve essere una voce per collegamento, una per riga. Questo è un esempio nel caso di cinque utenti:

```
iba         java_guru
iba         crisha
iba         ducky
crisha      java_guru
crisha      iba
ducky       java_guru
ducky       iba
java_guru   iba
java_guru   crisha
java_guru   ducky
wittless    java_guru
```

In questa rete, tutti hanno `java_guru` come amico. `iba` è amico di `java_guru`, `crisha` e `ducky`. Si noti che i collegamenti non sono bidirezionali: `wittless` è collegato a `java_guru`, ma `java_guru` non è collegato a `wittless`.

Per prima cosa si crei una classe `Utente` con una variabile di istanza per contenere il nome dell'utente e una di tipo `HashSet<Utente>`. La variabile `HashSet<Utente>` conterrà i riferimenti agli oggetti `Utente` ai quali l'utente corrente è collegato. Per esempio, per l'utente `iba` ci sarebbero tre elementi, uno per `java_guru`, uno per `crisha` e uno per `ducky`. Si crei poi nella classe principale una variabile di istanza di tipo `HashMap<String, Utente>` usata per associare un nome utente all'oggetto `Utente` corrispondente. Il programma dovrebbe:

- leggere il file all'avvio e popolare le strutture dati di tipo `HashMap` e `HashSet` in base ai collegamenti specificati nel file;
- consentire all'utente di inserire un nome;
- se il nome è presente nella mappa, stampare tutti i nomi utente collegati direttamente a quello specificato dall'utente;
- se il nome è presente nella mappa, stampare tutti i nomi utente a distanza di due collegamenti da quello specificato dall'utente. Per fare questo in generale si può considerare l'uso di una procedura ricorsiva.

Non si dimentichi che la classe `User` deve ridefinire i metodi `hashCode` ed `equals`.

4. È stato preparato un file di giudizi su vari film, nel quale a ogni film è stato assegnato un giudizio compreso tra 1 (brutto) e 5 (eccellente). La prima riga del file contiene un numero che identifica il numero di giudizi nel file. Ogni giudizio consiste poi di due righe: il nome del film seguito dal giudizio numerico tra 1 e 5. Questo è un file di esempio con quattro film distinti e sette giudizi:

```
7
Harry Potter and the Half-Blood Prince
4
Harry Potter and the Half-Blood Prince
5
Army of the Dead
1
Harry Potter and the Half-Blood Prince
4
Army of the Dead
2
The Uninvited
4
Pandorium
3
```

Scrivere un programma che legga un file in questo formato, calcoli il giudizio medio per ogni film e stampi la media insieme al numero di recensioni. Questo è l'output atteso per i dati di esempio:

```
Harry Potter and the Half-Blood Prince: 3 recensioni, media 4.3 / 5
Army of the Dead: 2 recensioni, media 1.5 / 5
The Uninvited: 1 recensione, media 4 / 5
Pandorium: 1 recensione, media 3 / 5
```

Usare una o più `HashMap` per calcolare il risultato. Le mappe dovrebbero associare le stringhe che identificano ciascun film agli interi che rappresentano il numero di recensioni per il film e la somma dei giudizi.

Capitolo 17

Interfacce utente grafiche

OBIETTIVI

- Descrivere i principi di base della programmazione a eventi.
- Progettare e implementare semplici interfacce utente grafiche che includano pulsanti e testo.
- Utilizzare classi standard appartenenti alle librerie grafiche di Java.

Finora, quasi tutti i programmi presentati hanno utilizzato la forma più semplice di input, nella quale l'utente inserisce del testo semplice da tastiera e del testo semplice viene mostrato come output sullo schermo. I programmi moderni, tuttavia, non utilizzano queste forme così semplici di input e output.

In realtà, i programmi moderni utilizzano interfacce a finestre con caratteristiche, come i menu e i pulsanti, che consentono all'utente di svolgere operazioni mediante il mouse o un altro dispositivo di puntamento. In questo capitolo si mostrerà come scrivere programmi Java che sfruttino le interfacce grafiche per le operazioni di input e di output. Si utilizzerà la libreria standard di classi Java denominata **Swing**. A questa libreria sono stati dedicati interi libri e questo capitolo non può certo essere sufficiente a fornirne una descrizione completa. Tuttavia, verrà presentato quanto basta per poter costruire semplici interfacce a finestre.

Esiste anche un'altra (più vecchia) libreria di classi per la costruzione di interfacce a finestre, nota come **Abstract Window Toolkit** o **AWT**. La libreria Swing può essere vista come una versione migliorata della AWT. Tuttavia, Swing non sostituisce completamente AWT, ma si aggiunge a essa fornendo una collezione più ricca di classi e AWT resta un complemento necessario alla libreria Swing. In questo capitolo verranno utilizzate classi sia da Swing che da AWT.

Prerequisiti

Prima di leggere questo capitolo occorre aver compreso i concetti di base presentati nei Capitoli da 1 a 4 e da 8 a 9, aver imparato le basi dell'ereditarietà presenti nel Capitolo 10 e avere dimestichezza con le interfacce presentate nel Capitolo 11. Infine, l'ultima sezione del capitolo richiede anche la conoscenza delle eccezioni trattate nel Capitolo 13.

17.1 Introduzione

Per prima cosa verranno discusse le caratteristiche di base degli elementi di un'interfaccia a finestre e della tecnica di programmazione detta programmazione a eventi.

17.1.1 Interfacce utente grafiche

Le interfacce a finestre con le quali un utente può interagire sono dette anche **interfacce utente grafiche** o, in inglese, ***graphical user interface*** **(GUI)**. Questa denominazione è autoesplicativa: l'espressione *interfaccia utente* indica che si tratta della parte di un programma che si interfaccia (cioè che interagisce) con l'utente. Il termine *grafica* specifica che l'interfaccia è basata su elementi grafici come finestre, menu e pulsanti. Una GUI ottiene informazioni dall'utente e le passa al programma affinché vengano elaborate. Una volta terminata l'elaborazione, la GUI comunica il risultato all'utente, in genere utilizzando delle finestre.

È utile passare rapidamente in rassegna i termini utilizzati per indicare alcuni degli elementi di base che compongono una GUI. Infatti, anche se tali elementi sono di uso molto comune, è possibile che non li si sia mai sentiti indicare con i termini che saranno utilizzati in questo capitolo. Una **finestra** (***window***) è una porzione dello schermo dell'utente che si comporta a sua volta come un piccolo schermo. Generalmente, una finestra ha un bordo che ne delimita i confini e un titolo. All'interno di una finestra saranno presenti altri oggetti, per esempio dei menu. Un **menu** è una lista di alternative offerte all'utente, che sceglie tra di esse, tipicamente utilizzando un pulsante del mouse quando il cursore si trova sopra l'elemento desiderato del menu (operazione che viene spesso menzionata come "cliccare una voce di menu"). Un **pulsante**, o **bottone** (***button***), assomiglia a un pulsante reale e normalmente è caratterizzato da un'etichetta (*label*). Per "premere" il pulsante si fa un click su di esso utilizzando il mouse.

GUI

I sistemi a finestre che interagiscono con l'utente sono spesso chiamati interfacce utente grafiche. In genere, queste interfacce sono indicate semplicemente tramite l'acronimo inglese GUI, che sta per *graphical user interface*.

17.1.2 Programmazione a eventi

Nell'ambito delle interfacce utente grafiche, con il termine **evento** (***event***) si indica un oggetto che rappresenta un'azione, come la pressione di un pulsante, il trascinamento del cursore del mouse, la pressione di un tasto sulla tastiera, la chiusura di una finestra o qualunque altra azione che ci si aspetta scateni una risposta. In altri contesti, il termine evento può avere un significato più generale. Per esempio, il messaggio con il quale una stampante comunica al sistema operativo di essere pronta per stampare un nuovo documento può essere considerato un evento. In questo capitolo, tuttavia, si considereranno solo eventi generati all'interno delle interfacce utente grafiche.

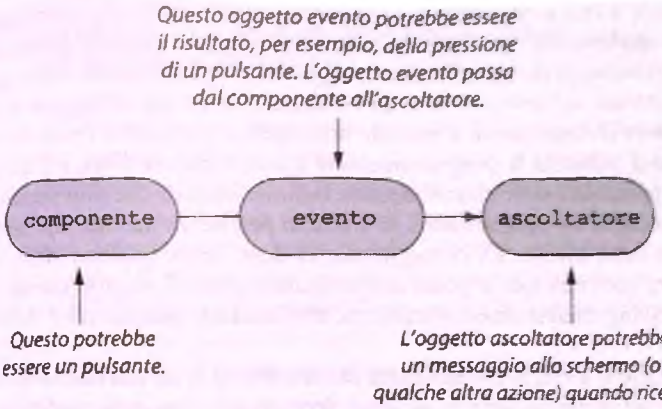


Figura 17.1 Generazione e ascolto degli eventi.

Quando un oggetto genera un evento, si dice anche che lo **scatena** o che lo **emette**. Nella libreria Swing, qualunque oggetto che possa generare eventi, come per esempio un pulsante che possa essere premuto, può avere uno o più **oggetti ascoltatori** (***listener***). Il programmatore specifica gli ascoltatori per ogni oggetto che può generare eventi. Per esempio, se si preme un pulsante che genera un evento e se il pulsante ha un ascoltatore associato, l'evento viene inviato automaticamente all'oggetto ascoltatore. Un ascoltatore ha metodi che specificano cosa accade quando vengono ricevuti eventi di vario tipo. I metodi che gestiscono gli eventi sono detti **gestori di eventi** (***event handlers***). Il programmatore definirà (o ridefinirà) questi metodi di gestione. La relazione tra un oggetto che genera eventi e il corrispondente ascoltatore che li gestisce è illustrata nella Figura 17.1.

La maggior parte dei programmi visti finora era costituita da una lista di istruzioni eseguite in un qualche ordine. Potevano comparire variazioni sul tema, come cicli che ripetessero più volte un insieme di istruzioni, espressioni condizionali che selezionassero le istruzioni da eseguire tra più possibilità o chiamate a metodo che portassero in esecuzione una diversa sequenza di istruzioni. In ogni caso tutti quei programmi erano progettati per essere eseguiti da un singolo agente (il computer) che seguisse un insieme semplice di istruzioni del tipo "per prima cosa fai questo, poi fai quest'altro, poi fai qualcos'altro ancora" e così via.

I programmi basati su Swing, e in genere tutte le GUI, utilizzano invece eventi e gestori di eventi. La programmazione a eventi è piuttosto diversa da quella vista finora. Nella programmazione a eventi si creano oggetti che possono generare eventi e se ne creano altri che reagiscono a essi. Nella maggior parte dei casi, il programmatore non sa in che ordine verranno eseguite le varie parti. L'ordine sarà determinato dagli eventi. Durante l'esecuzione di un programma basato su eventi, ciò che il programma eseguirà dipende da quali eventi verranno generati.

Gli oggetti ascoltatori sono simili a persone che aspettano chiamate sui propri telefoni cellulari. Quando suona un telefono, il proprietario risponde e reagisce in modo appropriato. Un messaggio potrebbe essere "chiudi la finestra nella tua stanza". La perso-

na quindi tornerebbe a casa e chiuderebbe la finestra. In un'interfaccia utente grafica, il messaggio sarebbe qualcosa del tipo "chiudi la finestra", "è stato premuto il tasto Invio", o "il mouse è stato trascinato da qui a lì". Il messaggio è l'evento. Quando viene generato un evento, esso è inviato automaticamente all'ascoltatore associato all'oggetto che lo ha generato. L'ascoltatore chiama quindi il metodo appropriato per gestire l'evento.

Se non si è mai utilizzata la programmazione a eventi prima d'ora, c'è un aspetto che può sembrare particolarmente strano: occorre definire metodi che non verranno mai invocati esplicitamente in un programma. Ciò è strano perché un metodo che non venga mai invocato è del tutto inutile. Di conseguenza, ci deve essere qualcun altro, o qualcos'altro, che invoca questi metodi al posto del programmatore. È proprio questo ciò che accade. Il sistema Swing chiama automaticamente certi metodi quando ciò è richiesto da un evento.

La programmazione a eventi che utilizza la libreria Swing fa un uso estensivo dell'ereditarietà. Le classi che verranno definite saranno derivate da classi base predefinite della libreria. Di conseguenza, le classi da definire erediteranno i metodi dalle rispettive classi base. Alcuni dei metodi ereditati funzioneranno come desiderato così come sono. Altri dovranno spesso essere ridefiniti in modo appropriato.

17.2 Caratteristiche di base della libreria Swing

Come accennato precedentemente, una finestra è una porzione dello schermo dell'utente che si comporta come un schermo più piccolo all'interno dello schermo dell'utente. In questo paragrafo si spiegherà come creare semplici finestre utilizzando la libreria Swing.

Esistono molti elementi che possono essere inseriti in una finestra. Qui si comincerà presentandone alcuni molto semplici ma molto utili. Tali elementi consentiranno di chiudere la finestra, inserirvi del testo, colorarla e assegnarle un titolo. Queste operazioni potrebbero sembrare limitate, ma costituiranno una buona introduzione alla programmazione con Swing.

Questo paragrafo presenterà un programma di esempio, del quale verranno fornite due versioni. La prima versione sarà deliberatamente molto semplificata per consentire di illustrare meglio alcuni dettagli di Swing. Alla fine del paragrafo, il programma verrà riscritto nel modo più corretto.

Salvare tutto prima di eseguire un programma Swing

I programmi che utilizzano la libreria Swing possono prendere il controllo dello schermo, del mouse e della tastiera. Se qualcosa va storto (e spesso succede), i tradizionali metodi per impartire comandi al computer potrebbero non funzionare. Ciò richiederebbe di riavviare il sistema. Pertanto, prima di eseguire un programma Swing che non sia stato ancora adeguatamente verificato, è consigliabile salvare e chiudere i file aperti, per evitare possibili perdite di dati in caso sia necessario un riavvio.

ESEMPIO DI PROGRAMMAZIONE
UNA SEMPLICE FINESTRA

Il Listato 17.1 contiene un programma Java che utilizza la libreria Swing per costruire una semplice finestra. Dopo il codice viene riportata un'immagine di come apparirà lo schermo quando viene eseguito il programma (l'aspetto potrebbe variare leggermente a seconda del sistema sul quale si esegue il programma). La finestra di questo esempio non fa molto. Semplicemente compare sullo schermo e mostra il testo `"Non premere quel pulsante!"`. L'unica altra cosa che può fare è scomparire. Se si preme il pulsante di chiusura della finestra, il programma terminerà e la finestra scomparirà. Ora il codice dell'esempio verrà analizzato in dettaglio.

La prima riga indica che il programma utilizza la classe `JFrame` della libreria Swing:

```
import javax.swing.JFrame;
```

Il resto del programma è una semplice definizione di classe che contiene solo un metodo `main`. La prima riga di questo metodo crea un'istanza della classe `JFrame`:

```
JFrame finestra = new JFrame();
```

Il nome `finestra` è scelto dal programmatore. Un oggetto `JFrame` è ciò che si intende comunemente come finestra. Si tratta di una finestra molto semplice, ma possiede comunque, tra le altre cose, un bordo, uno spazio per il titolo (anche se in questo esempio non viene utilizzato) e il pulsante di chiusura che ci si aspetta di trovare in ogni finestra. In seguito si vedrà che normalmente non si utilizzano direttamente oggetti della classe `JFrame`, ma oggetti di classi da essa derivate. Tuttavia, per questo esempio si utilizzerà direttamente `JFrame`.

La riga successiva,

```
finestra.setSize(LARGHEZZA, ALTEZZA);
```

imposta le dimensioni della finestra sfruttando il metodo `setSize` della classe `JFrame`. La finestra sarà larga `LARGHEZZA` unità e alta `ALTEZZA` unità. Il significato delle unità utilizzate da Swing per le dimensioni verrà spiegato più avanti; per ora non è necessario preoccuparsene.

La riga successiva crea un oggetto della classe `JLabel` chiamandolo `etichetta`:

```
JLabel etichetta = new JLabel("Non premere quel pulsante!");
```

Anche in questo caso il nome etichetta può essere scelto liberamente. Un oggetto della classe `JLabel` è normalmente detto semplicemente **etichetta** (***label***) e costituisce un tipo particolare di testo che può essere aggiunto a un `JFrame` (e ad altri tipi di oggetto). La stringa da visualizzare nell'etichetta è passata come parametro al costruttore della classe `JLabel`.

La riga successiva,

```
finestra.add(etichetta);
```

aggiunge l'etichetta alla finestra.

La finestra può essere chiusa premendo il pulsante di chiusura. Quando si preme il pulsante, la finestra genera un evento e lo invia a un ascoltatore che chiude la finestra. In questo programma, l'oggetto ascoltatore è chiamato `ascoltatore` ed è una istanza della classe `DistruttoreFinestra`. La riga che segue crea un nuovo oggetto della classe `DistruttoreFinestra` chiamandolo `ascoltatore`:

```
DistruttoreFinestra ascoltatore = new DistruttoreFinestra();
```

La riga successiva,

```
finestra.addWindowListener(ascoltatore);
```

associa l'oggetto `ascoltatore` alla finestra, in modo che `ascoltatore` riceva tutti gli eventi generati dalla finestra. Un oggetto ascoltatore che riceve gli eventi, come quello della pressione del pulsante di chiusura, da una finestra è detto **ascoltatore di finestre** (***window listener***). L'operazione che associa un ascoltatore con un oggetto che genera eventi è detta **registrazione dell'ascoltatore**.

Un oggetto della classe `DistruttoreFinestra` chiuderà la finestra quando questa avrà generato un evento appropriato. La classe `DistruttoreFinestra` verrà discussa più avanti. Per ora, si supponga che questa classe sia stata definita in modo che quando l'utente preme il pulsante di chiusura, l'oggetto `ascoltatore` chiuda la finestra e termini il programma.

L'ultima istruzione,

```
finestra.setVisible(true);
```

è una chiamata al metodo `setVisible` che rende visibile la finestra sullo schermo. Il metodo `setVisible` fa parte della classe `JFrame` (e anche di molte altre classi nella libreria Swing). Quando il metodo viene invocato con il parametro `true`, come nel Listato 17.1, l'oggetto in questione viene mostrato. Se si utilizza invece il parametro `false`, l'oggetto viene nascosto.

A questo punto è stata raggiunta la fine del metodo `main`, ma non la fine dell'esecuzione del programma. La finestra se ne sta semplicemente ferma sullo schermo finché l'utente non preme il pulsante di chiusura. A quel punto, l'oggetto `finestra` genera un evento che viene inviato all'oggetto `ascoltatore`. Questo oggetto riconosce l'evento come un'indicazione di chiudere la finestra e terminare l'esecuzione del programma. La Figura 17.2 illustra la gestione degli eventi in questo caso.

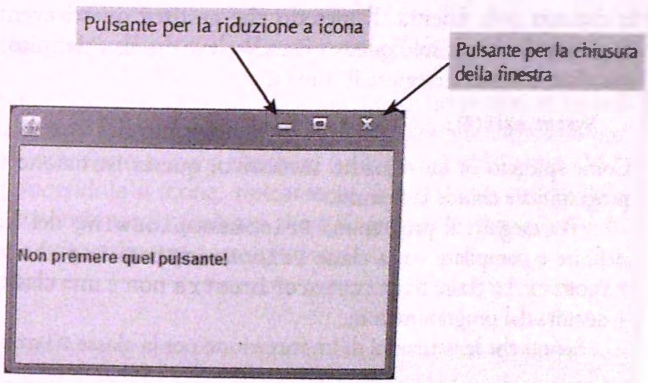
LISTATO 17.1 Un esempio molto semplice di utilizzo di Swing.

Questo è un semplice programma di esempio e non utilizza lo stile più corretto per i programmi Swing.

```
import javax.swing.JFrame;
import javax.swing.JLabel;

/**
 * Un esempio di semplice finestra costruita utilizzando Swing.
 */
public class PrimoEsempioSwing {
    public static final int LARGHEZZA = 300;
    public static final int ALTEZZA = 200;

    public static void main(String args[]) {
        JFrame finestra = new JFrame();
```

```
        finestra.setSize(LARGHEZZA, ALTEZZA);
        JLabel etichetta = new JLabel("Non premere quel pulsante!");

        finestra.add(etichetta);

        DistruttoreFinestra ascoltatore = new DistruttoreFinestra();
        finestra.addWindowListener(ascoltatore);

        finestra.setVisible(true);
    }
}
```

Output sullo schermo



L'oggetto ascoltatore è una istanza della classe `DistruttoreFinestra`. Finora questa classe è stata considerata come già definita, ma in realtà occorre definirla e compilarla prima di poter eseguire il programma del Listato 17.1.

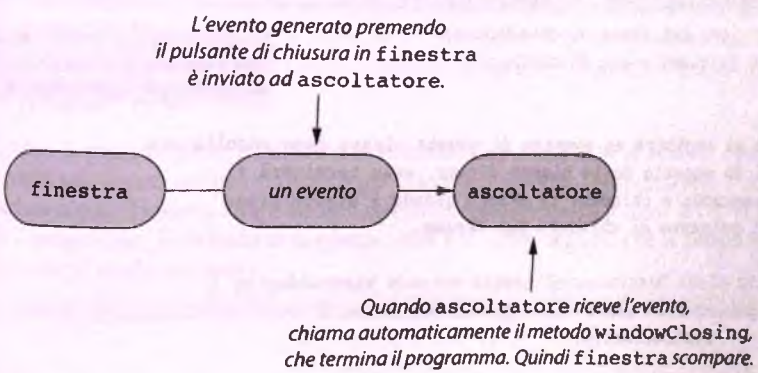


Figura 17.2 Pressione del pulsante di chiusura della finestra.

La classe ascoltatore `DistruttoreFinestra` è definita nel Listato 17.2. Una classe ascoltatore per una GUI che coinvolga delle finestre (o, più precisamente, che coinvolga oggetti della classe `JFrame`) sarà spesso una classe derivata dalla classe `WindowAdapter`, come indicato dalla specifica `extends WindowAdapter` nella prima riga della definizione della classe. In quanto derivata da `WindowAdapter`, la classe `DistruttoreFinestra` eredita tutti i metodi di `WindowAdapter`. Ogni metodo reagisce automaticamente a un tipo diverso di evento. Normalmente non vengono aggiunti nuovi metodi, dato che esistono già metodi per tutti i tipi di evento. Tuttavia, il modo in cui un evento è gestito deve essere stabilito dal programmatore e dipenderà dalla finestra che si sta costruendo. Di solito è necessario ridefinire uno o più metodi ereditati da `WindowAdapter`.

Occorre ridefinire solo i metodi che saranno effettivamente utilizzati. In questo esempio l'unico tipo di evento al quale l'ascoltatore dovrà rispondere è quello che segnala la chiusura della finestra. Il metodo che gestisce questi eventi è chiamato `windowClosing`. Pertanto, solo questo metodo deve essere ridefinito. La definizione è molto semplice: si limita a eseguire il comando

```
System.exit(0);
```

Come spiegato in un riquadro successivo, questa istruzione termina l'esecuzione del programma e chiude la finestra.

Per eseguire il programma `PrimoEsempioSwing` del Listato 17.1, è necessario definire e compilare sia la classe `PrimoEsempioSwing` che la classe `DistruttoreFinestra`. La classe `DistruttoreFinestra` non è una classe della libreria Swing, ma è definita dal programmatore.

Si noti che le istruzioni di importazione per la classe `DistruttoreFinestra` sono

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
```

Queste istruzioni dicono semplicemente al compilatore dove trovare le definizioni della classi `WindowAdapter` e `WindowEvent`. Si ricordi che la libreria AWT è precedente alla Swing. Le classi `WindowAdapter` e `WindowEvent` fanno parte della libreria AWT.

Lab

LISTATO 17.2 Una classe per la gestione di eventi generati da finestre.

La classe `DistruttoreFinestra` non è predefinita in Java. Deve essere definita dal programmatore.

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

/**
 * Se si registra un oggetto di questa classe come ascoltatore
 * di un oggetto della classe JFrame, esso terminerà il
 * programma e chiuderà il JFrame quando l'utente preme
 * il pulsante di chiusura del JFrame.
 */
public class DistruttoreFinestra extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
}
```

La classe `JLabel`

Un oggetto della classe `JLabel` contiene una riga di testo che può essere aggiunta a un oggetto `JFrame` (o a certi altri tipo di oggetto).

Esempio

```
JFrame finestra = new JFrame();
JLabel etichetta = new JLabel("Non premere quel pulsante!");
finestra.add(etichetta);
```

FAQ Come funzionano gli altri pulsanti predefiniti in un `JFrame`?

Il pulsante di chiusura della finestra in un `JFrame` fa parte di un gruppo di tre pulsanti. Gli altri due funzionano già in modo adeguato senza che sia necessario programmarli. Il pulsante di riduzione a icona mostrato nell'immagine del Listato 17.1, minimizza la finestra riducendola a icona, tipicamente in basso o su un lato dello schermo. L'altro pulsante consente di cambiare alternativamente le dimensioni della finestra da una versione a tutto schermo a una più piccola.

Terminare l'esecuzione di un programma Swing

Un programma con interfaccia utente grafica si basa, normalmente, su un qualche tipo di ciclo infinito. Il programma potrebbe non contenere esplicitamente un ciclo Java, ma in ogni caso potrebbe non terminare mai. L'interfaccia utente grafica rimane visibile sullo schermo e attiva finché l'utente non richiede che venga rimossa, per esempio premendo il pulsante di chiusura della finestra. Se l'utente non richiede mai la chiusura del programma, questo non terminerà mai. Di conseguenza, quando si scrive un programma con interfaccia utente grafica utilizzando la libreria Swing, occorre un modo per terminarlo. Il metodo `exit` terminerà l'esecuzione di un programma non appena verrà eseguito. Nel Listato 17.2 viene utilizzata la seguente chiamata a questo metodo:

```
System.exit(0);
```

Il valore `0`, passato come parametro al metodo, viene restituito al sistema operativo. In molti casi si può utilizzare qualunque valore senza che il comportamento del programma cambi. Tuttavia, molti sistemi operativi considerano lo `0` come un'indicazione che il programma è terminato correttamente e `1` come segnale che il programma è terminato in modo anomalo.

Dimenticarsi di programmare il pulsante di chiusura di una finestra

Le righe seguenti, tratte dal Listato 17.1, garantiscono che quando l'utente preme il pulsante di chiusura della finestra il programma termini e la finestra venga chiusa:

```
DistruttoreFinestra ascoltatore = new DistruttoreFinestra();
finestra.addWindowListener(ascoltatore);
```

Non è necessario utilizzare esattamente queste istruzioni per programmare l'azione del pulsante di chiusura di una finestra. Per esempio, si potrebbe definire un'altra classe al posto di `DistruttoreFinestra`. Tuttavia, occorre comunque prevedere delle istruzioni per garantire che, quando l'utente preme il pulsante di chiusura, l'interfaccia utente grafica si comporti come ci si aspetta. Se non lo si fa, la finestra scomparirà comunque quando l'utente preme il pulsante di chiusura, ma l'esecuzione del programma non terminerà. Se l'interfaccia utente grafica è costituita da una sola finestra, ciò significa che non si avrà più a disposizione un metodo semplice per concludere l'esecuzione del programma. Il fatto che la finestra scompaia in ogni caso contribuisce a creare confusione, dato che porta a credere che il programma sia terminato mentre è ancora in esecuzione.

17.2.1 Ulteriori dettagli sui *window listener*

Come nel caso della classe `DistruttoreFinestra` del Listato 17.2, le classi *window listener* sono derivate, solitamente, dalla classe `WindowAdapter`. Questa classe ha molti metodi, ognuno dei quali è invocato automaticamente quando l'oggetto ascoltatore riceve un evento compatibile con quel metodo. Questi metodi e gli eventi corrispondenti sono riportati nella tabella della Figura 17.3. Quando si definisce una classe derivata dalla classe `WindowAdapter`, è necessario ridefinire solo i metodi effettivamente da personalizzare. Dato che l'unico compito della classe `DistruttoreFinestra` è chiudere le finestre, nella definizione di `DistruttoreFinestra` è stato necessario ridefinire solo il metodo `windowClosing`.

La classe `WindowAdapter` è astratta, quindi non è possibile crearne istanze utilizzando un'istruzione come `new WindowAdapter()`. La classe `WindowAdapter` può essere utilizzata solo come classe base per definirne altre.

17.2.2 Unità di misura per le dimensioni degli oggetti sullo schermo

Nella libreria Swing, le dimensioni di un oggetto visualizzato sullo schermo sono espresse in **pixel**. Un pixel è la più piccola unità di spazio che può essere gestita indipendentemente su uno schermo. Si può pensare a un pixel come a un piccolo rettangolo che può assumere solo uno tra un ristretto gruppo di colori prefissati e allo schermo come a un mosaico composto da questi piccoli rettangoli. Potrebbe essere utile pensare in termini di un semplice schermo in bianco e nero nel quale ogni pixel è bianco o nero. Quanto

`public void windowOpened(WindowEvent e)` È invocato quando una finestra è stata aperta.
`public void windowClosing(WindowEvent e)` È invocato quando una finestra sta per essere chiusa. La pressione del pulsante di chiusura provoca un'invocazione di questo metodo.
`public void windowClosed(WindowEvent e)` È invocato quando una finestra è stata chiusa.
`public void windowIconified(WindowEvent e)` È invocato quando una finestra viene ridotta ad icona, ad esempio quando viene premuto il pulsante che minimizza un `JFrame`.
`public void windowDeiconified(WindowEvent e)` È invocato quando una finestra ridotta ad icona viene ingrandita nuovamente o viene resa attiva.
`public void windowActivated(WindowEvent e)` È invocato quando una finestra viene resa attiva, ad esempio cliccando su di essa con il mouse, o tramite altre operazioni.
`public void windowDeactivated(WindowEvent e)` È invocato quando una finestra viene resa non attiva. Quando una finestra viene resa attiva, tutte le altre diventano non attive. Esistono anche altre azioni che possono rendere non attiva una finestra.
`public void windowGainedFocus(WindowEvent e)` È invocato quando una finestra acquisisce il focus (il focus non verrà trattato in questo testo).
`public void windowLostFocus(WindowEvent e)` È invocato quando una finestra perde il focus (il focus non verrà trattato in questo testo).
`public void windowStateChanged(WindowEvent e)` È invocato quando una finestra cambia stato.

Figura 17.3 Metodi della classe `WindowAdapter`.

più numerosi sono i pixel che compongono lo schermo, tanto più alta è la **risoluzione** di quest'ultimo. Quindi, maggiore è il numero di pixel, più piccoli saranno i dettagli che si possono distinguere sullo schermo.

Le dimensioni di un pixel dipendono dalle dimensioni dello schermo e dalla risoluzione. Pertanto, anche se la libreria Swing utilizza il pixel come unità di lunghezza, questa non corrisponde a una lunghezza reale fissa. Su uno schermo ad alta risoluzione (cioè con molti pixel) un oggetto con dimensioni 300 x 200 pixel apparirà molto piccolo. Lo stesso oggetto apparirà invece molto grande su uno schermo a bassa risoluzione.

Per esempio, si consideri la seguente istruzione, tratta dal Listato 17.1:

```
finestra.setSize(LARGHEZZA, ALTEZZA);
```

Essa è equivalente a

```
finestra.setSize(300, 200);
```

Questa istruzione specifica che l'oggetto `finestra` sarà largo 300 pixel e alto 200 pixel. Le dimensioni reali dipenderanno dalla risoluzione e dalle dimensioni dello schermo sul quale viene visualizzata la finestra.

Quindi, nonostante sia vero che i programmi basati su Java e sulla libreria Swing sono portabili e che funzioneranno su qualunque sistema che supporti Java, le dimensioni effettive degli oggetti visualizzati cambieranno, in generale, da uno schermo all'altro, come conseguenza delle diverse dimensioni reali dei pixel. Questa è una caratteristica di Java che non è esattamente portabile come si vorrebbe. Per ottenere una finestra di dimensioni reali prefissate, potrebbe essere necessario cambiare le sue dimensioni in pixel a seconda della risoluzione e delle dimensioni dello schermo sul quale sarà visualizzata. In ogni caso, sono solo le dimensioni assolute che cambieranno da uno schermo all'altro. Le dimensioni relative tra gli oggetti saranno sempre le stesse, indipendentemente dallo schermo sul quale gli oggetti sono visualizzati. Inoltre, nella maggior parte dei sistemi l'utente può utilizzare il mouse per ridimensionare una finestra una volta che questa è stata visualizzata.

Pixel

Un pixel è la più piccola unità di spazio che può essere gestita indipendentemente su uno schermo. Nella libreria Swing, sia le dimensioni che la posizione di un oggetto sullo schermo sono misurate in pixel.

Risoluzione

La risoluzione di uno schermo è una misura del numero di pixel che può visualizzare. Più elevato è il numero di pixel, maggiori saranno la risoluzione e il livello di dettaglio ottenibile. Se uno schermo ha una risoluzione bassa, non sarà possibile vedere dettagli troppo fini. L'unico modo per visualizzare un oggetto piccolo su uno schermo a bassa risoluzione è rendendolo più grande. Di conseguenza, un oggetto apparirà più piccolo su uno schermo ad alta risoluzione e più grande su uno a bassa risoluzione.

17.2.3 Ulteriori dettagli sul metodo `setVisible`

Si considerino di nuovo il metodo `setVisible` e la sua invocazione nel Listato 17.1:

```
finestra.setVisible(true);
```

Il metodo `setVisible` richiede un argomento di tipo `boolean`. In altre parole, l'argomento di `setVisible` sarà sempre `true` oppure `false`. Se `w` è un oggetto, come una finestra, che possa essere visualizzato sullo schermo, la chiamata

```
w.setVisible(true);
```

renderà w visibile, mentre la chiamata

```
w.setVisible(false);
```

lo renderà invisibile. Molte delle classi della libreria Swing hanno un metodo `setVisible` che funziona in questo modo.

Si potrebbe essere portati a pensare che la visualizzazione di un oggetto sullo schermo dovrebbe essere automatica. Dopo tutto, che senso avrebbe definire un elemento di un'interfaccia utente grafica se poi non lo si visualizza? La risposta è che si potrebbe volere che l'oggetto non sia sempre visibile. Senza dubbio sarà capitato di lavorare con sistemi di gestione delle interfacce utente grafiche nei quali alcuni elementi compaiono e scompaiono, per esempio perché non sono più necessari (come nel caso dell'elenco delle voci di un menu dopo che è stata effettuata la scelta) o perché una finestra ne nasconde un'altra. La libreria Swing non può sapere a priori quando una finestra o un altro elemento dell'interfaccia utente grafica debba essere mostrato. È il programmatore che deve dire al sistema quando mostrare la finestra, inserendo una chiamata al metodo `setVisible`.

Se si esegue il programma del Listato 17.1 omettendo la chiamata a `setVisible`, non si vedrà nulla sullo schermo. La finestra verrà costruita, ma non mostrata (ma attenzione: poiché la finestra non viene mostrata, non sarà visualizzato nemmeno il pulsante di chiusura e quindi non ci sarà modo di terminare il programma!).

Il metodo `setVisible`

Molte delle classi della libreria Swing hanno un metodo `setVisible`. Questo metodo richiede un argomento di tipo `boolean`. Se `w` è un oggetto, come una finestra, che possa essere visualizzato sullo schermo, la chiamata

```
w.setVisible(true);
```

renderà w visibile, mentre la chiamata

```
w.setVisible(false);
```

lo renderà invisibile, cioè lo nasconderà.

Sintassi

```
oggetto.setVisible(espressione_booleana);
```

Esempio (dal Listato 17.1):

```
public static void main(String args[]) {
    JFrame finestra = new JFrame();
    ...
    finestra.setVisible(true);
}
```

ESEMPIO DI PROGRAMMAZIONE
UNA VERSIONE MIGLIORATA DEL PRIMO PROGRAMMA SWING

È ora possibile riformulare il programma di esempio del Listato 17.1 nello stile che è buona norma seguire quando si costruisce un'interfaccia utente grafica. Questa nuova versione si comporterà essenzialmente come la precedente, ma mostrerà due finestre anziché una. La definizione della finestra è in una classe a parte. La finestra verrà poi visualizzata da un programma separato. Una classe come `PrimaFinestra` del Listato 17.3 è un esempio tipico di classe per un'interfaccia utente grafica di input/output utilizzabile in vari programmi.

Si osservi per prima cosa che `PrimaFinestra` è una classe derivata da `JFrame`. Questo è il modo usuale per definire una finestra. La classe base `JFrame` fornisce alcune funzionalità di base, mentre la classe derivata aggiunge tutte le caratteristiche in più richieste. Una classe derivata da `JFrame` sarà anche detta **classe `JFrame`.**

Si noti che il costruttore del Listato 17.3 inizia con una chiamata al costruttore della classe base:

```
super();
```

Come osservato nel Capitolo 10, questa istruzione garantisce che sia effettuata l'inizializzazione normalmente richiesta per gli oggetti `JFrame`. Se il costruttore della classe base non ha argomenti, verrà comunque chiamato automaticamente, anche se non lo si fa esplicitamente. In questo caso, quindi, la chiamata al costruttore della classe base potrebbe essere omessa, ma è stata inclusa per ricordare che tale chiamata è invece necessaria se, come accade a volte, occorre passare dei parametri al costruttore della classe base.

Si noti che quasi tutta l'inizializzazione della finestra `PrimaFinestra` del Listato 17.3 è collocata nel costruttore della classe. Questo è l'approccio più corretto. Tutte le operazioni di inizializzazione, come l'impostazione delle dimensioni iniziali, dovrebbero far parte della definizione della classe e non essere realizzate come operazioni sulle istanze della classe, come accadeva nel Listato 17.1. Tutti i metodi di inizializzazione, come `setSize` a `addWindowListener`, sono ereditati dalla classe `JFrame`. Poiché sono utilizzati nel costruttore, è la finestra stessa l'oggetto chiamante. In altre parole, un'invocazione di metodo come

```
setSize(LARGHEZZA, ALTEZZA);
```

è equivalente a

```
this.setSize(LARGHEZZA, ALTEZZA);
```

A parte il fatto di essere gestiti nel costruttore di una classe derivata, i dettagli della definizione di `PrimaFinestra` sono essenzialmente gli stessi del Listato 17.1.

LISTATO 17.3 Una classe finestra basata su Swing.

```
import javax.swing.JFrame;
import javax.swing.JLabel;

/**
```

```
public class PrimaFinestra extends JFrame {
    public static final int LARGHEZZA = 300;
    public static final int ALTEZZA = 200;

    public PrimaFinestra() {
        super();

        setSize(LARGHEZZA, ALTEZZA);
        JLabel etichetta = new JLabel("Non premere quel pulsante!");
        add(etichetta);

        DistruttoreFinestra ascoltatore = new DistruttoreFinestra();
        addWindowListener(ascoltatore);
    }
}
```

Si consideri ora il programma riportato nel Listato 17.4, che utilizza la classe `PrimaFinestra`. Il programma non fa altro che mostrare due oggetti identici della classe `PrimaFinestra`. Le due finestre verranno probabilmente visualizzate esattamente una sopra l'altra, quindi eseguendo il programma potrebbe sembrare che ne venga mostrata solo una. Tuttavia, utilizzando il mouse per spostare quella visibile si vedrà anche l'altra sotto di essa.

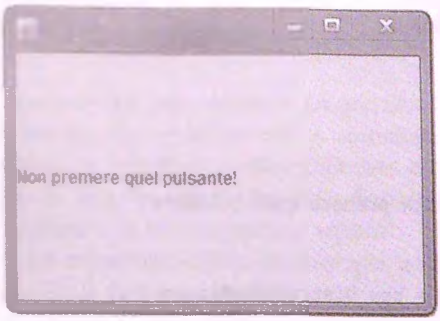
Anche se la maggior parte dell'inizializzazione per le finestre del Listato 17.4 è stata spostata nel costruttore della classe finestra `PrimaFinestra`, le chiamate al metodo `setVisible` sono state lasciate nel programma che utilizza la classe. Se anche queste chiamate fossero state spostate nel costruttore di `PrimaFinestra`, eseguendo il programma si sarebbe ottenuto lo stesso risultato. Tuttavia, normalmente è il programma applicativo che dovrebbe decidere quando mostrare ogni finestra e, quindi, dovrebbe contenere le chiamate a `setVisible`. Chi scrive la classe `PrimaFinestra` non può sapere a priori quando chi scrive il programma che la utilizza vorrà rendere visibili o invisibili le finestre.

LISTATO 17.4 Un programma che utilizza la classe `PrimaFinestra`.

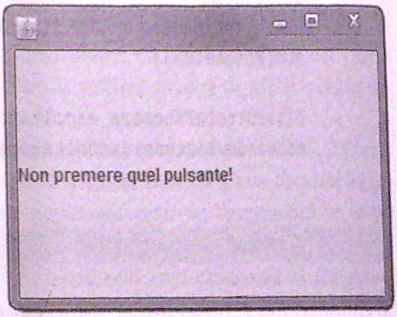
MyLab

```
/**
 * Un semplice programma di esempio di utilizzo di una classe finestra.
 * Per vedere entrambe le finestre, probabilmente sarà necessario
 * spostare quella visibile.
 */
public class PrimaFinestraDemo {
    public static void main(String args[]) {
        PrimaFinestra finestra1 = new PrimaFinestra();
        finestra1.setVisible(true);

        PrimaFinestra finestra2 = new PrimaFinestra();
        finestra2.setVisible(true);
    }
}
```

Output sullo schermo



Se quando si esegue il programma sembra che ci sia una sola finestra, la si sposti. Le due finestre potrebbero essere una sopra l'altra.



Aggiunta di elementi a una finestra `JFrame`

È possibile aggiungere un oggetto `JLabel` a una finestra `JFrame` come segue:

Sintassi (all'interno di un costruttore)

```
add(oggetto_JLabel);
```

Esempio (all'interno di un costruttore)

```
JLabel etichetta = new JLabel("Non premere quel pulsante!");
add(etichetta);
```

È anche possibile aggiungere altri tipi di oggetto a un `JFrame`, come si vedrà più avanti nel capitolo.

ESEMPIO DI PROGRAMMAZIONE
UNA FINESTRA COLORATA

Il Listato 17.5 contiene una variante della classe del Listato 17.3 con due costruttori. Il costruttore di default (quello senza argomenti) è come quello del Listato 17.3, ma comprende tre nuovi elementi:

- un titolo ("Seconda Finestra");
- un colore di sfondo (rosso);
- un nuovo modo di aggiungere un *window listener*.

Si considerino ora questi nuovi elementi uno alla volta.

Si può assegnare un titolo alla finestra nel modo seguente:

```
setTitle("Seconda Finestra");
```

Il metodo `setTitle` è ereditato dalla classe `JFrame`. Questo metodo riceve come parametro una stringa che viene riportata nella barra del titolo della finestra.

Si notino le seguenti istruzioni nel costruttore di default nel Listato 17.5:

```
Container pannelloDelContenuto = getContentPane();
pannelloDelContenuto.setBackground(Color.BLUE)
```

Il metodo `getContentPane` è ereditato dalla classe `JFrame` e restituisce il **pannello del contenuto** (***content pane***) della finestra, che è un oggetto di tipo `Container` (classe che sarà approfondita più avanti). Questo pannello occupa tutta la "parte interna" del `JFrame` e quindi colorando il pannello del contenuto si colora la parte interna della finestra. Si noti che anche la classe `JFrame` ha un metodo `setBackground`. Tuttavia, utilizzando direttamente questo metodo non si noterebbe alcun effetto, dato che la parte interna del `JFrame` è coperta interamente dal pannello. È necessario tenere conto esplicitamente della presenza del pannello del contenuto di un `JFrame` solo quando si vuole cambiare il colore dell'interno di una finestra. Per tutte le altre operazioni, si può agire direttamente sul `JFrame`. Inoltre, come si vedrà più avanti, questa complicazione non si presenta quando si vuole cambiare il colore di qualche altro elemento dell'interfaccia utente grafica, come un pulsante. In quel caso, è sufficiente utilizzare direttamente il metodo `setBackground` dell'elemento in questione.

Sarebbe stato anche possibile modificare il colore della finestra semplicemente con la seguente istruzione che non fa uso di una variabile locale di tipo `Container` come mostrato nel secondo costruttore nel Listato 17.5:

```
getContentPane().setBackground(colorePersonalizzato);
```

Il metodo `setBackground` richiede un argomento che descriva il colore da dare allo sfondo. La classe `Color` fornisce delle costanti predefinite per molti dei colori più utilizzati. Le costanti disponibili sono elencate nella Figura 17.4. Per vedere a quale colore corrisponda effettivamente ogni costante, si può sostituire la costante `Color.BLUE` nel Listato 17.5 con una delle altre e poi ricompilare ed eseguire il programma. È anche possibile utilizzare la classe `Color` per definire colori personalizzati, ma questo argomento non sarà trattato in questo testo. Poiché la classe `Color` fa parte del package AWT, occorre inserire la seguente istruzione di importazione quando si utilizza la classe `Color` in un programma:

```
import java.awt.Color;
```

Alla fine della definizione del primo costruttore, si aggiunge un *window listener* alla finestra chiamando il metodo `addWindowListener`:

```
addWindowListener(new DistruttoreFinestra());
```

`Color.BLACK`	Nero
`Color.BLUE`	Blu
`Color.CYAN`	Ciano
`Color.DARK_GRAY`	Grigio scuro
`Color.GRAY`	Grigio
`Color.GREEN`	Verde
`Color.LIGHT_GRAY`	Grigio chiaro
`Color.MAGENTA`	Magenta
`Color.ORANGE`	Arancione
`Color.PINK`	Rosa
`Color.RED`	Rosso
`Color.WHITE`	Bianco
`Color.YELLOW`	Giallo

Figura 17.4 Le costanti della classe `Color`.

Alcuni metodi, come `addWindowListener`, richiedono degli oggetti come argomenti, ma una volta che l'oggetto è stato passato come parametro al metodo, non è più necessario utilizzarlo nel programma. Ciò significa che non occorre assegnare un nome all'oggetto. Un argomento come `new DistruttoreFinestra()` nell'invocazione a `addWindowListener` rappresenta un modo per creare un oggetto e passarlo come parametro a un metodo senza la necessità di assegnargli un nome. Dato che non ha nome, l'oggetto creato da `new DistruttoreFinestra()` è detto **oggetto anonimo**.

L'invocazione appena discussa è equivalente al seguente codice:

```
DistruttoreFinestra ascoltatore = new DistruttoreFinestra();
addWindowListener(ascoltatore);
```

L'unica differenza tra i due modi di assegnare un ascoltatore alla finestra è che nel primo non ci si preoccupa di assegnare un nome all'oggetto di tipo `DistruttoreFinestra`, mentre nel secondo gli si dà il nome `ascoltatore`.

Si consideri ora il secondo costruttore. È molto simile al primo, a eccezione di come gestisce il colore di sfondo. Il secondo costruttore richiede un parametro di tipo `Color` e imposta il colore di sfondo a quello specificato da tale parametro.

Si può evidenziare facilmente la differenza tra i due costruttori osservando come vengono utilizzati nel programma di esempio del Listato 17.6. Le due finestre visualizzate sono identiche tranne per il fatto che una ha lo sfondo blu e l'altra ha lo sfondo rosa. Come nel caso dell'esempio precedente, quando si esegue il programma del Listato 17.6 le due finestre compariranno probabilmente una sopra l'altra e sembrerà che ne sia stata visualizzata una sola. È però sufficiente utilizzare il mouse per spostare quella visibile per vedere anche la seconda.

LISTATO 17.5 Un altro esempio di finestra basata su Swing.

MyLab

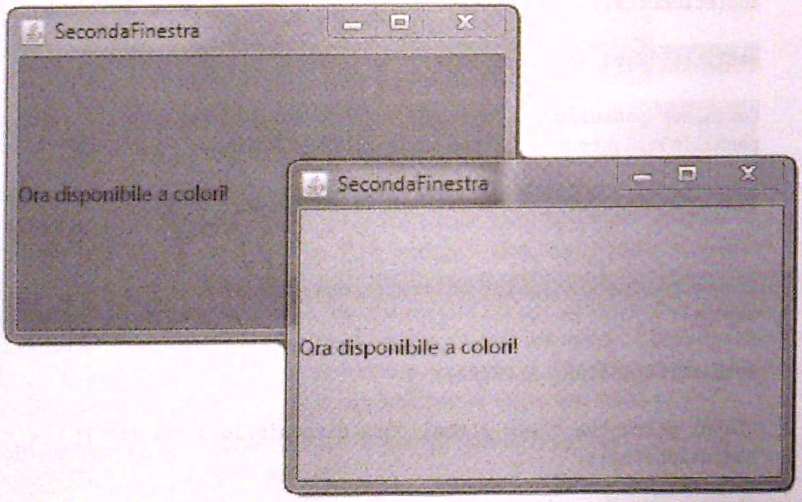
```
import javax.swing.JFrame;
import javax.swing.JLabel;
import java.awt.Color;
import java.awt.Container;

public class SecondaFinestra extends JFrame {
   public static final int LARGHEZZA = 300;
   public static final int ALTEZZA = 200;

   public SecondaFinestra() {
      super();

      setSize(LARGHEZZA, ALTEZZA);

      JLabel etichetta = new JLabel("Ora disponibile a colori!");
      add(etichetta);

      setTitle("Seconda Finestra");

      Container pannelloDelContenuto = getContentPane();
      pannelloDelContenuto.setBackground(Color.BLUE);

      addWindowListener(new DistruttoreFinestra());
   }

   public SecondaFinestra(Color colorePersonalizzato) {
      super();

      setSize(LARGHEZZA, ALTEZZA);

      JLabel etichetta = new JLabel("Ora disponibile a colori!");
      add(etichetta);

      setTitle("SecondaFinestra");

      getContentPane().setBackground(colorePersonalizzato);

      addWindowListener(new DistruttoreFinestra());
   }
}
```

MyLab

LISTATO 17.6 Un programma di esempio per la classe `SecondaFinestra`.

```
import java.awt.Color;

public class SecondaFinestraDemo {
    /**
     * Crea e visualizza due finestre della classe SecondaFinestra.
     */
    public static void main(String args[]) {
        SecondaFinestra finestra1 = new SecondaFinestra();
        finestra1.setVisible(true);

        SecondaFinestra finestra2 = new SecondaFinestra(Color.PINK);
        finestra2.setVisible(true);
    }
}
```

Output sullo schermo



FAQ Perché una finestra `JFrame` ha un pannello del contenuto?

Sfortunatamente, la risposta alla domanda sul perché una finestra `JFrame` abbia un pannello che occupa tutta la parte interna e che contiene tutto il contenuto della finestra coinvolgerebbe modi di utilizzare un `JFrame` che non saranno discussi in questo testo. Se si ritiene che un tale pannello sia superfluo, in effetti si tratta di una osservazione sensata. Non occorre utilizzare direttamente questo pannello per le operazioni descritte in questo testo, a eccezione dell'impostazione del colore di sfondo di una finestra. Infatti, il precursore della classe `JFrame`, in una libreria di classi più vecchia, non aveva alcun pannello per il proprio contenuto. Tuttavia un `JFrame` ce l'ha, ed è necessario ricordarsene affinché certe operazioni funzionino correttamente.

FAQ Che colori sono ciano e magenta?

Il ciano è un blu molto chiaro. Il magenta è simile al porpora. Ci si potrebbe chiedere perché i progettisti delle librerie Java abbiano deciso di includere certi colori nella classe `Color`. Colori come nero, bianco, rosso, verde e la maggior parte degli altri sono comuni, ma perché includere anche ciano e magenta? La risposta è legata a uno dei modi più comuni di produrre colori arbitrari, che consiste nel mescolare i quattro colori base nero, giallo, ciano e magenta. Ciano e magenta sono quindi a tutti gli effetti dei "colori di base". L'altro modo comunemente utilizzato per produrre colori arbitrari è mescolare rosso, verde e blu. I display dei computer e i televisori utilizzano il secondo metodo, mentre la stampa a colori sfrutta il primo.

Le istruzioni di importazione per l'utilizzo delle classi Swing

Può risultare difficile ricordarsi quali istruzioni di importazione siano necessarie per un programma o una definizione di classe che utilizzino Swing. Nonostante questo, negli esempi di questo libro vengono sempre importate solo le classi strettamente necessarie. Tuttavia, alcuni programmatori, per semplicità, importano interi package al posto di singole classi. Se si preferisce agire così, le seguenti istruzioni saranno sufficienti per la maggior parte delle interfacce utente grafiche basate su Swing:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

17.2.4 Alcuni metodi della classe `JFrame`

La Figura 17.5 riporta alcuni dei metodi della classe `JFrame`. Si ricordi che una classe di tipo finestra è normalmente derivata da `JFrame` e quindi erediterà tutti questi metodi (a eccezione dei costruttori).

17.2.5 Gestori di layout

Si è visto che è possibile aggiungere un oggetto `JLabel` a una finestra `JFrame` semplicemente tramite il metodo `add`. La classe `SecondaFinestra` del Listato 17.5 mostra un esempio di questa tecnica.

Se si aggiunge un'unica etichetta, sembra che non ci possano essere dubbi su dove essa verrà piazzata, ma cosa accade se si aggiungono due o più etichette? Come saranno disposte? Una sopra l'altra? O una a fianco dell'altra? Quale sarà la prima e quale la seconda? E così via. La disposizione è gestita, secondo regole opportune, da un oggetto noto come **gestore di struttura** o **gestore di layout** (***layout manager***). Gestori di layout diversi seguono regole diverse. Verranno ora mostrati alcuni esempi.

Il Listato 17.7 contiene una classe che crea una finestra con tre etichette. La classe utilizza un gestore di layout per piazzare le tre etichette una sopra l'altra su tre righe. `BorderLayout` è una classe per la gestione di layout. Un oggetto di tipo `BorderLayout`

è aggiunto come gestore di layout all'interfaccia utente grafica del Listato 17.7 tramite l'istruzione

```
setLayout(new BorderLayout());
```

Può essere utile notare che questa istruzione è equivalente alle due seguenti:

```
BorderLayout gestore = new BorderLayout();
setLayout(gestore);
```

`public JFrame()` Crea una nuova finestra `JFrame`.
`public JFrame(String titolo)` Crea una nuova finestra `JFrame` con il titolo specificato.
`public void add(Component componente)` Aggiunge il componente specificato al `JFrame`.
`public void addWindowListener(WindowListener ascoltatore)` Registra `ascoltatore` come ascoltatore di eventi generati dalla finestra.
`public Container getContentPane()` Restituisce il pannello che comprende l'intero contenuto del `JFrame`. Si noti che il pannello restituito è di tipo `Container` (si veda più avanti nel capitolo per una discussione di pannelli e contenitori). Questo metodo serve soltanto quando si vuole impostare il colore di sfondo della finestra.
`public void setBackground(Color c)` Imposta il colore di sfondo a quello specificato. Questo metodo può essere utilizzato direttamente solo sui componenti derivati da `JFrame`. Per cambiare il colore di sfondo di una finestra `JFrame`, occorre passare attraverso il suo pannello del contenuto.
`public void setForeground(Color c)` Imposta il colore degli oggetti in primo piano nel `JFrame`. Come il metodo precedente, anche questo è utile solo quando applicato ad un componente derivato da `JFrame`.
`public void setSize(int larghezza, int altezza)` Imposta le dimensioni della finestra ai valori specificati.
`public void setTitle(String titolo)` Mostra il titolo specificato nella barra del titolo della finestra.
`public void setVisible(boolean visibile)` Rende la finestra visibile se l'argomento è `true`, la nasconde se l'argomento è `false`.

Figura 17.5 Alcuni metodi della classe `JFrame`.

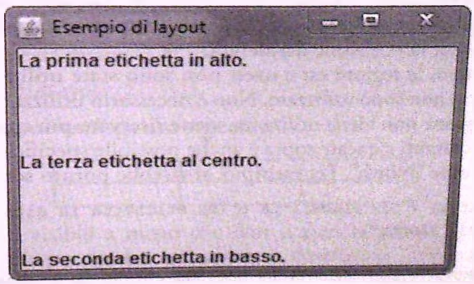
Lab

LISTATO 17.7 Utilizzo di un gestore `BorderLayout`.

```
import javax.swing.JFrame;
import javax.swing.JLabel;
import java.awt.BorderLayout;
import java.awt.Container;
```

```
/**
 * Un semplice esempio di utilizzo di un gestore di layout
 * per disporre delle etichette.
 */
public class BorderLayoutDemo extends JFrame {
   public static final int LARGHEZZA = 300;
   public static final int ALTEZZA = 200;

   public BorderLayoutDemo() {
      setSize(LARGHEZZA, ALTEZZA);
      addWindowListener(new DistruttoreFinestra());
      setTitle("Esempio di layout");

      setLayout(new BorderLayout());

      JLabel etichetta1 = new JLabel("La prima etichetta in alto.");
      add(etichetta1, BorderLayout.NORTH);

      JLabel etichetta2 = new JLabel("La seconda etichetta in basso.");
      add(etichetta2, BorderLayout.SOUTH);

      JLabel etichetta3 = new JLabel("La terza etichetta al centro.");
      add(etichetta3, BorderLayout.CENTER);
   }

   /**
    * Crea e visualizza una finestra della classe EsempioBorderLayout.
    */
   public static void main(String[] args) {
      BorderLayoutDemo gui = new BorderLayoutDemo();
      gui.setVisible(true);
   }
}
```

Output sullo schermo



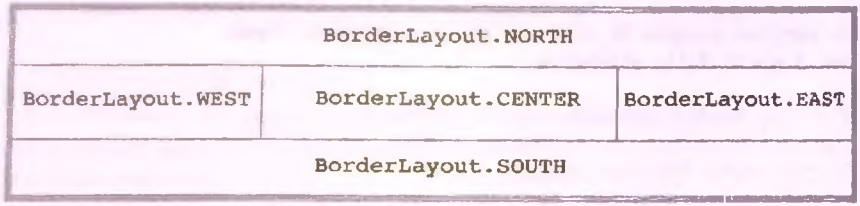


Figura 17.6 Le cinque regioni di un gestore `BorderLayout`.

Un gestore di layout di tipo `BorderLayout` dispone gli elementi in cinque regioni denominate nord (*north*), sud (*south*), est (*east*), ovest (*west*) e centro (*center*). Queste regioni sono descritte dalle cinque costanti `BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.EAST`, `BorderLayout.WEST` e `BorderLayout.CENTER`, mostrate nella Figura 17.6. In questa figura, il rettangolo esterno rappresenta la finestra o qualunque altro tipo di contenitore al quale venga applicato questo tipo di gestore. Le cinque regioni sono invece mostrate come separate l'una dall'altra da righe più sottili. Si noti che in una vera interfaccia utente grafica queste righe non saranno visibili a meno che non si faccia qualcosa per renderle tali. Sono state mostrate nella figura solo per evidenziare dove è posizionata ogni regione.

Si consideri ancora l'esempio del Listato 17.7. Le etichette vengono aggiunte tramite le seguenti istruzioni:

```
JLabel etichetta1 = new JLabel("La prima etichetta in alto.");
add(etichetta1, BorderLayout.NORTH);

JLabel etichetta2 = new JLabel("La seconda etichetta in basso.");
add(etichetta2, BorderLayout.SOUTH);

JLabel etichetta3 = new JLabel("La terza etichetta al centro.");
add(etichetta3, BorderLayout.CENTER);
```

Quando si utilizza un gestore di tipo `BorderLayout`, si specifica la regione nella quale posizionare un elemento come secondo argomento del metodo `add`. Si noti che non occorre aggiungere gli elementi in un ordine particolare, dato che il secondo argomento specifica completamente la posizione desiderata.

In questo esempio, le regioni est e ovest non sono state utilizzate. Cosa ne è stato? La risposta è semplice: non sono utilizzate. Non è necessario utilizzare tutte e cinque le regioni. Se qualche regione non viene utilizzata, viene riservato più spazio a quella centrale.

Al posto delle costanti elencate sopra è anche possibile specificare una regione utilizzando una stringa come "North". Per esempio, si sarebbe potuto scrivere

```
JLabel etichetta1 = new JLabel("La prima etichetta in alto.");
add(etichetta1, "North");

JLabel etichetta2 = new JLabel("La seconda etichetta in basso.");
add(etichetta2, "South");
```

```
JLabel etichetta3 = new JLabel("La terza etichetta al centro.");
add(etichetta3, "Center");
```

La variante con le stringhe è un po' più veloce da scrivere, ma entrambi gli approcci sono corretti.

In base a quanto detto finora, potrebbe sembrare che si possa inserire un'unica etichetta (o un unico oggetto di altro tipo) in ogni regione, ma più avanti nel capitolo si vedrà che è possibile raggruppare tra loro più elementi e includerli nella stessa regione. Di conseguenza, in ogni regione è possibile inserire più elementi.

Nonostante sia possibile definire gestori di layout personalizzati, nella maggior parte dei casi i gestori predefiniti saranno più che sufficienti, pertanto non si discuterà qui il problema della definizione di gestori personalizzati.

Il tipo più semplice di gestore di layout è un oggetto della classe `FlowLayout`, che dispone gli elementi uno dopo l'altro da sinistra a destra, nell'ordine nel quale essi sono stati aggiunti. Per esempio, se nella classe del Listato 17.7 si fosse utilizzato il gestore `FlowLayout`, il codice sarebbe stato il seguente:

```
setLayout(new FlowLayout());

JLabel etichetta1 = new JLabel("La prima etichetta in alto.");
add(etichetta1);

JLabel etichetta2 = new JLabel("La seconda etichetta in basso.");
add(etichetta2);

JLabel etichetta3 = new JLabel("La terza etichetta al centro.");
add(etichetta3);
```

Si noti che in questo caso il metodo `add` viene utilizzato con un unico argomento e che con un `FlowLayout` gli elementi sono visualizzati nell'ordine in cui sono stati inseriti. Di conseguenza, le etichette dell'esempio comparirebbero nella forma

```
La prima etichetta in alto.La seconda etichetta in basso.La terza etichetta
al centro.
```

Nel corso del capitolo verranno illustrati vari esempi di interfacce utente grafiche che sfruttano il gestore `FlowLayout`.

Un oggetto della classe `GridLayout` dispone gli elementi in righe e colonne, assegnando a ognuno la stessa quantità di spazio. Per esempio, l'istruzione

```
setLayout(new GridLayout(2, 3));
```

produce una disposizione analoga alla seguente:

Le righe non saranno visibili a meno che non si faccia qualcosa per renderle tali. Qui sono state mostrate solo per rendere più chiara la descrizione. I due numeri passati come parametri al costruttore di `GridLayout` specificano, rispettivamente, il numero di righe e di colonne.

Gestore di layout	Descrizione
`BorderLayout`	Dispone i componenti in cinque aree: nord, sud, est, ovest e centro. L'area da utilizzare è specificata come secondo argomento del metodo `add`.
`FlowLayout`	Dispone i componenti da sinistra a destra, nello stesso modo in cui, normalmente, si scrive su un foglio di carta.
`GridLayout`	Dispone i componenti in una griglia composta da righe e colonne, nella quale ogni componente viene ridimensionato in modo che occupi tutta la sua cella nella griglia.

Figura 17.7 Alcuni gestori di layout.

Anche se in fase di costruzione di un `GridLayout` è necessario specificare un numero di colonne, l'effettivo numero di colonne utilizzate sarà determinato dal numero di elementi aggiunti al contenitore. Se si aggiungono sei elementi, la griglia apparirà come mostrato. Se però se ne aggiungono sette o otto, verrà aggiunta automaticamente una quarta colonna, e così via. Se invece gli elementi sono meno di cinque, ci saranno due righe e un numero minore di colonne.

Quando si utilizza la classe `GridLayout`, il metodo `add` ha un unico argomento. Gli elementi vengono disposti nella griglia da sinistra a destra completando inizialmente la prima riga, poi la seconda e così via. Non è possibile saltare qualche posizione della griglia, anche se più avanti si vedrà che si può inserire qualcosa che non viene visualizzato, in modo da dare l'impressione di aver saltato delle posizioni.

La descrizione dei tre gestori di layout appena discussi è riportata nella Figura 17.7.

Se non si definisce esplicitamente un gestore di layout, ne verrà applicato uno di default. Per esempio, nel Listato 17.5 non è stato specificato alcun gestore, ma è stato comunque possibile inserire un'etichetta, proprio grazie al gestore di default. Quando si usa un `JFrame`, il gestore di layout di default è `BorderLayout`. Se si utilizza `BorderLayout` (esplicitamente o nel caso di default) e non si specifica un secondo argomento per il metodo `add`, l'effetto è lo stesso che se si utilizzasse l'opzione `BorderLayout.CENTER` come secondo argomento. Nei primi programmi presentati in questo capitolo è stato sfruttato questo comportamento, ma d'ora in poi il secondo argomento verrà sempre specificato quando si utilizzerà un `BorderLayout`.

Il metodo `main` del Listato 17.7

Nel Listato 17.7, il metodo `main`, che crea e visualizza l'interfaccia utente grafica, è stato incluso nella definizione della classe finestra. Normalmente, come negli esempi precedenti, questo metodo viene posto in una classe a parte, ma includerlo nella definizione della classe finestra è perfettamente legale e, a volte, comodo per provare velocemente il comportamento della classe. Si noti che, anche quando è inserito nella classe finestra stessa, il metodo `main` viene scritto allo stesso modo di quando è incluso in qualche altra classe. In particolare, è necessario costruire un oggetto della classe, come nella riga seguente, tratta dal Listato 17.7:

```
BorderLayoutDemo gui = new BorderLayoutDemo();
```

Gestori di layout

La disposizione degli elementi che vengono aggiunti a una classe di tipo contenitore è gestita da un oggetto noto come gestore di layout. Per aggiungere un gestore di layout, si utilizza il metodo `setLayout`, che è disponibile in ogni classe di tipo contenitore, come una finestra `JFrame` o un oggetto di una delle classi contenitore che verranno introdotte più avanti. Se non si definisce un gestore di layout, ne verrà utilizzato uno di default.

Sintassi

oggetto_contenitore`.setLayout(new` *Classe_Gestore_Layout*`());`

Esempio (in un costruttore)

```
setLayout(new FlowLayout());
JLabel etichetta = new JLabel("Le etichette sono belle.");
add(etichetta);
```

Utilizzare un gestore di layout esplicito

Prima di affrontare il tema dei gestori di layout, è stata sfruttata semplicemente la presenza di un gestore di default. Tuttavia, è sempre preferibile specificare un gestore di layout. Questo approccio, infatti, rende il codice più chiaro e aumenta la probabilità che il comportamento di un'interfaccia utente grafica rimanga invariato nonostante possibile cambiamenti nelle future versioni della libreria Swing. D'ora in poi si utilizzeranno sempre gestori di layout espliciti.

17.3 Pulsanti e *action listener*

Finora, le interfacce utente grafiche presentate in questo capitolo hanno incluso azioni molto limitate: l'unica azione intrapresa era la terminazione del programma in seguito alla pressione del pulsante di chiusura di una finestra. In questa sezione si inizierà a mostrare come progettare interfacce utente grafiche che possano svolgere azioni più complicate, come modificare colori e testo, o altre operazioni complesse. Nei primi esempi, tali azioni avverranno come conseguenza della pressione di un pulsante da parte dell'utente.

Un pulsante (o bottone) è semplicemente un elemento di un'interfaccia utente grafica che assomiglia a un vero pulsante e che scatena qualche azione quando lo si preme utilizzando il mouse. I pulsanti vengono creati in modo molto simile alle etichette. Anche il modo di aggiungere i pulsanti a un `JFrame` è lo stesso valido per le etichette, ma in questo caso c'è anche un aspetto nuovo: è possibile associare un'azione al pulsante, in modo che quando l'utente preme il pulsante l'interfaccia utente grafica esegua qualche operazione. Per rendere più chiari i dettagli di questo comportamento, verrà ora presentato un semplice esempio.

ESEMPIO DI PROGRAMMAZIONE
AGGIUNTA DI PULSANTI

Il Listato 17.8 contiene un programma che crea una finestra contenente due pulsanti, denominati `Rosso` e `Verde`. Se l'utente preme il pulsante `Rosso`, il colore della finestra cambia da quello attuale a rosso. Se invece l'utente preme il pulsante `Verde`, il colore della finestra diventa verde. Questo è tutto ciò che il programma fa, ma, come si può vedere, a poco a poco si sta imparando come costruire interfacce utente grafiche più complicate.

Gran parte di quanto presente nel Listato 17.8 è già familiare. La classe `PulsantiDemo` è derivata dalla classe `JFrame`, quindi si tratta di un'interfaccia a finestra simile a quelle viste precedentemente in questo capitolo. Anche in questo caso, un ascoltatore di tipo `DistruttoreFinestra` viene aggiunto tramite il metodo `addWindowListener`. Inoltre, vengono impostate le dimensioni esattamente come negli esempi già visti. Il gestore di layout utilizzato è un `FlowLayout`.

La novità del Listato 17.8 è costituita dall'utilizzo dei pulsanti della classe `JButton` e di un nuovo tipo di classe per l'ascolto di eventi. I pulsanti e questo nuovo tipo di ascoltatore saranno discussi nel prossimo paragrafo.

MyLab

LISTATO 17.8 Un'interfaccia utente grafica dotata di pulsanti.

Per comprendere appieno questo programma, è necessario studiare i prossimi due paragrafi, intitolati "Pulsanti" e "Action listener ed eventi di tipo azione"

```
import javax.swing.JButton;
import javax.swing.JFrame;
import java.awt.Color;
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * Un semplice esempio dell'uso di pulsanti in un JFrame.
 */
public class PulsantiDemo extends JFrame implements ActionListener {
    public static final int LARGHEZZA = 300;
    public static final int ALTEZZA = 200;

    public PulsantiDemo() {
        setSize(LARGHEZZA, ALTEZZA);

        addWindowListener(new DistruttoreFinestra());
        setTitle("Esempio pulsanti");
        Container pannello = getContentPane();
        pannello.setBackground(Color.BLUE);

        setLayout(new FlowLayout());

        JButton pulsanteStop = new JButton("Rosso");
        pulsanteStop.addActionListener(this);
        add(pulsanteStop);
```

```
        JButton pulsanteVia = new JButton("Verde");
        pulsanteVia.addActionListener(this);
        add(pulsanteVia);
    }

    public void actionPerformed(ActionEvent e) {
        Container pannello = getContentPane();

        if (e.getActionCommand().equals("Rosso"))
            pannello.setBackground(Color.RED);
        else if (e.getActionCommand().equals("Verde"))
            pannello.setBackground(Color.GREEN);
        else
            System.out.println("Errore nell'interfaccia.");
    }

    /**
     * Crea e visualizza una finestra della classe EsempioPulsanti.
     */
    public static void main(String args[]) {
        PulsantiDemo gui = new PulsantiDemo();
        gui.setVisible(true);
    }
}
```

Output sullo schermo (all'inizio, prima che venga premuto un pulsante)

17.3.1 Pulsanti

Questo paragrafo spiegherà come aggiungere pulsanti a un'interfaccia utente grafica, mentre il successivo illustrerà come specificare quali azioni debbano essere intraprese quando un pulsante viene premuto.

Un oggetto di tipo pulsante è costruito esattamente come qualunque altro oggetto, in questo caso utilizzando la classe `JButton`. Per esempio, la riga seguente, tratta dal Listato 17.8, crea un pulsante:

```
JButton pulsanteStop = new JButton("Rosso");
```

L'argomento al costruttore, `"Rosso"` nell'esempio, è la stringa che sarà scritta sul pulsante quando questo verrà visualizzato. Guardando l'interfaccia utente grafica del Listato 17.8, si vedrà che i due pulsanti sono etichettati `Rosso` e `Verde`.

L'istruzione seguente aggiunge il pulsante alla finestra:

```
add(pulsanteStop);
```

In questo caso, non è necessario specificare un secondo argomento, dato che si sta utilizzando un gestore di layout di tipo `FlowLayout`. Se si fosse utilizzato, invece, un `BorderLayout`, sarebbe stato possibile specificare un secondo argomento, come per esempio `BorderLayout.NORTH`.

Nel prossimo paragrafo verranno analizzate le istruzioni del Listato 17.8 che coinvolgono il metodo `addActionListener`.

La classe `JButton`

Un oggetto della classe `JButton` viene visualizzato in un'interfaccia utente grafica come un elemento che assomiglia a un vero pulsante. Cliccando con il mouse su di esso, si simula la pressione di un pulsante reale. Quando si crea un oggetto della classe `JButton`, è possibile passare al costruttore una stringa, che verrà scritta sul pulsante.

Gli oggetti della classe `JButton` possono essere aggiunti a una finestra `JFrame` tramite il metodo `add`. Come si vedrà in seguito, i pulsanti possono essere aggiunti anche ad altri elementi dell'interfaccia utente grafica in modo simile.

Esempio (in un costruttore)

```
setLayout(new FlowLayout());
JButton pulsanteStop = new JButton("Rosso");
add(pulsanteStop);
```

Il pulsante di chiusura di una finestra non è un oggetto `JButton`

I pulsanti che possono essere aggiunti esplicitamente a un'interfaccia utente grafica sono tutti oggetti della classe `JButton`. Tuttavia, il pulsante di chiusura di una finestra, che si ha a disposizione automaticamente in ogni classe derivata da `JFrame`, non è un oggetto della classe `JButton`, ma è semplicemente una parte di un oggetto `JFrame`.

FAQ Perché tanti nomi di classe cominciano con la lettera "J"?

La *J* sta per *Java*, no? Beh, sì, ma non è tutta qui la storia. Perché le classi `JFrame`, `JLabel` e `JButton` non sono state chiamate semplicemente `Frame`, `Label` e `Button`? La risposta è che la libreria AWT, precedente a Swing, conteneva già le classi `Frame`, `Label` e `Button`. Dato che Swing non sostituisce completamente SWT, e che quindi è necessario utilizzare ancora AWT insieme a Swing, i nomi delle classi Swing devono essere diversi da quelli delle classi AWT. Pertanto, i nomi delle classi Swing che hanno un corrispondente in AWT iniziano con una *J*.

Mescolare classi AWT e classi Swing

Una volta che si utilizza almeno una classe Swing, è opportuno utilizzare solo classi Swing al posto delle corrispondenti classi AWT. Per esempio, non si dovrebbe aggiungere un oggetto della classe AWT `Button` a una finestra `JFrame`. Si dovrebbe piuttosto utilizzare la classe Swing `JButton`.

17.3.2 *Action listener* ed eventi di tipo azione

L'attivazione di certi elementi di un'interfaccia utente grafica, come per esempio la pressione di un pulsante, genera un oggetto noto come **evento** e lo invia a uno o più altri oggetti noti come ascoltatori (*listener*). Si dice che viene "lanciato" un evento. Quando riceve un evento, un ascoltatore esegue qualche azione. Dicendo che l'evento viene "inviato" a un oggetto ascoltatore, si intende in realtà che viene invocato qualche metodo dell'ascoltatore al quale l'oggetto evento è passato come parametro. Questa invocazione avviene automaticamente, quindi la definizione di una classe per un'interfaccia utente grafica Swing non conterrà normalmente chiamate esplicite a questi metodi. Tuttavia, la definizione di un'interfaccia utente grafica deve fare due cose:

- per ogni pulsante deve specificare quali oggetti ascoltatori risponderanno agli eventi generati da quel pulsante; questo passo è detto registrazione dell'ascoltatore;
- deve definire i metodi che saranno chiamati quando l'evento è inviato all'ascoltatore.

La riga seguente, tratta dal Listato 17.8, registra `this` come ascoltatore degli eventi generati dal pulsante `pulsanteStop`:

```
pulsanteStop.addActionListener(this);
```

Un'istruzione analoga registra `this` anche come ascoltatore degli eventi generati dal pulsante `pulsanteVia`. Poiché l'argomento utilizzato è `this`, queste istruzioni significano che è la classe `PulsantiDemo` stessa a fare da ascoltatore per i suoi pulsanti. Si ricordi, infatti, che nell'ambito della definizione di una classe la parola chiave `this` indica l'istanza corrente della classe. Più precisamente, ogni oggetto della classe `PulsantiDemo` è l'ascoltatore dei pulsanti contenuti in quell'oggetto.

Tipi diversi di elementi di un'interfaccia utente grafica richiedono tipi diversi di classi ascoltatore per gestire gli eventi corrispondenti. Un pulsante genera eventi noti come **eventi azione** (***action event***), gestiti da ascoltatori noti come **ascoltatori di azioni** (anche se generalmente è più utilizzata l'espressione inglese ***action listener***). Un *action listener* è un oggetto di tipo `ActionListener`. Si noti che `ActionListener` non è una classe, ma un'interfaccia. Per rendere una classe adeguata a svolgere le funzioni di *action listener*, occorre fare due cose:

- aggiungere l'espressione `implements ActionListener` all'intestazione della definizione della classe;
- definire un metodo chiamato `actionPerformed`.

Nel Listato 17.8, la classe `EsempioPulsanti` è stata resa un *action listener* esattamente in questo modo. Di seguito è riportata la parte della definizione della classe che è rilevante per questa discussione:

```
public class PulsantiDemo extends JFrame implements ActionListener {
    ...
    public void actionPerformed(ActionEvent e) {
        ...
    }
    ...
}
```

Sarebbe stato anche possibile definire una classe a parte il cui unico compito consisteva nel gestire gli eventi generati dai pulsanti, ma è più semplice fare in modo che sia la classe `PulsantiDemo` stessa ad avere questo ruolo. Questo approccio è conveniente perché ci si aspetta che gli eventi generati dai pulsanti provochino modifiche alla finestra e il modo più semplice per modificare una finestra è farlo tramite un metodo della classe finestra stessa.

Si supponga di aver creato un oggetto, denominato `gui`, della classe `PulsantiDemo`:

```
PulsantiDemo gui = new PulsantiDemo();
```

All'interno della definizione della classe, la parola chiave `this` si riferisce all'oggetto `gui`. Quindi `gui` è l'ascoltatore degli eventi generati da entrambi i pulsanti all'interno di `gui`. Quando viene premuto un pulsante, il metodo `actionPerformed` viene chiamato automaticamente con l'evento generato dal pulsante come parametro. L'interazione tra i pulsanti e l'ascoltatore è illustrata nella Figura 17.8.

Se si fa click con il mouse su uno dei pulsanti di `gui`, un evento azione viene generato ed inviato all'ascoltatore registrato per quel pulsante. Ma è l'oggetto `gui` stesso l'ascoltatore per gli eventi dei suoi pulsanti, quindi l'evento viene inviato a `gui`. Quando un *action listener* riceve un evento di tipo azione, quest'ultimo è passato automaticamente al metodo `actionPerformed`, che solitamente contiene delle istruzioni condizionali per determinare l'azione corretta da eseguire per ogni evento.

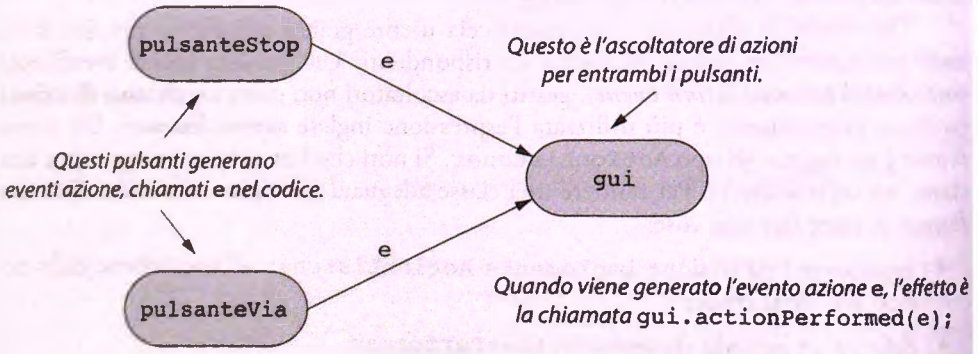


Figura 17.8 Pulsanti e ascoltatori di eventi azione.

Il metodo `actionPerformed` della classe `PulsantiDemo` del Listato 17.8 inizia con la riga seguente:

```
public void actionPerformed(ActionEvent e) {
```

Nell'esempio, il metodo deve sapere se l'evento `e` è stato generato dal pulsante `Rosso` o da quello `Verde`. Poiché il metodo `e.getActionCommand()` restituisce la stringa scritta sul pulsante che ha generato l'evento, in questo caso restituirà `"Rosso"` o `"Verde"`. Quindi tutto ciò che il metodo `actionPerformed` deve fare è controllare se `e.getActionCommand()` è uguale a `"Rosso"` oppure a `"Verde"` ed agire di conseguenza. Per fare questo confronto si può utilizzare il metodo `equals` della classe `String`. La chiamata

```
e.getActionCommand().equals(argomento_stringa)
```

restituirà `true` o `false` a seconda che il **comando associato all'azione** (***action command***) `e.getActionCommand()` sia o meno uguale a *argomento_stringa*. Quindi, il codice che segue controlla il valore di `e.getActionCommand()` e cambia il colore dell'interfaccia utente grafica nel modo adeguato:

```
if (e.getActionCommand().equals("Rosso"))
    pannello.setBackground(Color.RED);
else if (e.getActionCommand().equals("Verde"))
    pannello.setBackground(Color.GREEN);
else
    System.out.println("Errore nell'interfaccia.");
```

La clausola `else` finale non dovrebbe mai essere eseguita. Qui è stata inserita solo per ricevere un avvertimento nel caso si fosse fatto qualche errore nel codice.

Eventi di tipo azione e *action listener*

I pulsanti e certi altri tipi di elementi di un'interfaccia utente grafica generano eventi di tipo azione, che sono oggetti della classe `ActionEvent`. Tali eventi devono essere gestiti da un *action listener*. Qualunque classe può agire da gestore di eventi, in aggiunta alle sue altre funzioni. Per rendere una classe (che potrebbe essere una classe finestra) un *action listener*, si aggiunge l'espressione

```
implements ActionListener
```

all'intestazione della definizione della classe e si definisce un metodo chiamato `actionPerformed`.

Per fare in modo che un oggetto di una classe ascoltatrice di eventi sia effettivamente il gestore degli eventi generati da qualche elemento dell'interfaccia utente grafica, occorre registrare l'oggetto con il pulsante o altro elemento che genererà gli eventi di tipo azione. Per fare ciò, si utilizza il metodo `addActionListener`.

Esempio

```
public class PulsantiDemo extends JFrame implements ActionListener {
    ...
```

```
        public PulsantiDemo() {
            JButton pulsanteStop = new JButton("Rosso");
            pulsanteStop.addActionListener(this);
            add(pulsanteStop);
            ...
        }
        ...
        public void actionPerformed(ActionEvent e) {
            ...
        }
    }
```

I dettagli completi sono riportati nel Listato 17.8.

Il metodo `actionPerformed`

Una classe ascoltatrice di eventi di tipo azione deve avere un metodo chiamato `actionPerfomed` che accetti un argomento di tipo `ActionEvent`. Questo è l'unico metodo richiesto dall'interfaccia `ActionListener`.

Sintassi

```
public void actionPerformed(ActionEvent e) {
    codice_per_eseguire_le_azioni
}
```

Il *codice_per_eseguire_le_azioni* contiene, tipicamente, un'istruzione condizionale che dipende da qualche proprietà dell'argomento corrispondente al parametro `e`. Spesso l'istruzione condizionale dipende dal valore restituito da `e.getActionCommand()`. Se `e` è un evento generato dalla pressione di un pulsante, `e.getActionCommand()` restituisce una stringa, nota come comando associato all'azione. Di default, il comando è la stringa scritta sul pulsante, ma si può specificare un comando diverso utilizzando il metodo `setActionCommand`, come si vedrà più avanti nel capitolo.

Esempio

```
public void actionPerformed(ActionEvent e) {
    Container pannello = getContentPane();

    if (e.getActionCommand().equals("Rosso"))
        pannello.setBackground(Color.RED);
    else if (e.getActionCommand().equals("Verde"))
        pannello.setBackground(Color.GREEN);
    else
        System.out.println("Errore nell'interfaccia.");
}
```

Separare l'aspetto e il comportamento di un'interfaccia utente grafica

La scrittura del codice per un'interfaccia utente grafica basata su Swing può essere suddivisa in due parti: quella che definisce l'aspetto dell'interfaccia utente grafica e quella che ne definisce il comportamento. Per esempio, si consideri di nuovo il programma del Listato 17.8. Una prima versione di questo programma potrebbe utilizzare la seguente definizione del metodo `actionPerformed`:

```
public void actionPerformed(ActionEvent e) {
}
```

Con questa versione "nullafacente" del metodo `actionPerformed` (detta "segnaposto" o *stub*), il programma potrà essere eseguito e visualizzerà sullo schermo la finestra proprio come mostrato nel Listato 17.8. Premendo uno dei pulsanti, però, non accadrà nulla. Tuttavia, si può sfruttare questa fase della codifica per sistemare i dettagli dell'interfaccia utente grafica, per esempio per stabilire quale pulsante viene mostrato per primo. Si noti che sarebbe necessario definire comunque una versione vuota del metodo `actionPerformed`, perché questo metodo è richiesto dall'interfaccia `ActionListener` e l'intestazione della classe dichiara che la classe implementa tale interfaccia.

Una volta che si è soddisfatti dell'aspetto dell'interfaccia utente grafica, si può passare a definirne le azioni, che tipicamente sono determinate dal corpo del metodo `actionPerformed`. Suddividere la scrittura del codice di un'interfaccia utente grafica complessa in due parti più semplici (aspetto e comportamento) può rendere più facilmente affrontabile un compito che altrimenti potrebbe rivelarsi molto difficile.

17.3.3 Il pattern Model-View-Controller

La tecnica suggerita nel TIP precedente è un esempio di una tecnica più generale nota come pattern **Modello-Vista-Controllore** (***Model-View-Controller***). La Figura 17.9 illustra questo pattern. La parte di Modello del pattern è quella che esegue le operazioni che costituiscono il cuore di un'applicazione. La parte di Vista è quella di output: mostra una raffigurazione dello stato del Modello. Il Controllore rappresenta la parte di input: gestisce i comandi dall'utente verso il Modello. Ognuna di queste tre parti interagenti è realizzata normalmente come un oggetto che si occupa solo della parte che gli compete. In un caso semplice, come quello del Listato 17.8, un singolo oggetto può avere metodi diversi che realizzano i tre ruoli di Modello, Vista e Controllore.

Per semplicità, il pattern Model-View-Controller è stato presentato nel caso in cui l'utente interagisce direttamente con il Controllore. Tuttavia, non è necessario che il Controllore sia sotto il diretto controllo dell'utente. Il Controllore potrebbe anche essere gestito da qualche altro componente software o dall'hardware. In un'interfaccia utente grafica Swing, le parti Vista e Controllore potrebbero corrispondere a classi separate combinate in una classe più grande che visualizzi una singola finestra, consentendo così a entrambe le parti di interagire con l'utente.

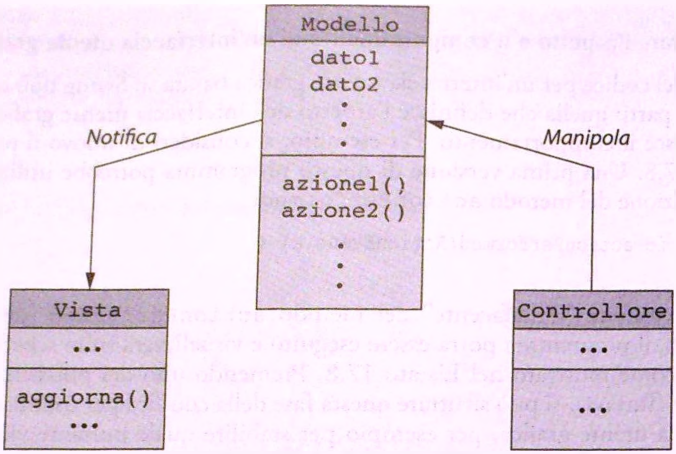


Figura 17.9 Il pattern Model-View-Controller.

Utilizzare il metodo `setActionCommand`

Come accennato precedentemente, la pressione di un pulsante provoca la generazione di un evento di tipo azione. Normalmente, questo evento viene passato come argomento al metodo `actionPerformed` degli ascoltatori per quel pulsante. Quando riceve un evento, il metodo `actionPerformed` deve scoprire quale pulsante lo ha generato. Nella discussione del Listato 17.8 si è detto che la chiamata `e.getActionCommand()` restituisce la stringa scritta sul pulsante. In realtà, è stata fatta una piccola semplificazione. L'invocazione di `e.getActionCommand()`, infatti, restituisce una stringa nota come il comando associato all'azione del pulsante. Il comando di default è la stringa scritta sul pulsante, ma volendo si può specificare una stringa diversa.

Per esempio, nel Listato 17.9 il pulsante `pulsanteStop` è stato creato nel modo seguente:

```
JButton pulsanteStop = new JButton("Rosso");
```

Se non si fa nient'altro, il comando per il pulsante `pulsanteStop` sarà `"Rosso"`. Tuttavia, si potrebbe cambiare tale comando in qualche altra stringa, come `"Stop"`. Per fare ciò, si dovrebbe agire come segue:

```
pulsanteStop.setActionCommand("Stop");
```

Il metodo `actionPerformed` dovrebbe a questo punto cercare la stringa `"Stop"` al posto della stringa `"Rosso"`.

Ci si potrebbe ritrovare in una situazione nella quale si vuole che la stessa stringa compaia su due pulsanti diversi. In un caso del genere, è possibile distinguere i due pulsanti utilizzando `setActionCommand` per assegnare loro comandi diversi.

`setActionCommand` e `getActionCommand`

A ogni pulsante e, più in generale, a ogni elemento che possa generare eventi di tipo azione, è associata una stringa nota come comando associato all'azione. Quando il pulsante viene premuto, viene generato un evento di tipo azione. La seguente chiamata restituisce il comando associato al pulsante che ha generato l'evento `e`:

```
e.getActionCommand()
```

Normalmente, il metodo `actionPerformed` utilizzerà questo comando per determinare quale pulsante è stato premuto.

Il comando di default associato a un pulsante è la stringa visualizzata su di esso, ma se si vuole si può cambiare il comando mediante una chiamata al metodo `setActionCommand`. Per esempio, il codice che segue scriverà la stringa `"Rosso"` sul pulsante `pulsanteStop`, ma imposterà a `"Stop"` il comando associato al pulsante.

Esempio

```
JButton pulsanteStop = new JButton("Rosso");
pulsanteStop.setActionCommand("Stop");
```

FAQ Dov'è salvato il comando associato all'azione?

Il comando associato all'azione di un pulsante è conservato in una variabile di istanza privata del pulsante e in una variabile di istanza privata di ogni evento generato dal pulsante. Quindi i metodi `getActionCommand` e `setActionCommand` sono rispettivamente, a tutti gli effetti, dei metodi *get* e *set*.

17.4 Approfondimenti su finestre ed eventi

17.4.1 L'interfaccia `WindowListener`

Nel paragrafo precedente, quando sono stati inseriti dei pulsanti in una finestra, la classe finestra stessa è stata scelta come ascoltatore degli eventi generati dai pulsanti. D'altra parte, per gestire gli eventi generati dalla chiusura di una finestra era stata utilizzata una classe esterna, denominata `DistruttoreFinestra`. In questo paragrafo si mostrerà come far sì che sia la finestra stessa a gestire anche gli eventi da essa generati.

Per fare di una finestra il gestore degli eventi generati dai suoi pulsanti, si è richiesto che la classe corrispondente implementasse l'interfaccia `ActionListener`. Per esempio, nel Listato 17.8 la classe `PulsantiDemo` è stata trasformata in un *action listener* definendola nel modo seguente:

```
public class DemoPulsanti extends JFrame implements ActionListener
```

Esiste un'interfaccia simile anche per gli eventi generati dalle finestre, denominata `WindowListener`. Per rendere la classe `DemoPulsanti` un gestore di eventi generati da finestre anziché di eventi generati da pulsanti, la definizione dovrebbe essere

```
public class PulsantiDemo extends JFrame implements WindowListener
```

Se, com'è più probabile, si vuole che la classe possa gestire sia gli eventi generati dai pulsanti che quelli generati dalle finestre, la si può definire come segue:

```
public class PulsantiDemo extends JFrame
                                  implements ActionListener, WindowListener
```

Si noti che quando una classe implementa più interfacce, la parola chiave `implements` deve essere specificata una volta sola, con i nomi delle interfacce separati da virgole.

Il motivo per il quale finora si è evitato di utilizzare una classe finestra come gestore degli eventi generati dalla finestra stessa è che quando una classe implementa un'interfaccia, come `ActionListener` o `WindowListener`, deve includere le definizioni di *tutti* i metodi richiesti dall'interfaccia. L'interfaccia `ActionListener` ha solo il metodo `actionPerformed`, quindi implementare questa interfaccia non comporta alcun carico di lavoro aggiuntivo. L'interfaccia `WindowListener`, al contrario, dichiara sette metodi, che sono compresi tra i dieci presentati nella Figura 17.3 per la classe `WindowAdapter`. Quindi, se una classe implementa l'interfaccia `WindowListener`, deve fornire definizioni per tutti questi metodi anche se non tutti vengono effettivamente utilizzati. In ogni caso, si può sempre fornire una definizione con corpo vuoto per i metodi che non servono, come in questo esempio:

```
public void windowDeiconified(WindowEvent e) {
}
```

Se si utilizzano solo pochi metodi dell'interfaccia `WindowListener`, questo vincolo è fastidioso, anche se non è difficile da rispettare.

Ci sono però anche dei vantaggi nell'utilizzare l'interfaccia `WindowListener`. Per esempio, se la classe estende `JFrame` e implementa `WindowListener`, è più semplice chiamare un metodo della classe finestra all'interno della classe che gestisce gli eventi, dato che si tratta della stessa classe.

Per esempio, il programma riportato nel Listato 17.9 è una variante di quello del Listato 17.8, nel quale la classe interfaccia utente grafica fa da gestore dei propri eventi. Ciò permette di utilizzare la seguente istruzione nel metodo `windowClosing`:

```
this.dispose();
```

Il metodo `dispose` fa parte della classe `JFrame` ed è quindi ereditato dalla classe `WindowListenerDemo`. Questo metodo libera tutte le risorse eventualmente utilizzate dalla finestra. In un programma semplice come quello riportato, questa operazione non è realmente necessaria, dato che tutte le risorse utilizzate sarebbero comunque liberate automaticamente al termine dell'esecuzione del programma. Se però il programma avesse più finestre e dovesse chiuderne solo una, proseguendo comunque l'esecuzione, sarebbe utile poter chiamare `dispose`, anziché `System.exit`. Nel Listato 17.9 il metodo `dispose` è stato utilizzato solo a titolo dimostrativo. Nel paragrafo successivo sarà utilizzato in un esempio più realistico.

FAQ Perché considerare sia l'interfaccia `WindowListener` che la classe `WindowAdapter`? Dopo tutto, dichiarano o definiscono gli stessi metodi.

La classe `WindowAdapter` è fornita unicamente come comodità per il programmatore. È una classe che implementa tutti i metodi dell'interfaccia `WindowListener` dando a ognuno di essi un corpo vuoto. In questo modo, quando si definisce una classe derivata da `WindowAdapter`, non occorre includere definizioni vuote. Ma allora perché non si può utilizzare sempre `WindowAdapter`? Il fatto è che, spesso, si vuole che una classe ne estenda qualcun'altra, come `JFrame`, oltre a essere un gestore di eventi, e una classe non può estenderne due, come `JFrame` e `WindowAdapter`. In questo caso, si fa derivare la classe da `JFrame` e le si fa implementare l'interfaccia `WindowListener`. Una classe, infatti, può essere derivata da un'unica altra classe, ma allo stesso tempo può implementare una o più interfacce.

L'interfaccia `WindowListener`

Per fare in modo che una classe possa gestire gli eventi generati da una finestra, la si può far derivare dalla classe `WindowAdapter` o farle implementare l'interfaccia `WindowListener`. Un esempio di classe che implementa questa interfaccia è `JFrame`, che gestisce gli eventi generati dal proprio pulsante di chiusura. Tuttavia, qualunque classe può implementare l'interfaccia `ActionListener`, anche se non estende `JFrame`.

Affinché una classe implementi l'interfaccia `WindowListener`, è necessario aggiungere la specifica

```
implements WindowListener
```

nell'intestazione della classe, dopo l'eventuale indicazione della classe da estendere. Inoltre, la classe deve implementare tutti i sette metodi dichiarati da `WindowListener`.

Il metodo `dispose`

La classe `JFrame` ha un metodo, denominato `dispose`, che elimina la finestra senza terminare l'esecuzione del programma. Quando il metodo viene invocato, le risorse precedentemente utilizzate dalla finestra vengono liberate in modo che possano essere riutilizzate, così che la finestra viene chiusa, ma l'esecuzione del programma prosegue. Il metodo `dispose` è utilizzato spesso per eliminare una o più finestre di un programma senza terminarne l'esecuzione.

Sintassi

oggetto_JFrame`.dispose();`

L'oggetto referenziato da *oggetto_JFrame* è spesso un `this` implicito. Un esempio completo di utilizzo del metodo `dispose` è riportato nel Listato 17.9.

MyLab

LISTATO 17.9 Un gestore di eventi generati da finestre.

Questa classe non necessita della classe `DistruttoreFinestra` o di altre classi simili.

```
import javax.swing.JButton;
import javax.swing.JFrame;
import java.awt.Color;
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

public class WindowListenerDemo extends JFrame
                    implements ActionListener, WindowListener {
    public static final int LARGHEZZA = 300;
    public static final int ALTEZZA = 200;

    public WindowListenerDemo() {
        setSize(LARGHEZZA, ALTEZZA);

        addWindowListener(this);

        setTitle("Esempio Window Listener");
        Container pannello = getContentPane();
        pannello.setBackground(Color.BLUE);

        setLayout(new FlowLayout());

        JButton pulsanteStop = new JButton("Rosso");
        pulsanteStop.addActionListener(this);
        add(pulsanteStop);

        JButton pulsanteVia = new JButton("Verde");
        pulsanteVia.addActionListener(this);
        add(pulsanteVia);
    }

    public void actionPerformed(ActionEvent e) {
        Container pannello = getContentPane();

        if (e.getActionCommand().equals("Rosso"))
            pannello.setBackground(Color.RED);
        else if (e.getActionCommand().equals("Verde"))
            pannello.setBackground(Color.GREEN);
        else
            System.out.println("Errore nell'interfaccia.");
    }
```

```
    public void windowClosing(WindowEvent e) {
        this.dispose();
        System.exit(0);
    }

    public void windowOpened(WindowEvent e) {}

    public void windowClosed(WindowEvent e) {}

    public void windowIconified(WindowEvent e) {}

    public void windowDeiconified(WindowEvent e) {}

    public void windowActivated(WindowEvent e) {}

    public void windowDeactivated(WindowEvent e) {}

    public static void main(String args[]) {
        WindowListenerDemo gui = new WindowListenerDemo();
        gui.setVisible(true);
    }
}
```

Output sullo schermo

17.4.2 Programmare il pulsante di *chiusura*

È possibile programmare il pulsante di chiusura di una finestra utilizzando l'interfaccia `WindowListener`, come nel Listato 17.9, o sfruttando una classe esterna, come `DistruttoreFinestra`. Tuttavia, a volte si vuole ottenere un comportamento diverso in risposta alla pressione del pulsante di chiusura.

Se si vuole che in seguito alla pressione del pulsante di chiusura di un `JFrame` il programma esegua qualcosa di diverso dall'eliminare la finestra, il costruttore della classe deve chiamare il metodo `setDefaultCloseOperation`, come nell'esempio seguente:

```
setDafultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
```

Questo metodo fa proprio ciò che il suo nome suggerisce: imposta il comportamento di default da tenere in seguito alla pressione del tasto di chiusura in base all'argomento specificato. Tale argomento deve essere uno dei quattro valori statici di tipo `int` definiti nell'interfaccia `WindowConstants` della libreria Swing. Si ricordi, infatti, che un'interfaccia può definire costanti pubbliche oltre a dichiarare metodi pubblici. Questa particolare interfaccia definisce solo delle costanti. Poiché l'interfaccia non dichiara metodi, non occorre specificare la clausola `implements` nell'intestazione della classe.

Se non lo si cambia tramite il metodo `setDefaultCloseOperation`, il comportamento di default prevede che la finestra venga chiusa, ma che l'esecuzione del programma prosegua. Si noti che la ridefinizione del metodo `windowClosing` non modifica il comportamento predefinito. Se si ridefinisce il metodo `windowClosing` senza chiamare nel costruttore il metodo `setDefaultCloseOperation`, quando l'utente preme il pulsante di chiusura, la finestra farà effettivamente quanto specificato nel metodo `windowClosing`, ma scomparirà anche. Se non si vuole che la finestra venga chiusa, bisogna modificare il comportamento predefinito con una chiamata a `setDefaultCloseOperation`.

La classe `ChiusuraFinestraDemo` del Listato 17.10 chiama il metodo `setDefaultCloseOperation` e programma il pulsante di chiusura in modo che quando l'utente lo preme compaia una seconda finestra che chiede all'utente se vuole terminare il programma. Se l'utente preme il pulsante `No`, la seconda finestra scompare, ma la prima rimane visibile. Se invece l'utente preme il pulsante `Sì`, entrambe le finestre vengono chiuse e il programma termina. È utile analizzare alcuni dettagli di questa classe (per una migliore disposizione degli elementi, la classe utilizza la classe `JPanel` della libreria Swing, che sarà discussa nel Paragrafo 17.5; per il momento, i dettagli dell'utilizzo di `JPanel` possono essere ignorati per concentrarsi sulla parte relativa agli eventi generati dalla finestra e dai pulsanti).

Oltre a invocare il metodo `setDefaultCloseOperation`, il costruttore della classe `ChiusuraFinestraDemo` registra un oggetto della classe `AltroDistruttoreFinestra` come gestore degli eventi della finestra, nel modo seguente:

```
addWindowListener(new AltroDistruttoreFinestra());
```

Quando l'utente preme il pulsante di chiusura della finestra principale `ChiusuraFinestraDemo`, viene eseguito il seguente metodo `windowClosing`, definito nella classe `AltroDistruttoreFinestra`:

```
public void windowClosing(WindowEvent e) {
    FinestraConferma finestraChiedi = new FinestraConferma();
    finestraChiedi.setVisible(true);
}
```

Il metodo crea un oggetto della classe `FinestraConferma`, chiamandolo `finestraChiedi`, e lo rende visibile.

Il comportamento della finestra `finestraChiedi` è definito dal metodo `actionPerformed` della classe `FinestraConferma`. Il codice rilevante è il seguente:

```
if (e.getActionCommand().equals("Sì"))
    System.exit(0);
else if (e.getActionCommand().equals("No"))
    dispose(); // Elimina solo la finestra di conferma
```

Se l'utente preme il pulsante Sì, il programma termina ed entrambe le finestre vengono chiuse. Se invece l'utente preme il pulsante No, la chiamata al metodo dispose elimina la seconda finestra, ma la prima rimane visualizzata.

LISTATO 17.10 Programmare il pulsante di chiusura di una finestra.

Per semplicità, le classi sono riportate nello stesso listato.

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.WindowConstants;
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

/**
 * Questa classe gestirà gli eventi della finestra principale.
 */
public class AltroDistruttoreFinestra extends WindowAdapter {
    /**
     * Mostra una finestra che chiede conferma all'utente.
     */
    public void windowClosing(WindowEvent e) {
        FinestraConferma finestraChiedi = new FinestraConferma();
        finestraChiedi.setVisible(true);
    }
}

/**
 * Questa classe costruisce una finestra per chiedere conferma all'utente.
 */
public class FinestraConferma extends JFrame implements ActionListener {
    public static final int LARGHEZZA = 200;
    public static final int ALTEZZA = 100;

    public FinestraConferma() {
        setSize(LARGHEZZA, ALTEZZA);

        getContentPane().setBackground(Color.WHITE);
        setLayout(new BorderLayout());

        JLabel messaggio = new JLabel("Sei sicuro di voler uscire?");
        add(messaggio, BorderLayout.CENTER);

        JPanel pannelloPulsanti = new JPanel();
        setLayout(new FlowLayout());
```

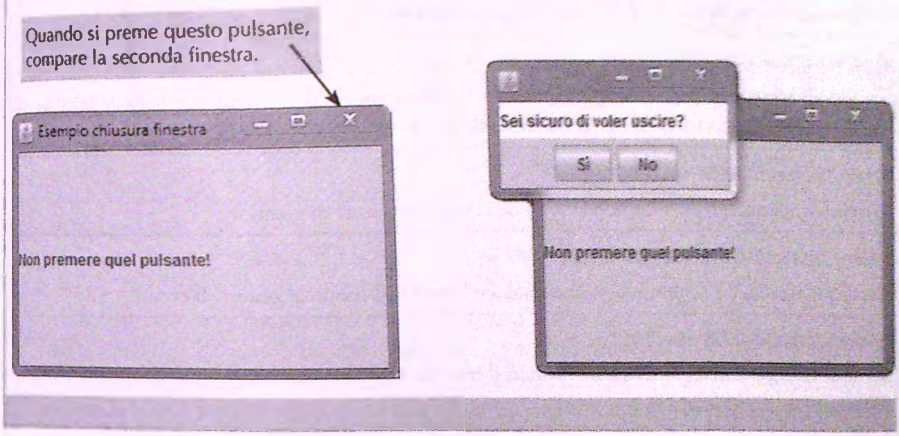
MyLab

```
        JButton pulsanteEsci = new JButton("Sì");
        pulsanteEsci.addActionListener(this);
        pannelloPulsanti.add(pulsanteEsci);

        JButton pulsanteAnnulla = new JButton("No");
        pulsanteAnnulla.addActionListener(this);
        pannelloPulsanti.add(pulsanteAnnulla);

        add(pannelloPulsanti, BorderLayout.SOUTH);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("Sì"))
            System.exit(0);
        else if (e.getActionCommand().equals("No"))
            dispose(); // Elimina solo la finestra di conferma
        else
            System.out.println("Errore nell'interfaccia.");
    }
}
```

MyLab

```
/**
 * Esempio di programmazione del pulsante di chiusura di una finestra.
 */
public class ChiusuraFinestraDemo extends JFrame {
    public static final int LARGHEZZA = 300;
    public static final int ALTEZZA = 200;

    public ChiusuraFinestraDemo() {
        setSize(LARGHEZZA, ALTEZZA);

        setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
        addWindowListener(new AltroDistruttoreFinestra());

        setTitle("Esempio chiusura finestra");
        setLayout(new BorderLayout());

        JLabel messaggio = new JLabel("Non premere quel pulsante!");
        add(messaggio, BorderLayout.CENTER);
    }

    public static void main(String args[]) {
        ChiusuraFinestraDemo gui = new ChiusuraFinestraDemo ();
        gui.setVisible(true);
    }
}
```

Output sullo schermo



Programmare il pulsante di chiusura di una finestra

Se si vuole programmare il pulsante di chiusura di una finestra in modo che faccia qualcosa di più complicato che semplicemente terminare il programma o chiudere la finestra, occorre aggiungere la seguente istruzione al costruttore della classe finestra o in qualche altro punto equivalente:

```
setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
```

Altri dettagli sul metodo `setDefaultCloseOperation`

Come argomento al metodo `setDefaultCloseOperation` è possibile utilizzare una qualunque delle costanti definite dall'interfaccia `WindowConstants`, riportate nella Figura 17.10. In alcuni casi, la chiamata al metodo è sufficiente per programmare completamente il comportamento del pulsante di chiusura. Se il metodo non viene chiamato, il comportamento predefinito è quello corrispondente alla costante `WindowConstants.HIDE_ON_CLOSE`.

Quasi tutte le interfacce utente grafiche Swing presentate finora potrebbero essere riprogrammate eliminando il gestore di eventi e ottenendo comunque il comportamento desiderato per il pulsante di chiusura per mezzo della seguente chiamata:

```
setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
```

Questa chiamata sostituirebbe l'istruzione

```
addWindowListener(new DistruttoreFinestra());
```

Tuttavia, la tecnica che utilizza un gestore di eventi è più potente, perché in generale consente di effettuare operazioni più complicate.

Le seguenti costanti sono utilizzate con il metodo `setDefaultCloseOperation` per specificare il comportamento predefinito da tenere in seguito alla pressione del tasto di chiusura di una finestra.
`WindowConstants.DO_NOTHING_ON_CLOSE` Specifica di non fare nulla. Il programmatore dovrà includere ogni azione desiderata nella definizione del metodo `windowClosing` dell'oggetto `WindowListener` registrato alla finestra.
`WindowConstants.HIDE_ON_CLOSE` Specifica di nascondere la finestra dopo l'invio dell'evento al gestore di eventi.
`WindowConstants.DISPOSE_ON_CLOSE` Specifica di nascondere ed eliminare la finestra dopo l'invio dell'evento al gestore di eventi.
`WindowConstants.EXIT_ON_CLOSE` Specifica di terminare l'applicazione chiamando il metodo `System.exit`. È equivalente alla costante `JFrame.EXIT_ON_CLOSE`.

Figura 17.10 Le costanti dell'interfaccia `WindowConstants`.

17.5 Classi contenitore

Quando si utilizza la libreria Swing per creare un'interfaccia utente grafica come quelle degli esempi visti finora, si creano classi a partire da altre classi già esistenti. Ci sono essenzialmente due modi per fare ciò. Uno è quello di sfruttare l'ereditarietà. Per esempio, per costruire una finestra solitamente si crea una classe estendendo la classe `JFrame`. Il secondo modo consiste nell'utilizzare una delle classi Swing come **contenitore** nel quale posizionare gli elementi. Per esempio, nella classe `PulsantiDemo` del Listato 17.10 sono stati aggiunti dei pulsanti a una finestra nel modo seguente:

```
JPanel pannelloPulsanti = new JPanel();
...
JButton pulsanteEsci = new JButton("Sì");
...
pannelloPulsanti.add(pulsanteEsci);
```

Non occorre scegliere uno solo dei due modi per costruire un'interfaccia utente grafica. Nella maggior parte dei casi, si sfrutteranno entrambe le tecniche.

In questo paragrafo verrà analizzata la classe `JPanel`, spesso utilizzata per definire parti di una finestra. In seguito, si considereranno le proprietà generali delle classi contenitore come `JPanel`.

17.5.1 La classe `JPanel`

Un'interfaccia utente grafica è solitamente organizzata secondo una struttura gerarchica, con contenitori, simili a finestre, a loro volta inseriti in altri contenitori. In questo paragrafo si introdurrà una nuova classe che semplifica la creazione di strutture di questo tipo. La classe `JPanel` è una classe di tipo contenitore molto semplice, che fa poco più che raggruppare oggetti. Nonostante sia una classe dal comportamento semplice, la si utiliz-

zerà molto spesso. Un oggetto `JPanel` ha un ruolo analogo alle parentesi che si usano per combinare più istruzioni Java in un'unica istruzione composta. La classe `JPanel` consente infatti di raggruppare oggetti più piccoli, come pulsanti ed etichette, in elementi più grandi, cioè gli oggetti `JPanel`, che possono poi essere aggiunti a una finestra `JFrame`. Quindi, una delle funzioni principali degli oggetti `JPanel` è quella di suddividere un `JFrame` in più parti. Gli oggetti `JPanel` sono anche detti, semplicemente, **pannelli**.

L'utilizzo ragionato dei pannelli può influire sulla disposizione degli elementi di una finestra tanto quanto la scelta di un gestore di layout. Per esempio, quando si usa un gestore di tipo `BorderLayout`, si possono posizionare gli elementi nelle cinque regioni nord, sud, est, ovest e centro. Ma come si potrebbe fare per posizionare due elementi nella parte bassa della finestra? Si potrebbero inserire i due elementi in un pannello e posizionare quest'ultimo nella posizione `BorderLayout.SOUTH`.

Il Listato 17.11 contiene una variante del programma del Listato 17.8 nella quale i pulsanti sono stati inseriti in un pannello, denominato `pannelloPulsanti`, in modo che la porzione di finestra che ospita i pulsanti non cambi colore con il resto della finestra. Come accadeva con la versione originale del programma, quando si preme il pulsante `Rosso`, il colore della finestra diventa il rosso, mentre diventa il verde quando si preme il pulsante `Verde`. In questo caso, però, i pulsanti sono in un pannello separato bianco, che non cambia colore. Come si può vedere, con un oggetto `JPanel` si utilizzano un gestore di layout e il metodo `add` esattamente come con un `JFrame`. Per prima cosa, si usa il metodo `add` per posizionare i pulsanti nel pannello, come nell'esempio che segue:

```
pannelloPulsanti.add(pulsanteStop);
```

Poi si usa il metodo `add` per inserire il pannello nella finestra, in questo modo:

```
add(pannelloPulsanti, BorderLayout.SOUTH);
```

Si noti che la finestra e il pannello hanno ognuno il proprio gestore di layout.

Si noti come vengono impostati i gestori degli eventi. Ogni pulsante registra `this` come gestore, come nell'istruzione che segue:

```
pulsanteStop.addActionListener(this);
```

Poiché l'istruzione compare nel costruttore della classe `PannelloDemo`, il `this` si riferisce all'oggetto corrente della classe `PannelloDemo`, che corrisponde all'intera interfaccia utente grafica. Quindi è l'intera finestra a fare da gestore degli eventi dei pulsanti, e non il pannello che li contiene. Quindi, quando si preme uno dei pulsanti è solo lo sfondo della finestra che cambia colore. Il pannello che contiene i pulsanti rimane bianco. Inoltre, è la classe `PannelloDemo` che implementa l'interfaccia `ActionListener` e definisce il metodo `actionPerformed`.

Il Listato 17.11 introduce anche un'altra semplice tecnica. Oltre a modificare i colori del pannello e della finestra, è stato dato un colore anche ai pulsanti. Per fare questo, è stato utilizzato il metodo `setBackground`, in modo analogo agli esempi precedenti. Tramite il metodo `setBackground` si può assegnare un colore di sfondo a quasi qualunque elemento di un'interfaccia utente grafica.

Si può utilizzare il metodo `add` per aggiungere un pannello a un altro pannello e ognuno di essi può avere un gestore di layout diverso. Costruendo una struttura gerarchica di pannelli in questo modo, si può ottenere praticamente qualunque disposizione desiderata degli elementi dell'interfaccia utente grafica.

La classe JPanel

Un oggetto `JPanel` (cioè un pannello) raggruppa oggetti più piccoli, come pulsanti ed etichette, in elementi più grandi. Il pannello è poi tipicamente aggiunto a una finestra `JFrame` o a un altro pannello. Quindi, una delle funzioni principali dei pannelli è quella di dividere una finestra in più aree.

Un pannello non ha un pannello del contenuto

Diversamente da una finestra, un pannello non ha un pannello del contenuto che ne occupa interamente la parte interna. Pertanto, è possibile cambiare il colore di sfondo di un pannello direttamente tramite il metodo `setBackground`.

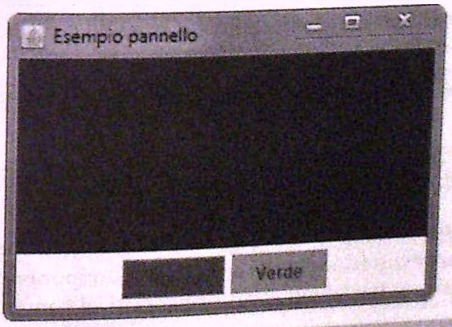
MyLab

LISTATO 17.11 Aggiunta di pulsanti a un pannello.

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import java.awt.Color;
import java.awt.Container;
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * Un semplice esempio dell'aggiunta di pulsanti ad un pannello.
 */
public class PannelloDemo extends JFrame implements ActionListener {
   public static final int LARGHEZZA = 300;
   public static final int ALTEZZA = 200;

   public PannelloDemo() {
      setSize(LARGHEZZA, ALTEZZA);
      addWindowListener(new DistruttoreFinestra());
      setTitle("Esempio pannello");
      Container pannelloContenuto = getContentPane();
      pannelloContenuto.setBackground(Color.BLUE);
      setLayout(new BorderLayout());

      JPanel pannelloPulsanti = new JPanel();
      pannelloPulsanti.setBackground(Color.WHITE);
```

```
        pannelloPulsanti.setLayout(new FlowLayout());

        JButton pulsanteStop = new JButton("Rosso");
        pulsanteStop.setBackground(Color.RED);
        pulsanteStop.addActionListener(this);
        pannelloPulsanti.add(pulsanteStop);

        JButton pulsanteVia = new JButton("Verde");
        pulsanteVia.setBackground(Color.GREEN);
        pulsanteVia.addActionListener(this);
        pannelloPulsanti.add(pulsanteVia);

        add(pannelloPulsanti, BorderLayout.SOUTH);
    }

    public void actionPerformed(ActionEvent e) {
        Container pannello = getContentPane();

        if (e.getActionCommand().equals("Rosso"))
            pannello.setBackground(Color.RED);
        else if (e.getActionCommand().equals("Verde"))
            pannello.setBackground(Color.GREEN);
        else
            System.out.println("Errore nell'interfaccia.");
    }

    public static void main(String args[]) {
        PannelloDemo gui = new PannelloDemo();
        gui.setVisible(true);
    }
}
```

Output sullo schermo



17.5.2 La classe `Container`

Negli esempi precedenti è stata utilizzata la classe predefinita `Container`, che ora sarà analizzata in maggiore dettaglio. Una **classe contenitore** è una qualunque classe che discenda da `Container`. I contenitori (cioè gli oggetti di una classe contenitore) possono contenere altri elementi. La classe `JFrame` discende da `Container`, quindi qualunque classe derivata da `JFrame` può fare da contenitore di etichette, pulsanti, pannelli e altri elementi.

La Figura 17.11 mostra la gerarchia di una parte delle classi Swing e delle classi AWT. Si noti che la classe `Container` fa parte della libreria AWT, non di quella Swing. Ciò non rappresenta un problema, ma bisogna ricordarsi di utilizzare le istruzioni di importazione corrette. Le classi `Component`, `Frame` e `Window` sono altre classi AWT delle quali si potrebbe aver sentito parlare. Queste classi sono state incluse nella Figura 17.11 per completezza, ma non saranno utilizzate in questo testo.

In quanto discendente della classe `Container`, la classe `JComponent` è una classe contenitore. Quindi, qualunque oggetto `JComponent` può fare da contenitore per etichette, pulsanti e così via. In aggiunta, qualunque classe che discenda dalla classe `JComponent` è detta **classe componente**, mentre qualunque oggetto di una di queste classi componente è detto, semplicemente, **componente**. Qualunque componente può essere aggiunto a qualunque contenitore. In particolare, un oggetto di una classe componente è sia un contenitore che un componente, il che significa che si può aggiungere un componente a un altro componente. A volte, tuttavia, sarà preferibile evitarlo. Nel seguito si discuteranno i vari casi. Dato che la classe `JPanel` discende da `JComponent`, si può aggiungere un pannello a un contenitore, mentre ogni pannello, a sua volta, può contenere etichette, pulsanti o altri componenti, tra i quali altri pannelli.

Classi contenitore

Una classe contenitore è una qualunque classe che discenda dalla classe `Container`. In particolare, ogni classe che discenda da `JFrame` o da `JPanel` è una classe contenitore, perché queste due classi discendono da `Container`. A ogni classe contenitore possono essere aggiunti elementi come etichette, pulsanti, pannelli e così via. Gli elementi possono essere aggiunti utilizzando il metodo `add`.

Il metodo `add`

Ogni classe contenitore della libreria Swing ha un metodo `add` che può essere utilizzato per aggiungere componenti (cioè oggetti della classe `JComponent` o di una qualunque delle sue discendenti), sia predefiniti che personalizzati, al contenitore.

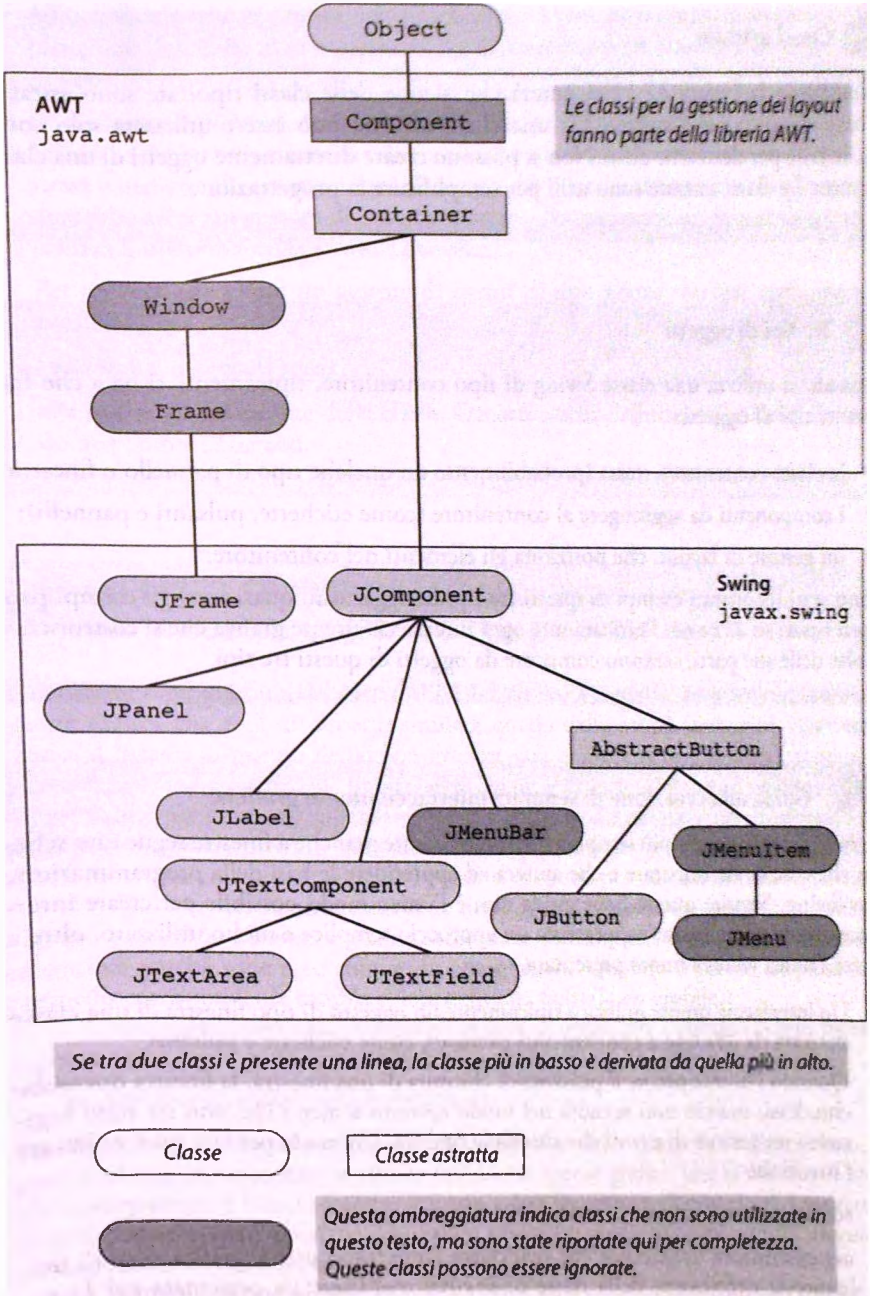


Figura 17.11 Gerarchia delle classi Swing.

Classi astratte

Guardando la Figura 17.11 si noterà che alcune delle classi riportate sono astratte. Come spiegato nel Capitolo 11, una classe astratta può essere utilizzata solo come classe base per derivarne altre. Non si possono creare direttamente oggetti di una classe astratta. Le classi astratte sono utili per semplificare la progettazione.

Tre tipi di oggetti

Quando si utilizza una classe Swing di tipo contenitore, tipicamente si ha a che fare con tre tipi di oggetti:

- la classe contenitore stessa (probabilmente un qualche tipo di pannello o finestra);
- i componenti da aggiungere al contenitore (come etichette, pulsanti e pannelli);
- un gestore di layout, che posiziona gli elementi nel contenitore.

Sono stati incontrati esempi di questi tre tipi di oggetto in quasi tutti gli esempi proposti basati su `JFrame`. Praticamente ogni interfaccia utente grafica che si costruisce, e molte delle sue parti, saranno composte da oggetti di questi tre tipi.

Guida alla creazione di semplici interfacce utente grafiche

La maggior parte delle più semplici interfacce utente grafiche a finestre segue uno schema che è facile da imparare e che aiuterà ad apprendere le basi della programmazione con Swing. Seguire queste linee guida non è l'unico modo possibile per creare interfacce utente grafiche, ma rappresenta un approccio semplice e molto utilizzato, oltre a essere l'unica tecnica finora presentata.

- Un'interfaccia utente grafica è tipicamente un oggetto di tipo finestra di una classe derivata da `JFrame` e contiene vari elementi, come etichette e pulsanti.
- Quando l'utente preme il pulsante di chiusura di una finestra, la finestra dovrebbe chiudersi, ma ciò non accadrà nel modo corretto a meno che non sia stato registrato un gestore di eventi che chiuda la finestra. Un modo per fare ciò è utilizzare l'istruzione

  ```
  addWindowListener(new DistruttoreFinestra());
  ```

  nel costruttore della classe che costituisce l'interfaccia utente grafica. Si può utilizzare la definizione della classe `DistruttoreFinestra` presentata nel Listato 17.2.
- Più componenti possono essere raggruppati tra loro posizionandoli in un pannello (un oggetto `JPanel`) e aggiungendo poi il pannello all'interfaccia utente grafica.

- All'interfaccia utente grafica (cioè la finestra `JFrame`) e a ogni pannello in essa contenuto dovrebbe essere assegnato un gestore di layout, tramite il metodo `setLayout`.
- Se ci sono componenti, come i pulsanti, che generano eventi di tipo azione, è necessario definire un gestore di tali eventi, che può essere l'interfaccia utente grafica stessa o qualche altra classe. Ogni elemento che possa generare eventi di tipo azione dovrebbe avere un gestore di eventi registrato. Per registrare un gestore di eventi si utilizza il metodo `addActionListener`.
- Per rendere una classe un gestore di eventi di tipo azione, occorre aggiungere la specifica

  ```
  implements ActionListener
  ```

  alla fine dell'intestazione della classe. Occorre anche definire, nella classe, il metodo `actionPerformed`.

ESEMPIO DI PROGRAMMAZIONE
UN TRICOLORE COSTRUITO CON PANNELLI

Si consideri il programma del Listato 17.12. Una volta eseguito, mostrerà un'interfaccia utente grafica che avrà un aspetto simile a quello della prima immagine riportata in fondo al listato: lo sfondo dell'intera finestra sarà grigio chiaro e saranno presenti tre pulsanti nella parte bassa della finestra denominati `"Verde"`, `"Bianco"` e `"Rosso"`. Se si preme uno dei pulsanti, appare una striscia verticale del colore corrispondente. I pulsanti possono essere premuti in qualunque ordine. Le altre tre viste riportate nel listato 17.12 mostrano cosa accade quando i pulsanti vengono premuti da sinistra verso destra.

Le strisce verde, bianca e rossa sono degli oggetti `JPanel` denominati `pannelloVerde`, `pannelloBianco` e `pannelloRosso`. Inizialmente i pannelli non sono distinguibili perché sono tutti colorati in grigio chiaro. Quando si preme un pulsante, il pannello corrispondente cambia colore e diventa quindi ben visibile.

Si noti come sono stati impostati i gestori di eventi per i pulsanti. Ogni pulsante registra `this` come gestore, come in questo esempio:

```
pulsanteRosso.addActionListener(this);
```

Poiché queste istruzioni compaiono nel costruttore della classe `TricoloreDemo`, è a questa classe, che rappresenta l'intera interfaccia utente grafica, che si riferisce il `this`. Di conseguenza, è l'intera finestra `JFrame` a fare da gestore di eventi per i pulsanti, e non il pannello che li contiene. Quando si preme un pulsante, viene quindi eseguito il metodo `actionPerformed` della classe `TricoloreDemo`.

Quando si preme un pulsante, viene invocato il metodo `actionPerformed`, che recupera il comando associato all'azione del pulsante utilizzando l'istruzione

```
String stringaPulsante = e.getActionCommand();
```

Il metodo `actionPerformed` usa poi un'istruzione `if-else` a più vie per determinare quale pulsante è stato premuto e cambiare colore al pannello giusto. L'approccio che utilizza istruzioni `if-else` a più vie è molto comune nella definizione dei metodi `actionPerformed`, anche se non è l'unico possibile.

Anche nel Listato 17.12, come già nel Listato 17.11, è stato utilizzato il metodo `setBackground` per assegnare un colore ai pulsanti.

MyLab

LISTATO 17.12 Un altro esempio di utilizzo di pulsanti e pannelli.

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.FlowLayout;
import java.awt.Color;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class TricoloreDemo extends JFrame implements ActionListener {
    public static final int LARGHEZZA = 300;
    public static final int ALTEZZA = 200;

    private JPanel pannelloVerde;
    private JPanel pannelloBianco;
    private JPanel pannelloRosso;

    public TricoloreDemo() {
        setSize(LARGHEZZA, ALTEZZA);
        addWindowListener(new DistruttoreFinestra());
        setTitle("Esempio tricolore");

        setLayout(new BorderLayout());

        JPanel pannelloGrande = new JPanel();
        pannelloGrande.setLayout(new GridLayout(1, 3));

        pannelloVerde = new JPanel();
        pannelloVerde.setBackground(Color.LIGHT_GRAY);
        pannelloGrande.add(pannelloVerde);

        pannelloBianco = new JPanel();
        pannelloBianco.setBackground(Color.LIGHT_GRAY);
        pannelloGrande.add(pannelloBianco);

        pannelloRosso = new JPanel();
        pannelloRosso.setBackground(Color.LIGHT_GRAY);
        pannelloGrande.add(pannelloRosso);

        add(pannelloGrande, BorderLayout.CENTER);
```

Oltre a essere una classe finestra, la classe `TricoloreDemo` è un gestore di eventi di tipo azione. Un oggetto di questa classe può quindi gestire gli eventi generati dai suoi pulsanti.

Queste sono state rese variabili di istanza perché è necessario utilizzarle sia nel costruttore che nel metodo `actionPerformed`.

```
        JPanel pannelloPulsanti = new JPanel();
        pannelloPulsanti.setBackground(Color.LIGHT_GRAY);
        pannelloPulsanti.setLayout(new FlowLayout());

        JButton pulsanteVerde = new JButton("Verde");
        pulsanteVerde.setBackground(Color.GREEN);
        pulsanteVerde.addActionListener(this);
        pannelloPulsanti.add(pulsanteVerde);

        JButton pulsanteBianco = new JButton("Bianco");
        pulsanteBianco.setBackground(Color.WHITE);
        pulsanteBianco.addActionListener(this);
        pannelloPulsanti.add(pulsanteBianco);

        JButton pulsanteRosso = new JButton("Rosso");
        pulsanteRosso.setBackground(Color.RED);
        pulsanteRosso.addActionListener(this);
        pannelloPulsanti.add(pulsanteRosso);

        add(pannelloPulsanti, BorderLayout.SOUTH);
    }

    public void actionPerformed(ActionEvent e) {
        String stringaPulsante = e.getActionCommand();

        if (stringaPulsante.equals("Verde"))
            pannelloVerde.setBackground(Color.GREEN);
        else if (stringaPulsante.equals("Bianco"))
            pannelloBianco.setBackground(Color.WHITE);
        else if (stringaPulsante.equals("Rosso"))
            pannelloRosso.setBackground(Color.RED);
        else
            System.out.println("Errore inatteso.");
    }

    public static void main(String args[]) {
        TricoloreDemo gui = new TricoloreDemo();
        gui.setVisible(true);
    }
}
```

È un oggetto della classe `TricoloreDemo` a fare da gestore degli eventi generati dai pulsanti che contiene.

Output sullo schermo (dopo la prima esecuzione)

Output sullo schermo (dopo aver fatto clic sul pulsante verde)

Output sullo schermo (dopo aver fatto clic sul pulsante bianco)

Output sullo schermo (dopo aver fatto clic sul pulsante rosso)

17.6 I/O di testo nelle interfacce utente grafiche

In questo paragrafo si mostrerà come gestire l'input e l'output di testo in un'interfaccia utente grafica Swing.

17.6.1 Aree e campi di testo

L'interfaccia utente grafica del Listato 17.7 contiene del testo, ma si tratta di testo statico: né l'utente né il programma lo possono modificare. Il Listato 17.13 contiene un programma che produce un'interfaccia utente grafica con un'**area di testo** (*text area*) nella quale l'utente può inserire del testo. Il programma consente all'utente di scrivere due note (o appunti) che possono essere salvate e recuperate in seguito. Questo esempio permette di gestire solo due note, ma è sufficiente per illustrare come si crea e si utilizza un'area di testo.

L'area al centro dell'interfaccia utente grafica, colorata in bianco, è un oggetto della classe `JTextArea`, nel quale l'utente può inserire del testo. Se l'utente preme il pulsante `Salva appunto 1`, il testo viene salvato come primo appunto. Se invece l'utente preme il pulsante `Salva appunto 2`, il testo è salvato come secondo appunto. Ognuno dei due appunti salvati può essere recuperato e riscritto nell'area di testo premendo, rispettivamente, i pulsanti `Carica appunto 1` e `Carica appunto 2`. Il testo presente nell'area di testo può essere cancellato premendo il pulsante `Cancella`. Si discuterà ora il modo in cui il programma realizza queste funzionalità.

I pulsanti creati nel Listato 17.13 sono inseriti in un pannello, che a sua volta è inserito nella finestra `JFrame`, esattamente come nel Listato 17.11. Analogamente, anche l'area di testo è inclusa in un pannello aggiunto alla finestra. Il codice che crea e posiziona l'area di testo è incluso nel costruttore della classe del Listato 17.13 ed è il seguente:

```
JPanel pannelloTesto = new JPanel();
pannelloTesto.setBackground(Color.BLUE);
testo = new JTextArea(RIGHE, CARATTERI_PER_RIGA);
testo.setBackground(Color.WHITE);
pannelloTesto.add(testo);
add(pannelloTesto, BorderLayout.CENTER);
```

Il pannello `pannelloTesto` è un oggetto della classe `JPanel`, mentre l'area di testo `testo` è un oggetto della classe `JTextArea`. Gli argomenti del costruttore di `JTextArea`, cioè le costanti `RIGHE` e `CARATTERI_PER_RIGA`, specificano, rispettivamente, il numero di righe di testo desiderate e il numero minimo di caratteri per riga. Se si inserisce un testo più lungo di quanto l'area di testo delle dimensioni specificate da questi due parametri possa contenere, il testo non sarà completamente visibile. Quindi, è opportuno utilizzare dei parametri abbastanza grandi da poter gestire la massima quantità attesa di testo (i problemi relativi a quantità eccessive di testo possono essere gestiti utilizzando il metodo `setLineWrap`, descritto in seguito, o utilizzando le barre di scorrimento, che però non saranno trattate in questo testo).

Il `pannelloTesto` è aggiunto alla finestra nella posizione `BorderLayout.CENTER`, utilizzando un gestore di tipo `BorderLayout`. La posizione centrale occupa sempre tutto lo spazio rimasto disponibile in una finestra. Questa parte dell'interfaccia utente grafica ne imposta l'aspetto, ma rimane da vedere come il testo inserito viene gestito.

I due appunti sono salvati nelle due variabili di istanza `appunto1` e `appunto2`, di tipo `String`. Quando l'utente preme il pulsante `Salva appunto 1`, il testo contenuto nell'area di testo viene salvato nella variabile `appunto1` per mezzo del seguente codice, che occupa le prime tre righe del metodo `actionPerformed`:

```
String comando = e.getActionCommand();
if (comando.equals("Salva appunto 1"))
    appunto1 = testo.getText();
```

Si ricordi che il metodo `getActionCommand` restituisce il testo scritto sul pulsante che è stato premuto. Il metodo `getText` restituisce il testo scritto nell'oggetto `testo` della classe `JTextArea`, cioè il testo inserito dall'utente. Il secondo appunto è salvato nello stesso modo.

La parte di codice che gestisce la visualizzazione degli appunti salvati nell'area di testo e la cancellazione del testo nell'area è la seguente:

```
else if (comando.equals("Cancella"))
    testo.setText("");
else if (comando.equals("Carica appunto 1"))
    testo.setText(appunto1);
else if (comando.equals("Carica appunto 2"))
    testo.setText(appunto2);
```

Il metodo `setText` della classe `JTextArea` cambia il testo nell'area di testo impostandolo alla stringa specificata come parametro. Nella seconda riga della parte di codice riportata sopra si passa al metodo `setText` una coppia di doppi apici senza altri caratteri in mezzo. Questa combinazione di caratteri costituisce una stringa vuota e ha l'effetto di eliminare il testo eventualmente presente nell'area di testo.

La classe `JTextField`, non utilizzata nel Listato 17.13, è molto simile alla classe `JTextArea`, a eccezione del fatto che la prima serve a creare un **campo di testo** (***text field***) per l'inserimento di una singola riga di testo. È utile quindi quando l'utente deve inserire pochi caratteri, come un numero o il nome di un file o di una persona. Si può specificare il numero di caratteri visibili in un campo di testo. L'utente può inserire un numero maggiore di caratteri, ma ne sarà visibile solo il numero specificato. Di conseguenza, è opportuno rendere un campo di testo più lungo di uno o due caratteri rispetto alla stringa più lunga che ci si aspetta possa essere inserita.

Sia per la classe `JTextArea` che per la classe `JTextField` è possibile specificare, in fase di costruzione, del testo che costituisca il contenuto iniziale del componente. Tale testo deve essere specificato sotto forma di una stringa passata come primo parametro al costruttore. Per esempio,

```
JTextField campoInputOutput = new JTextField("Inserire del testo:", 20);
```

Il campo di testo `campoInputOutput` avrà spazio per 20 caratteri visibili e sarà inizializzato con il testo `"Inserire del testo:"`.

Sia la classe `JTextArea` che la classe `JTextField` hanno un costruttore di default (cioè senza argomenti) che imposta il testo iniziale a `null` e le dimensioni a zero.

MyLab

LISTATO 17.13 Un'interfaccia utente grafica con un'area di testo.

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import java.awt.Color;
import java.awt.Container;
```

Un metodo `main` di esempio è incluso alla fine della definizione della classe.

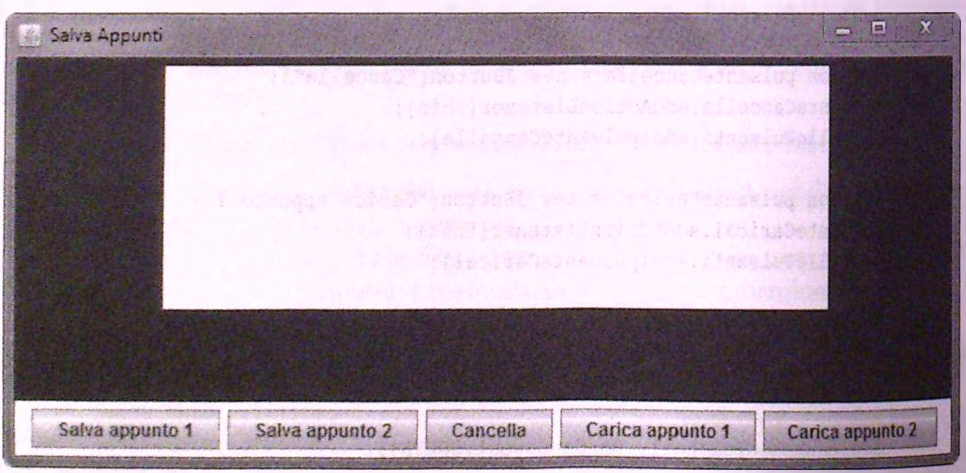
```
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class SalvaAppunti extends JFrame implements ActionListener {
   public static final int LARGHEZZA = 650;
   public static final int ALTEZZA = 300;
   public static final int RIGHE = 10;
   public static final int CARATTERI_PER_RIGA = 40;

   private JTextArea testo;
   private String appunto1 = "Nessun appunto 1.";
   private String appunto2 = "Nessun appunto 2.";

   public SalvaAppunti() {
      setSize(LARGHEZZA, ALTEZZA);
      addWindowListener(new DistruttoreFinestra());
      setTitle("Salva Appunti");
      setLayout(new BorderLayout());

      JPanel pannelloPulsanti = new JPanel();
      pannelloPulsanti.setBackground(Color.WHITE);
      pannelloPulsanti.setLayout(new FlowLayout());

      JButton pulsanteAppunto1 = new JButton("Salva appunto 1");
      pulsanteAppunto1.addActionListener(this);
      pannelloPulsanti.add(pulsanteAppunto1);

      JButton pulsanteAppunto2 = new JButton("Salva appunto 2");
      pulsanteAppunto2.addActionListener(this);
      pannelloPulsanti.add(pulsanteAppunto2);

      JButton pulsanteCancella = new JButton("Cancella");
      pulsanteCancella.addActionListener(this);
      pannelloPulsanti.add(pulsanteCancella);

      JButton pulsanteCarica1 = new JButton("Carica appunto 1");
      pulsanteCarica1.addActionListener(this);
      pannelloPulsanti.add(pulsanteCarica1);

      JButton pulsanteCarica2 = new JButton("Carica appunto 2");
      pulsanteCarica2.addActionListener(this);
      pannelloPulsanti.add(pulsanteCarica2);

      add(pannelloPulsanti, BorderLayout.SOUTH);

      JPanel pannelloTesto = new JPanel();
      pannelloTesto.setBackground(Color.BLUE);
```

Se si prova a recuperare un appunto prima di averlo salvato, si otterrà uno di questi errori.

```
        testo = new JTextArea(RIGHE, CARATTERI_PER_RIGA);
        testo.setBackground(Color.WHITE);
        pannelloTesto.add(testo);

        add(pannelloTesto, BorderLayout.CENTER);
    }

    public void actionPerformed(ActionEvent e) {
        String comando = e.getActionCommand();

        if (comando.equals("Salva appunto 1"))
            appunto1 = testo.getText();
        else if (comando.equals("Salva appunto 2"))
            appunto2 = testo.getText();
        else if (comando.equals("Cancella"))
            testo.setText("");
        else if (comando.equals("Carica appunto 1"))
            testo.setText(appunto1);
        else if (comando.equals("Carica appunto 2"))
            testo.setText(appunto2);
        else
            testo.setText("Errore nell'interfaccia.");
    }

    public static void main(String args[]) {
        SalvaAppunti gui = new SalvaAppunti();
        gui.setVisible(true);
    }
}
```

`testo` è una variabile di istanza.

Esempio di output sullo schermo



La classe `JTextArea`

La classe `JTextArea` può essere utilizzata per aggiungere a un'interfaccia utente grafica aree di testo modificabili. Un oggetto di questa classe è caratterizzato da dimensioni costituite da un numero di righe e da un numero di caratteri per riga specificati. In un'area di testo, si può inserire testo che occupi più spazio di quello disponibile, ma il testo in eccesso non sarà visibile. Inoltre, un'area di testo può essere inizializzata con del testo che viene visualizzato prima che l'utente inserisca del testo personalizzato.

Sintassi dei costruttori

```
JTextArea()
JTextArea(numero_righe, larghezza_riga)
JTextArea(testo_iniziale)
JTextArea(testo_iniziale, numero_righe, larghezza_riga)
```

Esempio

```
// Aggiunta di un'area di testo a un pannello
JPanel unPannello = new JPanel();
JTextArea testo = new JTextArea(10, 30);
unPannello.add(testo);
```

La classe `JTextField`

La classe `JTextField` può essere utilizzata per aggiungere a un'interfaccia utente grafica aree di testo modificabili costituite da un'unica riga. La riga potrà contenere un numero massimo specificato di caratteri. Se si inserisce un testo più lungo del numero massimo di caratteri, la parte in eccesso non sarà visibile.

Sintassi dei costruttori

```
JTextField()
JTextField(larghezza_riga)
JTextField(testo_iniziale)
JTextField(testo_iniziale, larghezza_riga)
```

Esempio

```
// Aggiunta di un campo di testo a un pannello
JPanel altroPannello = new JPanel();
JTextField nome = new JTextField(30);
altroPannello.add(nome);
```

Le dimensioni di un'area e di un campo di testo

Quando si crea un oggetto della classe `JTextArea` o della classe `JTextField`, si può specificare il numero massimo di caratteri per riga che verranno mostrati. Per esempio, le righe seguenti creano un'area e un campo di testo entrambi con 40 caratteri per riga:

```
JTextField campoTesto = new JTextField(40);
JTextArea areaTesto = new JTextArea(10, 40); // 10 righe
```

Il numero 40, in entrambi i casi, non rappresenta lo spazio occupato da 40 caratteri qualsiasi. In realtà, rappresenta uno spazio pari a 40 unità em. Un'**unità em** è lo spazio necessario per contenere una lettera *m*, che è la più larga dell'alfabeto italiano o inglese. Quindi, una riga di larghezza 40, come negli esempi riportati, sarà sempre in grado di ospitare almeno 40 caratteri e potrebbe riuscire a ospitarne anche di più, a seconda dell'effettivo spazio occupato da tali caratteri.

I metodi `getText` e `setText`

Entrambe le classi `JTextArea` e `JTextField` definiscono i metodi `getText` e `setText`. Il primo restituisce il testo contenuto nell'area o nel campo di testo, mentre il secondo modifica tale testo in base alla stringa specificata come parametro.

Esempio

```
// testo è un oggetto di una delle due classi JTExtArea e JTextField
appunto1 = testo.getText();           // leggi l'input dell'utente
testo.setText("Appunto 1 salvato."); // comunica un messaggio all'utente
```

Andare a capo nelle aree di testo

È possibile far sì che il testo in un'area di testo venga mandato o meno a capo automaticamente, quando una riga diventa troppo lunga, sfruttando il metodo `setLineWrap`. Questo metodo richiede un argomento di tipo `boolean`. Se l'argomento è `true` e l'utente inserisce più caratteri di quanti possano essere visualizzati su un'unica riga, i caratteri in più appariranno nella riga successiva. Se l'argomento è `false`, i caratteri in più rimarranno sulla stessa riga degli altri e non saranno visibili. Il secondo comportamento è quello predefinito se non si utilizza questo metodo.

Esempio

```
testo.setLineWrap(true);
```

Un'istruzione di questo tipo potrebbe essere inclusa nel costruttore del Listato 17.13.

Componenti di testo non modificabili

È anche possibile specificare che l'utente non possa scrivere in un'area o in un campo di testo. Per fare ciò, si utilizza il metodo `setEditable`, che è fornito sia dalla classe `JTextArea` che dalla classe `JTextField`. Se `testo` è un oggetto di una di queste due classi, l'istruzione

```
testo.setEditable(false);
```

impedirà all'utente di modificare il testo nel componente `testo`. Solo il programma sarà in grado di farlo. Anche se l'utente fa click con il mouse sul componente `testo` e poi scrive sulla tastiera, il testo nel componente non cambierà.

Per ripristinare la possibilità per l'utente di modificare il testo del componente, si utilizza lo stesso metodo con il parametro impostato a `true`, anziché a `false`, come segue:

```
testo.setEditable(true);
```

Se non si chiama il metodo `setEditable`, il comportamento predefinito prevede che l'utente possa cambiare il testo nel componente.

ESEMPIO DI PROGRAMMAZIONE
ETICHETTARE UN CAMPO DI TESTO

A volte può essere utile assegnare un'etichetta a un campo di testo. Per esempio, si supponga che l'interfaccia utente grafica chieda all'utente il nome e un numero identificativo per mezzo di due campi di testo. In un caso come questo, l'interfaccia utente grafica deve fornire delle etichette ai campi di testo in modo che l'utente sappia dove inserire i due dati. Gli oggetti della classe `JLabel` possono essere utilizzati per assegnare etichette ai campi di testo e a qualunque altro componente di un'interfaccia utente grafica.

Il Listato 17.14 contiene un programma che mostra come associare un'etichetta a un campo di testo. È sufficiente inserire il campo di testo e l'etichetta in un pannello e aggiungere poi quest'ultimo a un altro contenitore. La parte di codice evidenziata nel Listato 17.14 fa esattamente questo.

Il programma del Listato 17.14 serve solo come esempio e quindi non fa molto. Se l'utente inserisce un nome nel campo di testo e preme il pulsante `Prova`, l'interfaccia utente grafica fornisce un commento sul nome. Tutti i nomi ricevono però la stessa valutazione, cioè `Bel nome!` Quindi, se l'utente inserisce un nome e preme il pulsante `Prova`, l'interfaccia utente grafica assumerà l'aspetto riportato alla fine del Listato 17.14.

LISTATO 17.14 Assegnare un'etichetta a un campo di testo.

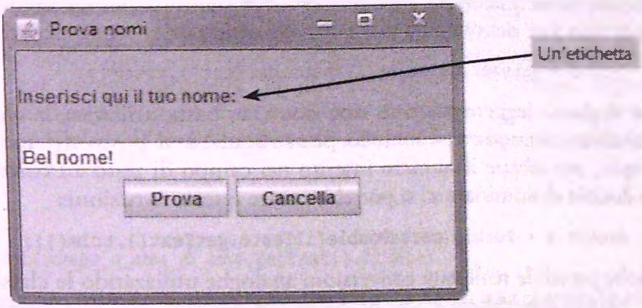
MyLab

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import java.awt.Color;
```

```
import java.awt.Container;
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * Esempio di associazione di un'etichetta ad un campo di testo.
 */
public class EtichettaDemo extends JFrame implements ActionListener {
    public static final int LARGHEZZA = 300;
    public static final int ALTEZZA = 200;
    private JTextField nome;

    public EtichettaDemo() {
        setSize(LARGHEZZA, ALTEZZA);
        addWindowListener(new DistruttoreFinestra());
        setTitle("Prova nomi");

        setLayout(new GridLayout(2, 1));

        JPanel pannelloNome = new JPanel();
        pannelloNome.setLayout(new BorderLayout());
        pannelloNome.setBackground(Color.LIGHT_GRAY);

        nome = new JTextField(20);
        pannelloNome.add(nome, BorderLayout.SOUTH);
        JLabel etichettaNome = new JLabel("Inserisci qui il tuo nome:");
        pannelloNome.add(etichettaNome, BorderLayout.CENTER);

        add(pannelloNome);

        JPanel pannelloPulsanti = new JPanel();
        pannelloPulsanti.setLayout(new FlowLayout());

        JButton pulsante = new JButton("Prova");
        pulsante.addActionListener(this);
        pannelloPulsanti.add(pulsante);

        pulsante = new JButton("Cancella");
        pulsante.addActionListener(this);
        pannelloPulsanti.add(pulsante);

        add(pannelloPulsanti);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("Prova"))
            nome.setText("Bel nome!");
```

```
        else if (e.getActionCommand().equals("Cancella"))
            nome.setText("");
        else
            System.out.println("Errore nell'interfaccia.");
    }

    public static void main(String args[]) {
        EtichettaDemo gui = new EtichettaDemo();
        gui.setVisible(true);
    }
}
```

Output sullo schermo



17.6.2 Input e output di numeri

Quando si vuole utilizzare un'interfaccia utente grafica basata su Swing per leggere dei numeri inseriti dall'utente, occorre convertire in numeri l'input in formato testuale. All'opposto, per mostrare dei numeri come output, è necessario convertire prima i numeri in stringhe e poi visualizzare queste ultime.

Per esempio, quando in un'interfaccia utente grafica si inserisce il valore `42` in un'area o in un campo di testo, l'interfaccia utente grafica leggerà la stringa `"42"`, non il numero `42`. Il programma deve quindi convertire la stringa `"42"` nel valore intero `42`. Se si vuole mostrare nell'interfaccia utente grafica il numero `43`, il programma deve prima convertire il valore intero `43` nella stringa `"43"`.

Si consideri per prima cosa l'input di numeri. Come si è detto, il programma deve convertire delle stringhe in numeri. Il metodo statico `parseInt` della classe `Integer` può svolgere questo compito nel caso dei numeri interi[1]. Per esempio,

```
Integer.parseInt("42")
```

restituisce l'intero `42`.

[1] Se questi dettagli non sono chiari, si riveda il Paragrafo 9.2.5 nel Capitolo 9.

L'input inserito dall'utente può essere recuperato utilizzando il metodo `getText`. Quindi, se per esempio il numero è stato inserito in un campo di testo chiamato `ilTesto`, l'espressione `ilTesto.getText()` restituirà l'input sotto forma di una stringa. Per convertire questa stringa in un numero intero, si può usare la seguente istruzione:

```
Integer.parseInt(ilTesto.getText())
```

Può succedere che l'utente includa degli spazi prima o dopo il numero; in questo caso, è opportuno aggiungere un'invocazione del metodo `trim` sulla stringa. Quindi, un modo più affidabile per ricavare il numero inserito come input sarebbe il seguente:

```
Integer.parseInt(ilTesto.getText().trim())
```

Una volta che il programma ha ottenuto il numero in questo modo, naturalmente lo può utilizzare come qualunque altro numero. Per esempio, per salvare il numero in una variabile di tipo `int` denominata `n` si potrebbe utilizzare la seguente istruzione:

```
int n = Integer.parseInt(ilTesto.getText().trim());
```

Se si vogliono leggere valori di tipo `double`, basta utilizzare la classe `Double` al posto della classe `Integer` e il metodo `parseDouble` al posto del metodo `parseInt`. Per esempio, per salvare il numero inserito nel campo di testo `ilTesto` in una variabile di tipo double denominata `x`, si potrebbe usare questa istruzione:

```
double x = Double.parseDouble(ilTesto.getText().trim());
```

È anche possibile realizzare conversioni analoghe utilizzando le classi `Long` e `Float`.

Non ci dovrebbero essere problemi nel comprendere e utilizzare espressioni come

```
Integer.parseInt(oggettoStringa.getText().trim());
```

Tuttavia, il codice sarà più semplice sia da leggere che da scrivere se si definisce un metodo per rendere questa operazione ancora più facile da eseguire. Un esempio di metodo di questo tipo è il seguente:

```
private static int daStringaAIntero(String oggettoStringa) {
    return Integer.parseInt(oggettoStringa.trim());
}
```

Con questo metodo, un'espressione come

```
n = Integer.parseInt(ilTesto.getText().trim());
```

potrebbe essere riscritta in modo più chiaro come

```
n = daStringaAIntero(ilTesto.getText());
```

Il metodo dovrebbe essere privato se, per la conversione, sarà utilizzato solo nella classe che definisce l'interfaccia utente grafica. Altrimenti, potrebbe essere incluso in una classe di servizio che raggruppi più metodi utili. In tal caso, avrebbe più senso rendere il metodo pubblico.

Per scrivere un numero in un campo o in un'area di testo si può utilizzare il metodo statico `toString`. Per esempio, si supponga che la variabile di tipo `int` `somma` contenga il valore `43`. Tale valore può essere convertito nella stringa `"43"` in questo modo:

```
Integer.toString(somma)
```

Quindi, per far comparire il valore della variabile `somma` nel campo di testo denominato `campoInputOutput`, si utilizzerebbe `setText` come segue:

```
campoInputOutput.setText(Integer.toString(somma));
```

In questo esempio è stato utilizzato il metodo `toString` della classe `Integer` perché la variabile da convertire era di tipo intero. Per una variabile di tipo `double totale`, si dovrebbe utilizzare

```
Double.toString(totale)
```

Queste tecniche saranno illustrate nel prossimo esempio di programmazione.

Leggere numeri da un'interfaccia utente grafica

La libreria Swing può essere usata per creare interfacce utente grafiche che consentano di fornire in input dei numeri mediante campi o aree di testo. L'esempio seguente restituirà il numero di tipo `int` che è stato inserito in un campo o in un'area di testo (nell'ipotesi che nel campo o nell'area di testo non sia stato scritto nient'altro all'infuori di cifre e spazi iniziali o finali):

Sintassi

```
Integer.parseInt(campo_o_area_di_testo.getText().trim())
```

Si può fare lo stesso per numeri di tipo `double`, `float` o `long` sostituendo la classe `Integer` rispettivamente con le classi `Double`, `Float` e `Long` ed utilizzando i metodi `parseDouble`, `parseFloat` e `parseLong` al posto del metodo `parseInt`.

Esempi

```
int n = Integer.parseInt(campoTesto.getText().trim());
double x = Double.parseDouble(campoTesto.getText().trim());
```

Queste istruzioni, un po' lunghe, possono anche essere racchiuse in metodi di servizio, come

```
private static double daStringaADouble(String oggettoStringa) {
    return Double.parseDouble(oggettoStringa.trim());
}
```

A seconda di dove questo metodo viene definito ed utilizzato, potrebbe essere meglio dichiararlo pubblico o privato. Con questo metodo, l'esempio precedente può essere semplificato in

```
double x = daStringaADouble(campoTesto.getText());
```

Scrivere numeri in un'interfaccia utente grafica

Un'interfaccia utente grafica basata su Swing può mostrare numeri in un campo o in un'area di testo tramite il metodo `setText`.

Sintassi

campo_o_area_di_testo`.setText(`*Classe_Wrapper*`.toString(`*variabile*`));`

Esempi

```
campoInputOutput.setText(Integer.toString(contatore));
campoInputOutput.setText(Long.toString(contatore));
campoInputOutput.setText(Double.toString(somma));
campoInputOutput.setText(Float.toString(somma));
```

ESEMPIO DI PROGRAMMAZIONE
UN'INTERFACCIA UTENTE GRAFICA PER LE ADDIZIONI

Il Listato 17.15 contiene un programma per eseguire delle addizioni. L'interfaccia utente grafica corrispondente è mostrata alla fine del listato. Il campo di testo contiene inizialmente il testo `Inserire qui i numeri`. L'utente può inserire un numero selezionando con il mouse questo testo e scrivendo poi il numero con la tastiera, in modo che il numero scritto diventi il contenuto del campo di testo. Quando l'utente preme il pulsante `Somma`, il numero nel campo di testo viene aggiunto al totale corrente, che viene poi mostrato nel campo di testo. L'utente può continuare a inserire e sommare numeri in questo modo finché lo desidera. Per azzerare il totale, può premere il pulsante `Azzera`.

Si noti che è la finestra stessa, che è un oggetto `JFrame`, a gestire gli eventi generati dai pulsanti. Ciò richiede che la classe implementi l'interfaccia `ActionListener`. La definizione della classe comincia quindi con

```
public class Sommatore extends JFrame implements ActionListener
```

Per consentire alla finestra di ricevere gli eventi generati dai suoi pulsanti, occorre registrare `this` come gestore di eventi per ogni pulsante. Per esempio, nel caso del pulsante `Somma` si scrive

```
JButton pulsanteSomma = new JButton("Somma");
pulsanteSomma.addActionListener(this);
```

La classe `Sommatore` deve inoltre definire il metodo `actionPerformed`, che determina come l'interfaccia utente grafica risponde alla pressione dei comandi. Come al solito, la parte fondamentale della definizione di questo metodo è un'istruzione condizionale a più vie che esegue operazioni diverse a seconda di quale pulsante è stato premuto. L'espressione `e.getActionCommand()` restituisce la stringa scritta sul pulsante che è stato premuto e questa stringa determina quale alternativa debba essere eseguita. Questa parte è evidenziata nel Listato 17.15.

Le operazioni necessarie per inizializzare i componenti e posizionarli nell'interfaccia utente grafica sono simili a quelle viste in precedenza.

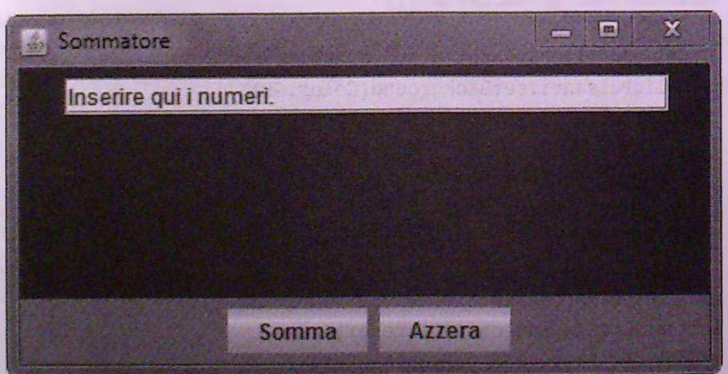
LISTATO 17.15 Un'interfaccia utente grafica per le addizioni.

MyLab

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;
import java.awt.Color;
import java.awt.Container;
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * Interfaccia per la somma di una serie di numeri.
 */
public class Sommatore extends JFrame implements ActionListener {
    public static final int LARGHEZZA = 400;
    public static final int ALTEZZA = 200;

    private JTextField campoInputOutput;
    private double somma = 0;

    public Sommatore() {
        setSize(LARGHEZZA, ALTEZZA);
        addWindowListener(new DistruttoreFinestra());
        setTitle("Sommatore");

        setLayout(new BorderLayout());

        JPanel pannelloPulsanti = new JPanel();
        pannelloPulsanti.setBackground(Color.GRAY);
        pannelloPulsanti.setLayout(new FlowLayout());

        JButton pulsanteSomma = new JButton("Somma");
        pulsanteSomma.addActionListener(this);
        pannelloPulsanti.add(pulsanteSomma);

        JButton pulsanteAzzera = new JButton("Azzera");
        pulsanteAzzera.addActionListener(this);
        pannelloPulsanti.add(pulsanteAzzera);

        add(pannelloPulsanti, BorderLayout.SOUTH);

        JPanel pannelloTesto = new JPanel();
        pannelloTesto.setBackground(Color.BLUE);
        pannelloTesto.setLayout(new FlowLayout());
```

```
        campoInputOutput = new JTextField("Inserire qui i numeri.", 30);
        campoInputOutput.setBackground(Color.WHITE);
        pannelloTesto.add(campoInputOutput);

        add(pannelloTesto, BorderLayout.CENTER);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("Somma")) {
            somma = somma + daStringaADouble(campoInputOutput.getText());
            campoInputOutput.setText(Double.toString(somma));
        } else if (e.getActionCommand().equals("Azzera")) {
            somma = 0;
            campoInputOutput.setText("0.0");
        } else
            campoInputOutput.setText("Errore nel codice.");
    }

    private static double daStringaADouble(String stringa) {
        return Double.parseDouble(stringa.trim());
    }

    public static void main(String args[]) {
        Sommatore gui = new Sommatore();
        gui.setVisible(true);
    }
}
```

Output sullo schermo



17.6.3 Gestire una `NumberFormatException`

Questo paragrafo utilizza dei concetti relativi alla gestione delle eccezioni presentati nel Capitolo 13. Se le eccezioni non sono ancora state studiate, si dovrebbe posticipare la lettura di questo paragrafo fino a quando non si sarà studiato il Capitolo 13.

L'interfaccia utente grafica del Listato 17.15 esegue delle addizioni sui numeri inseriti dall'utente. Quell'interfaccia utente grafica, però, presenta un problema. Se l'utente inserisce un numero in un formato non corretto, per esempio utilizzando la virgola al posto del punto decimale, uno dei metodi genererà un'eccezione di tipo `NumberFormatException`. In un programma basato su Swing, la generazione di un'eccezione non terminerà l'esecuzione del programma, ma lascerà l'interfaccia utente grafica in uno stato imprevedibile. Su alcuni sistemi l'utente potrebbe avere la possibilità di reinserire il numero, ma non si può fare affidamento sul fatto che ciò accada. Anche nel caso in cui ciò fosse possibile, l'utente potrebbe non sapere di dover (e poter) reinserire il numero. Il Listato 17.16 contiene una piccola variante dell'interfaccia utente grafica del Listato 17.15. Quando l'utente inserisce un numero in un formato non corretto, viene generata ancora una `NumberFormatException`, ma in questo caso l'eccezione è gestita e l'interfaccia utente grafica chiede all'utente di reinserire il numero.

Quando l'utente preme uno dei pulsanti dell'interfaccia utente grafica del Listato 17.16, viene generato un evento di tipo azione. Il metodo `actionPerformed` viene invocato automaticamente con quell'evento come argomento. Il metodo `actionPerformed` chiama il metodo `provaFormatiNumericiCorretti`, passandogli l'evento di tipo azione come argomento. A questo punto, il metodo `provaFormatiNumericiCorretti` elabora l'evento esattamente come faceva il metodo `actionPerformed` della versione precedente del programma. Se l'utente inserisce solo numeri in formato corretto, il comportamento è lo stesso del Listato 17.15. Ma se l'utente inserisce un numero in un formato non corretto, accade qualcosa di diverso.

Per esempio, quando l'utente inserisce `2,35`, con una virgola, invece di `2.35`, il metodo `provaFormatiNumericiCorretti` chiama il metodo `daStringaADouble` con la stringa `"2,35"` come argomento. A questo punto, `daStringaADouble` chiama `Double.parseDouble`, che però non può convertire `"2,35"` in un numero, perché in Java nessuna stringa in formato numerico corretto può contenere una virgola. Quindi il metodo `Double.parseDouble` genera una `NumberFormatException`. Dato che ciò accade in un'invocazione di `daStringaADouble`, quest'ultimo metodo genera a sua volta una `NumberFormatException`. La chiamata a `daStringaADouble` avviene all'interno dell'invocazione di `provaFormatiNumericiCorretti`, che quindi genera la `NumberFormatException` che ha ricevuto da `daStringaADouble`. Tuttavia, la chiamata a `provaFormatiNumericiCorretti` è inclusa in un blocco `try-catch` (evidenziato nel Listato 17.16). L'eccezione è quindi gestita nel blocco `catch`. A quel punto, il testo del campo di testo `campoInputOutput` è impostato al messaggio di errore `Errore: reinserire il numero.`, così che l'interfaccia utente grafica assume l'aspetto riportato alla fine del Listato 17.16.

Si noti che nel caso in cui il metodo `daStringaADouble` generi una `NumberFormatException`, il valore della variabile `somma` non viene modificato. L'esecuzione del metodo `provaFormatiNumericiCorretti` termina immediatamente, quindi l'operazione di addizione, che modificherebbe il valore di `somma`, non viene eseguita. Poiché `somma` non viene modificata, l'utente può reinserire l'ultimo numero e tutto procede come se il numero scorretto non fosse mai stato inserito.

Si noti anche che nessuno dei metodi del Listato 17.16 ha una clausola `throws` della forma

```
throws NumberFormatException
```

Questo accade perché `NumberFormatException` è una discendente della classe `RuntimeException` e Java non richiede che si dichiarino le eccezioni di tipo run-time in una clausola `throws`. Tuttavia, rimane possibile catturare un'eccezione di tipo `NumberFormatException`, come per qualunque altro tipo di eccezione.

MyLab

LISTATO 17.16 Un'interfaccia utente grafica con gestione delle eccezioni.

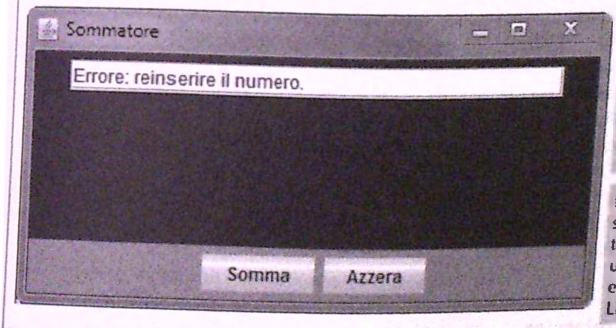
Questa classe è simile alla classe `Sommatore` del Listato 17.15, ma gestisce gli errori nell'inserimento dei dati utilizzando le eccezioni.

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;
import java.awt.Color;
import java.awt.Container;
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * Interfaccia per la somma di una serie di numeri.
 * Se l'utente inserisce un numero in formato non corretto,
 * viene generato un messaggio di errore ed è possibile
 * reinserire il numero.
 */
public class SommatoreMigliore extends JFrame implements ActionListener {
    public static final int LARGHEZZA = 400;
    public static final int ALTEZZA = 200;

    private JTextField campoInputOutput;
    private double somma = 0;

    public SommatoreMigliore() {
        <Il resto della definizione è lo stesso del costruttore del Listato 17.15.>
    }

    public void actionPerformed(ActionEvent e) {
        try {
            provaFormatiNumericiCorretti(e);
        } catch (NumberFormatException e2) {
            campoInputOutput.setText("Errore: reinserire il numero.");
        }
    }

    // Questo metodo può generare una NumberFormatException
    public void provaFormatiNumericiCorretti(ActionEvent e) {
        if (e.getActionCommand().equals("Somma")) {
            somma = somma + daStringaADouble(campoInputOutput.getText());
            campoInputOutput.setText(Double.toString(somma));
```

Una `NumberFormatException` è un'eccezione di tipo run-time, quindi non è necessario dichiararla in una clausola `throws`, ma a parte questo può essere gestita come qualunque altra eccezione.

```
        } else if (e.getActionCommand().equals("Azzera")) {
            somma = 0;
            campoInputOutput.setText("0.0");
        } else
            campoInputOutput.setText("Errore nel codice.");
    }

    // Questo metodo può generare una NumberFormatException
    private static double daStringaADouble(String stringa) {
        return Double.parseDouble(stringa.trim());
    }

    public static void main(String args[]) {
        SommatoreMigliore gui = new SommatoreMigliore();
        gui.setVisible(true);
    }
}
```

Output sullo schermo



Se l'utente inserisce un numero in formato non corretto, per esempio un numero con una virgola al posto del punto decimale, l'interfaccia utente grafica assumerà questo aspetto.

Fintantoché l'utente inserisce solo numeri formattati correttamente, questa interfaccia utente grafica si comporta esattamente come quella del Listato 17.15.

Gestire le `NumberFormatException` nelle interfacce utente grafiche con input numerico

Ogniqualvolta è necessario gestire input numerici in un'interfaccia utente grafica basata su Swing, il codice dovrà convertire una stringa, come `"2000"`, in un numero, in questo caso `2000`. Se l'utente inserisce una stringa che non è in un formato numerico appropriato, verrà generata una `NumberFormatException`. Di conseguenza, è sempre una buona idea gestire tutte le eccezioni di questo tipo, in modo che in caso di errore l'interfaccia utente grafica possa fare qualcosa di sensato, come per esempio mostrare un messaggio di errore, anziché ritrovarsi in un qualche stato non ben definito.

ESEMPIO DI PROGRAMMAZIONE
UNA CALCOLATRICE SEMPLIFICATA

Il Listato 17.17 contiene un programma che gestisce addizioni e sottrazioni. L'utente inserisce un numero in un campo di testo e successivamente preme il pulsante + o il pulsante -. Il numero inserito nel campo di testo è quindi sommato o sottratto di conseguenza dal totale mantenuto in memoria nella variabile `risultato`, il cui nuovo valore viene mostrato nel campo di testo. Se l'utente preme il pulsante `Azzera`, il totale corrente viene posto uguale a zero. Anche all'inizio dell'esecuzione del programma il totale viene posto uguale a zero.

Il programma del Listato 17.17 è simile a quello del Listato 17.16 e presenta in più la possibilità di scegliere tra due operazioni. Nei progetti di programmazione proposti come esercizio alla fine del capitolo, si chiederà di estendere ulteriormente questo programma per gestire anche altre operazioni.

MyLab

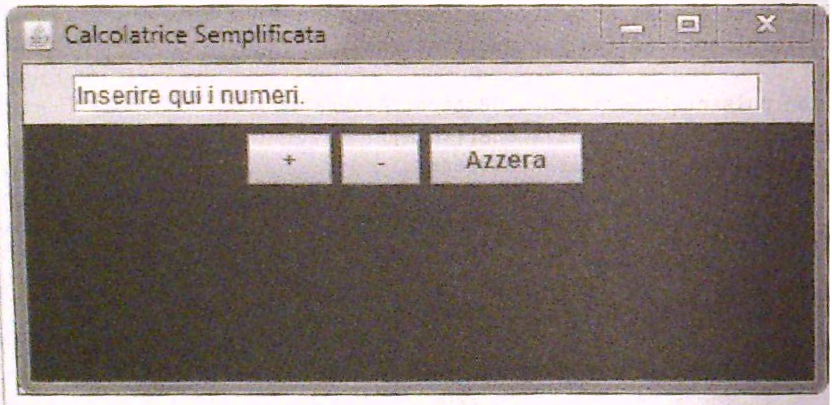
LISTATO 17.17 Una semplice calcolatrice.

```
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.JButton;
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.Color;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

/**
 * Una semplice calcolatrice.
 * Le uniche operazioni possibili sono addizione e sottrazione.
 */
public class Calcolatrice extends JFrame implements ActionListener {
    public static final int LARGHEZZA = 400;
    public static final int ALTEZZA = 200;
    public static final int NUMERO_DI_CIFRE = 30;

    private JTextField campoInputOutput;
    private double risultato = 0.0;

    public Calcolatrice() {
        setSize(LARGHEZZA, ALTEZZA);
        addWindowListener(new DistruttoreFinestra());
        setTitle("Calcolatrice Semplificata");

        setLayout(new BorderLayout());
```

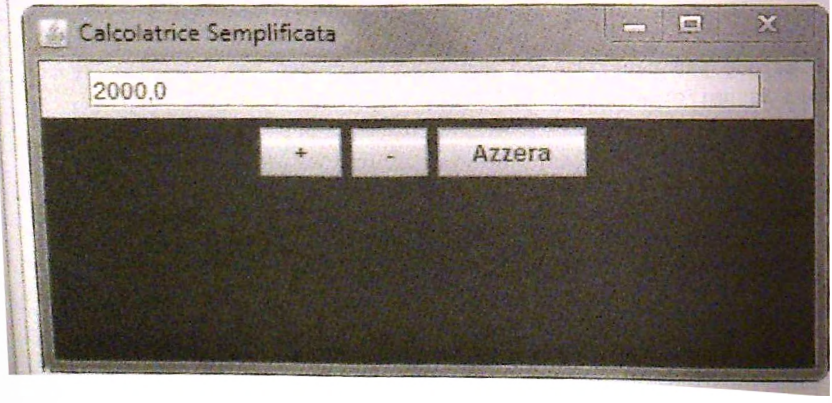
```
        JPanel pannelloTesto = new JPanel();
        pannelloTesto.setLayout(new FlowLayout());

        campoInputOutput = new JTextField("Inserire qui i numeri.",
                                          NUMERO_DI_CIFRE);
        campoInputOutput.setBackground(Color.WHITE);
        pannelloTesto.add(campoInputOutput);

        add(pannelloTesto, BorderLayout.NORTH);

        JPanel pannelloPulsanti = new JPanel();
        pannelloPulsanti.setBackground(Color.BLUE);
        pannelloPulsanti.setLayout(new FlowLayout());

        JButton pulsanteSomma = new JButton("+");
        pulsanteSomma.addActionListener(this);
        pannelloPulsanti.add(pulsanteSomma);

        JButton pulsanteSottrazione = new JButton("-");
        pulsanteSottrazione.addActionListener(this);
        pannelloPulsanti.add(pulsanteSottrazione);

        JButton pulsanteAzzera = new JButton("Azzera");
        pulsanteAzzera.addActionListener(this);
        pannelloPulsanti.add(pulsanteAzzera);

        add(pannelloPulsanti, BorderLayout.CENTER);
    }

    public void actionPerformed(ActionEvent e) {
        try {
            assumiFormatiNumericiCorretti(e);
        } catch (NumberFormatException e2) {
            campoInputOutput.setText("Errore: reinserire il numero.");
        }
    }

    // Questo metodo può generare una NumberFormatException
    public void assumiFormatiNumericiCorretti(ActionEvent e) {
        String comando = e.getActionCommand();

        if (comando.equals("+")) {
            risultato = risultato +
                    daStringaADouble(campoInputOutput.getText());
            campoInputOutput.setText(Double.toString(risultato));
        } else if (comando.equals("-")) {
            risultato = risultato -
                    daStringaADouble(campoInputOutput.getText());
            campoInputOutput.setText(Double.toString(risultato));
```

```
        } else if (comando.equals("Azzera")) {
            risultato = 0;
            campoInputOutput.setText("0.0");
        } else
            campoInputOutput.setText("Errore inatteso.");
    }

    // Questo metodo può generare una NumberFormatException
    private static double daStringaADouble(String stringa) {
        return Double.parseDouble(stringa.trim());
    }

    public static void main(String args[]) {
        Calcolatrice gui = new Calcolatrice();
        gui.setVisible(true);
    }
}
```

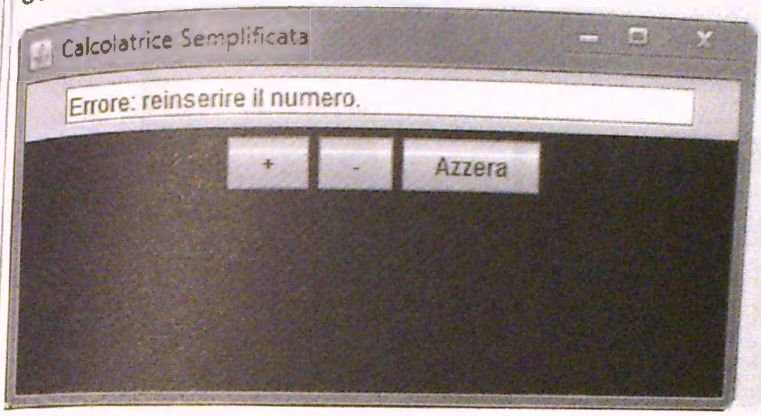
GUI risultante (all'inizio dell'esecuzione)



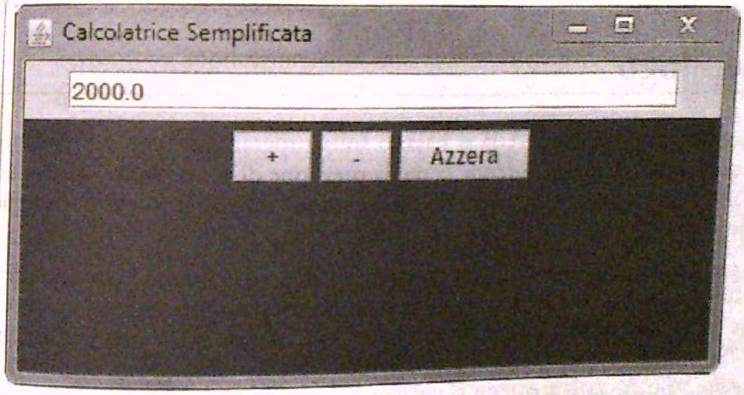
GUI risultante (dopo aver inserito 2000,0)



GUI risultante (dopo aver premuto +)

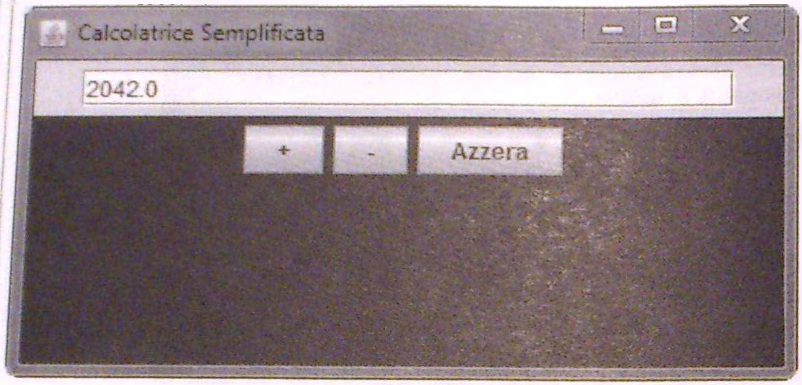


GUI risultante (dopo aver inserito 2000.0 e premuto +)



GUI risultante (dopo aver inserito 42)

GUI risultante (dopo aver premuto +)



17.7 Riepilogo

- Le interfacce utente grafiche (GUI) si basano sulla programmazione a eventi, secondo la quale ogni azione dell'utente, come la pressione di un pulsante, genera un evento. Tale evento è passato automaticamente a un metodo di gestione che esegue le operazioni appropriate.
- Esistono principalmente due modi per costruire un'interfaccia utente grafica utilizzando la libreria Swing. Si può utilizzare l'ereditarietà creando una classe derivata da una classe Swing predefinita o si possono aggiungere componenti a una classe contenitore. Normalmente, si utilizzano contemporaneamente entrambe le tecniche.
- Un'interfaccia a finestre è definita creando una classe derivata dalla classe `JFrame` della libreria Swing.
- Un'etichetta è un oggetto della classe `JLabel`. Le etichette possono essere utilizzate per aggiungere testo a un'interfaccia utente grafica.
- Un pulsante è un oggetto della classe `JButton`. La pressione di un pulsante genera un evento di tipo azione, gestito da un gestore di eventi di tipo azione (*action listener*). Un gestore di eventi di tipo azione è un oggetto di una qualunque classe che implementi l'interfaccia `ActionListener`.
- È possibile definire una classe ascoltatore di finestra implementando l'interfaccia `WindowListener`.
- Per aggiungere elementi a un oggetto di una classe contenitore si utilizza il metodo add. I componenti in un contenitore sono disposti da un oggetto detto gestore di layout.

- Un pannello è un oggetto di tipo contenitore utilizzato per raggruppare elementi in un oggetto più grande. I pannelli sono oggetti della classe `JPanel`.
- I campi di testo (oggetti della classe `JTextField`) e le aree di testo (oggetti della classe `JTextArea`) sono usati per gestire l'input e l'output di testo in un'interfaccia utente grafica basata su Swing.

17.8 Esercizi

1. Scrivere un'applicazione con interfaccia utente grafica che crei due finestre. Nella prima finestra, si inserisca l'etichetta `"Dov'è Giovanni?"` e nella seconda l'etichetta `"Giovanni sta sciando"`. *Suggerimento*: si utilizzi una sola classe per entrambe le finestre, usando un argomento del costruttore per specificare quale stringa debba essere mostrata nell'etichetta.
2. Si scriva un'applicazione con interfaccia utente grafica che crei tre finestre. Ogni finestra dovrà avere un colore diverso e il titolo di ogni finestra dovrà essere in accordo con il colore corrispondente. Utilizzare i colori magenta, arancione e verde.
3. Scrivere un'applicazione con interfaccia utente grafica che crei una finestra con un gestore di layout di tipo `BorderLayout`. Si inseriscano, ognuna nella posizione appropriata, le cinque seguenti etichette: `Posizione Nord`, `Posizione Sud`, `Posizione Ovest`, `Posizione Est` e `Posizione Centrale`.
4. Scrivere un'applicazione con interfaccia utente grafica che crei una finestra con un gestore di layout di tipo `FlowLayout`. Posizionare nella finestra, in ordine numerico crescente, le seguenti sette etichette: `Posizione uno`, `Posizione due`, `Posizione tre`, `Posizione quattro`, `Posizione cinque`, `Posizione sei`, `Posizione sette`.
5. Si ripeta l'esercizio precedente utilizzando un gestore di layout a griglia 2 per 2.
6. Scrivere un'applicazione con interfaccia utente grafica che crei una finestra contenente un pulsante, con la scritta `Cambia`, posizionandolo nella posizione nord. Impostare inizialmente il colore di sfondo della finestra a rosso. Ogni volta che l'utente preme un pulsante, si cambi il colore di sfondo da rosso a bianco, da bianco a blu, o da blu a rosso a seconda dello stato attuale.
7. Si supponga di voler scrivere un'applicazione con interfaccia utente grafica che faccia da cronometro. Per cominciare, si realizzerà un'interfaccia utente grafica nella quale i pulsanti mostreranno soltanto quale pulsante sia stato premuto, senza eseguire altre operazioni.

   Si crei un'applicazione con interfaccia utente grafica con una singola finestra, i tre pulsanti `Inizio`, `Fine` e `Azzeramento` e un'etichetta. Quando viene premuto il pulsante `Inizio`, si cambi il colore del testo dell'etichetta a verde e il suo testo a `Premuto Inizio`. Quando viene premuto il pulsante `Fine`, si cambi il colore dell'etichetta a rosso e il testo a `Premuto Fine`. Quando viene premuto il pulsante `Azzeramento`, si cambi il colore dell'etichette ad arancione e il testo a `Premuto Azzeramento`.

8. Scrivere un'applicazione che modelli la tastiera di un telefono. Si usi un `JPanel` per contenere dodici pulsanti, cioè `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `*`, `0`, `#`, al centro di una disposizione a griglia. Posizionare un'etichetta vuota in posizione sud. Ogni volta che viene premuto un pulsante, aggiungere la cifra o il simbolo corrispondente al testo dell'etichetta.

9. Scrivere un'applicazione che implementi una codifica per sostituzione. Occorrerà un'area di testo, al centro della finestra, che non possa essere modificata. Quest'area di testo mostrerà la codifica, in un formato come `A->C`, `B->Q`, `C->F` e così via. Ciò significa che `A` verrebbe sostituita da `C`, `B` da `Q`, `C` da `F` e così via. Il codice sarà generato una lettera alla volta, dalla `A` alla `Z`. Dovranno essere presenti 26 pulsanti, ognuno contrassegnato da una lettera dell'alfabeto inglese. Il primo pulsante premuto specificherà la lettera da sostituire alla `A`, il secondo pulsante premuto quella da sostituire alla `B`, e così via. Per esempio, per costruire la codifica riportata sopra, occorrerebbe premere i pulsanti in una sequenza che inizi con `C`, `Q`, `F`. Dopo la pressione di ogni pulsante, si aggiunga la sostituzione corrispondente all'area di testo e si nasconda il pulsante premuto.

10. Scrivere un'applicazione che crei una lista di nomi. Occorrerà un'area di testo, al centro della finestra, in grado di contenere dieci righe e che non possa essere modificata. Posizionare inoltre un campo di testo e un pulsante `Accetta` nella posizione sud. Quando l'utente inserisce un nome nel campo di testo e preme il pulsante `Accetta`, si prenda il nome dal campo di testo, lo si aggiunga all'area di testo e si cancelli il contenuto del campo di testo.

11. Scrivere un'applicazione con interfaccia utente grafica che possa costituire la base per un'applicazione più complessa. L'applicazione dovrà leggere numeri di carte di credito inseriti in un campo di testo. Quando l'utente preme un pulsante `Accetta`, l'applicazione dovrà controllare che il numero inserito sia composto esattamente da 16 cifre. In tal caso, dovrà mostrare in un'etichetta il messaggio `Numero accettato:` seguito dal numero inserito e successivamente cancellare il contenuto del campo di testo. Se il numero inserito non è composto da 16 cifre, dovrà essere mostrato nell'etichetta il messaggio `Numero rifiutato`.

12. Scrivere un'applicazione con interfaccia utente grafica che possa costituire la base per un'applicazione più complessa. L'applicazione dovrà consentire all'utente di inserire un nome utente e una password in campi di testo separati. Quando l'utente preme il pulsante `Accedi`, l'applicazione dovrà verificare che la stringa nel campo per il nome utente sia `"Giovanni"` e che quella nel campo per la password sia `"miaPassword"`. Se entrambe le stringhe sono corrette, l'applicazione mostrerà in un'etichetta il messaggio `Accesso effettuato correttamente`, altrimenti mostrerà il messaggio `Password non valida`.

13. Scrivere un'applicazione con interfaccia utente grafica che possa costituire la base per un'applicazione più complessa. L'applicazione dovrà consentire all'utente di modificare una password. L'utente dovrà inserire la password in un campo di testo ripetendola in un secondo campo di testo e poi premere il pulsante `Cambia`. L'applicazione dovrà verificare se le due stringhe coincidono. Se è così, mostrerà in un'etichetta il messaggio `Password modificata` e nasconderà il pulsante. Altrimenti, mostrerà nell'etichetta il messaggio `Errore: le password sono diverse`.

14. Scrivere un'applicazione con interfaccia utente grafica che possa costituire la base per un'applicazione più complessa. L'applicazione rivolgerà all'utente tre domande. Dovranno essere presenti tre etichette per le domande e tre campi di testo per le risposte. Quando l'utente preme il pulsante `Accetta`, l'applicazione dovrà verificare che ogni campo di testo contenga una stringa non vuota. Se è così, dovrà mostrare, in un'etichetta, il messaggio `Le risposte sono state registrate`. Altrimenti, dovrà cambiare in rosso il colore delle etichette corrispondenti ai campi di testo vuoti e mostrare il messaggio `È necessario rispondere a tutte le domande`.

15. Scrivere un'applicazione con interfaccia utente grafica che possa costituire la base per un'applicazione più complessa. L'applicazione leggerà una stringa che indichi la taglia di un capo di vestiario. Le taglie valide sono `S`, `M`, `L`, `XL` e `XXL`. Quando l'utente inserisce una taglia in un campo di testo e preme il pulsante `Accetta`, occorrerà controllare che essa sia tra quelle valide. Se la taglia è valida, mostrare in un'etichetta il messaggio `Taglia accettata:` seguito dalla taglia specificata, altrimenti mostrare il messaggio `Taglia non valida`.

16. Scrivere un'applicazione con interfaccia utente grafica che possa costituire la base per un'applicazione più complessa. L'applicazione chiederà all'utente se ha più di 16 anni. Si mostri in un'etichetta il testo `Hai più di 16 anni?`. Se l'utente preme il pulsante `Sì`, mostrare in un'altra etichetta il messaggio `L'utente ha più di 16 anni`, se preme il pulsante `No` mostrare il messaggio `L'utente ha meno di 16 anni`. In entrambi i casi, nascondere poi i due pulsanti.

17.9 Progetti

1. Riscrivere il programma del Listato 17.11 in modo che quando il pannello principale diventa rosso quello con i pulsanti diventi rosa. Analogamente, quando il pannello principale diventa verde, quello con i pulsanti dovrà diventare blu. Aggiungere anche al pannello principale un'etichetta con la scritta `Guarda questo pannello!` e al pannello dei pulsanti un pulsante con la scritta `Cambia`. Quando viene premuto questo pulsante, i colori dovranno cambiare (da rosa e rosso a blu e verde, rispettivamente, o viceversa). Il pulsante `Cambia` non dovrà avere alcun effetto sulla configurazione iniziale, nella quale il pannello principale è blu e quello dei pulsanti è grigio.

2. Riscrivere il programma del Listato 17.13 con i seguenti cambiamenti:
   - il nome della classe è `SalvaAppunti2`;
   - al posto dei cinque pulsanti in basso ce ne sono sei, disposti in questo modo:

`Salva appunto 1`	`Salva appunto 2`	`Cancella`
`Carica appunto 1`	`Carica appunto 2`	`Esci`

Suggerimento: utilizzare un gestore di layout a griglia per il pannello dei pulsanti.

quando l'utente salva il primo appunto, il testo dell'area di testo dovrà diventare `Appunto 1 salvato` e quando salva il secondo appunto il testo dovrà diventare `Appunto 2 salvato`;

- quando viene premuto il pulsante `Esci`, il programma termina e la finestra scompare. Lo stesso accadrà con il pulsante di chiusura della finestra. Quindi, il pulsante `Esci` e quello di chiusura della finestra eseguiranno la stessa operazione;
- aggiungere un costruttore che realizzi la stessa interfaccia utente grafica a eccezione del fatto che l'area di testo potrà ospitare dei numeri specificati di righe e di caratteri per riga. Il costruttore avrà la forma

```
public SalvaAppunti2(int righe, int caratteriPerRiga)
```

- l'area di testo va a capo automaticamente, in modo che se l'utente inserisce più caratteri di quanti siano visualizzabili su una singola riga, i caratteri in più vengano spostati automaticamente nella riga successiva;
- il metodo `main` costruisce due finestre, una utilizzando il costruttore di default e una utilizzando il nuovo costruttore con parametri, rispettivamente, `5` e `60`.

3. (È opportuno svolgere il progetto precedente prima di questo.) Scrivere un'interfaccia utente grafica che si comporti come segue. Quando il programma viene eseguito, compare una finestra che chiede all'utente il numero di righe e di caratteri per riga desiderati per il salvataggio degli appunti. Se l'utente preme il pulsante di chiusura di questa finestra, il programma termina. Se ciò non accade, l'utente inserirà due numeri in due campi di testo. È disponibile un pulsante `Continua` che, se premuto, chiude la prima finestra e ne fa comparire una seconda. Questa seconda finestra è come quella del Progetto 2, con il numero di righe e di caratteri per riga dell'area di testo uguali a quelli specificati nella prima finestra.
4. (La parte di questo progetto sulle interfacce è abbastanza semplice, ma occorre saper convertire numeri da una base a un'altra.) Scrivere un programma che converta numeri dalla base 10 alla base 2. Il programma utilizzerà componenti Swing per realizzare l'input e l'output in un'interfaccia a finestre. L'utente inserirà il numero in base 10 in un campo di testo e premerà il pulsante `Converti`. Il valore equivalente espresso in base 2 dovrà apparire in un secondo campo di testo. Ci si assicuri che i due campi di testo siano descritti da etichette. Si includa un pulsante `Cancella` che cancelli il contenuto dei campi di testo. Ci si assicuri anche che il pulsante di chiusura della finestra funzioni correttamente.
5. (Probabilmente sarebbe utile svolgere il Progetto 4 prima di questo.) Scrivere un programma che converta numeri dalla base 2 alla base 10. Il programma utilizzerà componenti Swing per realizzare l'input e l'output in un'interfaccia a finestre. L'utente inserirà il numero in base 2 in un campo di testo e premerà il pulsante `Converti`. Il valore equivalente espresso in base 10 dovrà apparire in un secondo campo di testo. Ci si assicuri che i due campi di testo siano descritti da etichette. Si includa un pulsante `Cancella` che cancelli il contenuto dei campi di testo. *Suggerimento*: includere un metodo privato che converta una stringa con un numero in base 2 nel valore `int` equivalente.

6. (Probabilmente sarebbe utile svolgere i Progetti 4 e 5 prima di questo.) Scrivere un programma che converta numeri dalla base 2 alla base 10 e viceversa. Il programma utilizzerà componenti Swing per realizzare l'input e l'output in un'interfaccia a finestre. Sono presenti due campi di testo, uno per i numeri in base 2 e uno per quelli in base 10, e tre pulsanti `In Base 10`, `In Base 2` e `Cancella`. Se l'utente inserisce un numero in base 2 nel campo di testo per i numeri in base 2 e preme il pulsante `In Base 10`, il corrispondente numero in base 10 appare nel campo di testo per i numeri in base 10. Analogamente, se l'utente inserisce un numero in base 10 nel campo di testo per i numeri in base 10 e preme il pulsante `In Base 2`, il numero equivalente in base 2 compare nel campo di testo corrispondente. Ci si assicuri che entrambi i campi di testo siano corredati da etichette che li descrivano. Il pulsante `Cancella` deve cancellare il contenuto di entrambi i campi di testo. Ci si assicuri anche di programmare correttamente il pulsante di chiusura della finestra.
7. Scrivere un programma che produca un'interfaccia utente grafica con le funzionalità e l'aspetto di una calcolatrice tascabile. La calcolatrice dovrà consentire di effettuare le operazioni di addizione, sottrazione, moltiplicazione e divisione. Dovrebbe anche permettere di salvare e recuperare in seguito due valori diversi. Si utilizzino i programmi dei Listati 17.16 e 17.17 come modelli.
8. Nell'Esercizio 8 di questo capitolo si è chiesto di realizzare un'interfaccia utente grafica che modellasse la tastiera di un telefono. Ora si vuole migliorare il comportamento di quell'applicazione. Ecco la lista dei miglioramenti da applicare:
   - la prima cifra di un numero di telefono non può essere 9. Se l'utente prova a inserire 9 come prima cifra, lo si ignori;
   - il numero deve essere formattato in questo modo:
     - `00-000000` se sono state inserite otto cifre;
     - `(000) 0000000` se sono state inserite dieci cifre;
     - `000000` se sono state inserite sei cifre (o meno).
   - non accettare un numero maggiore di cifre.
9. Scrivere un'applicazione chiamata `Rimescola` con un'interfaccia utente grafica che consenta di giocare con gli anagrammi. Si creino due array di stringhe. Il primo conterrà delle parole, mentre il secondo conterrà delle versioni rimescolate di queste parole. Il codice Java del programma può inizializzare direttamente questi array con delle parole. Si mostri in un'etichetta la versione anagrammata di una parola. L'utente proverà a indovinare la parola originale inserendola in un campo di testo e premendo il pulsante `Verifica`. Il programma verificherà se la parola è corretta. Se non lo è, dovrà cambiare il contenuto del campo di testo in `Spiacente, hai sbagliato. Prova ancora`. Se invece l'utente ha indovinato la parola corretta, il contenuto del campo di testo dovrà diventare la stringa `Esatto. Ecco un'altra parola da indovinare` e mostrare un nuovo anagramma. Fornire anche un pulsante `Mi arrendo` che, se premuto, mostra la parola originale e un nuovo anagramma da risolvere. Ecco alcune estensioni che possono migliorare questa applicazione:
   - leggere le parole da un file;

- non utilizzare un array di anagrammi predefiniti, ma sfruttare la generazione di numeri casuali in Java per rimescolare le lettere della parola originale appena prima di mostrare l'anagramma;
- selezionare in modo casuale la parola da mostrare;
- tenere traccia del punteggio. Assegnare 5 punti se l'utente indovina la parola al primo tentativo, 3 se la indovina al secondo e 1 se la indovina al terzo. Dividere il punteggio totale ottenuto per il numero di parole presentate.

10. Scrivere un'applicazione con un'interfaccia utente grafica che converta numeri dal formato binario a quello ottale. I numeri binari sono composti solo dalle cifre `0` e `1`. Quelli ottali usano le cifre `0`, `1`, `2`, `3`, `4`, `5`, `6` e `7`. Si noti che ogni cifra ottale corrisponde a un numero binario a tre bit, nel modo seguente:

Ottale	0	1	2	3	4	5	6	7
Binario	000	001	010	011	100	101	110	111

Per convertire un numero da binario a ottale, si raggruppino per prima cosa le sue cifre a tre a tre e si converta ogni gruppo nella cifra ottale corrispondente. Per esempio, i bit del numero binario `001000101110` sarebbero raggruppati come `001 000 101 110`, che corrispondono, rispettivamente, alle cifre ottali `1`, `0`, `5` e `6`. Quindi, il numero ottale corrispondente a quello binario specificato sarebbe `1056`. Se il numero di bit nel numero binario originale non è multiplo di 3, si aggiungano degli zeri all'inizio finché non lo diventa. Per esempio, poiché il numero binario `1011011100100` ha 13 bit, si dovrebbero aggiungere due zeri in modo da ottenere il numero `001011011100100`. Questi bit sarebbero quindi raggruppati come `001 011 011 100 100`, ottenendo `13344` come numero ottale equivalente.

Per convertire un numero ottale in formato binario, si utilizza la tabella delle corrispondenze tra cifre ottali e terne di bit in senso opposto. Per esempio, il numero ottale `716` corrisponde a `111 001 110`, cioè `111001110`, in formato binario. L'applicazione può omettere gli spazi utilizzati qui per separare le terne di bit.

L'applicazione dovrà avere un campo di testo per consentire all'utente di inserire un numero e un'etichetta per mostrare i risultati. Dovranno inoltre essere previsti tre pulsanti: `In Ottale`, `In Binario` e `Cancella`. Se l'utente preme il pulsante `Cancella`, l'applicazione dovrà cancellare il contenuto del campo di testo e il testo dell'etichetta. Se l'utente preme uno dei due pulsanti di conversione, l'applicazione deve verificare che il numero inserito nel campo di testo sia nel formato corretto. Se non lo è, si mostri il messaggio `Formato non corretto` nell'etichetta del risultato. Se invece il formato è corretto, si calcoli il valore convertito e lo si mostri nell'etichetta del risultato.

Appendice 1

Come ottenere una copia di Java

Numerosi compilatori Java e ambienti di sviluppo integrato (IDE – *Integrated Development Environment*) sono disponibili gratuitamente per i sistemi operativi più diffusi.

Il seguente elenco contiene i link di una selezione di compilatori e ambienti di sviluppo liberamente utilizzabili.

Java SE Development Kit (JDK)
JDK è sviluppato da Oracle ed è disponibile per i sistemi operativi Linux, Solaris e Windows. JDK include il Java Runtime Environment (JRE) e gli strumenti a riga di comando utilizzabili per scrivere applicazioni. È possibile scaricare la JDK al seguente indirizzo:

```
http://www.oracle.com/technetwork/java/javase/downloads/index.html
```

BlueJ
BlueJ è un ambiente di sviluppo di origine accademica. È stato specificatamente creato per studenti che si avvicinano a Java per la prima volta ed è disponibile per MacOS, Windows e altri sistemi operativi. È possibile scaricarlo al seguente indirizzo:

```
www.bluej.org/download/download.html
```

Java deve essere già stato installato sul computer perché BlueJ funzioni correttamente.

Eclipse
Eclipse è un IDE open source per Linux, MacOS e Windows. Può essere scaricato al seguente indirizzo:

```
www.eclipse.org/downloads/
```

Java deve essere già stato installato sul computer perché Eclipse funzioni correttamente.

NetBeans
NetBeans è un IDE open source per Linux, MacOS, Solaris e Windows. Può essere scaricato al seguente indirizzo:

```
www.netbeans.info/downloads/index.php
```

Java deve essere già stato installato sul computer perché NetBeans funzioni correttamente.

Per istruzioni dettagliate sulle modalità di installazione di questi strumenti si rimanda agli indirizzi indicati.

Appendice 2

Javadoc

La JDK viene distribuita assieme a `javadoc`: un programma in grado di generare automaticamente documenti HTML che descrivono le classi Java che fanno parte di un programma. La documentazione prodotta per ogni classe specifica le informazioni utili per poterla utilizzare e omette i dettagli implementativi, come il corpo dei metodi (pubblici e privati), le firme dei metodi privati e le variabili private.

`javadoc` è normalmente applicato a interi package, ma può essere utilizzato anche per singole classi. I package sono discussi nel Capitolo 9. Per poter visualizzare la documentazione prodotta da `javadoc` è necessario utilizzare un browser HTML (Web browser). Non è comunque necessario conoscere il linguaggio HTML per poter usare `javadoc`.

In questa appendice si vedrà innanzitutto come aggiungere commenti alle classi Java, in modo da ottenere il massimo beneficio da `javadoc`. Poi si parlerà dell'uso del programma `javadoc`.

Aggiungere i commenti per `javadoc`

Per generare documenti utili per `javadoc`, i commenti delle classi devono essere scritti in un modo particolare. `javadoc` estrae automaticamente i commenti, l'intestazione delle classi, l'intestazione di tutti i metodi pubblici e tutte le dichiarazioni di variabili pubbliche. Ignora invece il corpo dei metodi e tutti i membri privati.

Affinché i commenti siano estratti da `javadoc`, essi devono:

- precedere immediatamente la definizione di una classe pubblica, la definizione di un metodo pubblico e, in generale, un qualsiasi elemento pubblico;
- iniziare con `/**` e finire con `*/`.

Nel resto di questa appendice si chiameranno questi commenti, **commenti javadoc**. Si noti che i commenti che iniziano con `//` o che precedono i membri privati non saranno considerati da `javadoc`.

Nei commenti `javadoc` è possibile utilizzare dei **tag** che descrivono parametri, valori restituiti ed elementi analoghi. Di seguito viene mostrata la sintassi per alcuni dei tag:

`@param` *nome_parametro descrizione_parametro*
`@return` *descrizione_valore_restituito*
`@throws` *tipo_eccezione spiegazione*

Ogni tag deve trovarsi all'inizio di una nuova riga e deve iniziare con il simbolo `@`. Questi tag vanno utilizzati nell'ordine sopra riportato. I tag non necessari possono essere omessi e possono apparire più volte se necessario.

Per esempio, il metodo seguente ha un commento `javadoc` valido:

```
/**
Calcola il costo totale per un insieme di prodotti identici.
@param quantita numero di elementi acquistati
@param prezzo costo di un elemento
@return costo totale di tutti gli elementi al prezzo indicato
*/
public static double calcolaCosto(int quantita, double prezzo) {
   return quantita * prezzo;
}
```

La documentazione generata da `javadoc` per questo commento sarà:

calcolaCosto
public static double **calcolaCosto**(int quantita, double prezzo)
Calcola il costo totale per un insieme di prodotti identici.
Parameters:
quantita - numero di elementi acquistati
prezzo - costo di un elemento
Returns:
costo totale di tutti gli elementi al prezzo indicato

È possibile inserire del codice HTML nei commenti `javadoc` per produrre una documentazione dalla struttura più complessa. Non è comunque necessario utilizzare il codice HTML nei commenti e in generale non è opportuno farlo, perché il codice HTML renderebbe i commenti meno comprensibili quando vengono letti direttamente dal codice sorgente.

Per risparmiare spazio, i commenti `javadoc` impiegati in questo testo non includono né tag, né codice HTML.

Esecuzione di `javadoc`

È possibile eseguire `javadoc` su tutte le classi che fanno parte di un package e anche su classi che non appartengono ad alcun package.

Per eseguire `javadoc` su un package, la cartella corrente deve coincidere con quella che contiene la cartella corrispondente al package che `javadoc` deve elaborare. Il comando da eseguire per produrre la `javadoc` è il seguente:

```
javadoc –d CartellaDocumenti NomeDelPackage
```

CartellaDocumenti è il nome della cartella che `javadoc` utilizzerà per salvare i documenti HTML prodotti. È possibile chiedere a `javadoc` di produrre documenti HTML contenenti **link**. I link sono utili per esplorare rapidamente la documentazione. Il comando per far inserire i link a `javadoc` è il seguente:

```
javadoc –link Link –d CartellaDocumenti NomeDelPackage
```

Link può essere il percorso della versione locale della documentazione di Java oppure l'URL della documentazione standard di Java sul sito Oracle.

Per eseguire `javadoc` su un solo file, la cartella corrente deve coincidere con quella che contiene il file che `javadoc` deve elaborare. Il comando da eseguire per produrre la `javadoc` è il seguente:

```
javadoc –d CartellaDocumenti NomeClasse.java
```

CartellaDocumenti è il nome della cartella che `javadoc` utilizzerà per salvare i documenti HTML prodotti.

Infine, se si volesse eseguire `javadoc` su un insieme di file, la cartella corrente deve coincidere con quella che contiene i file che `javadoc` deve elaborare. Il comando da eseguire per produrre la `javadoc` è il seguente:

```
javadoc –d CartellaDocumenti *.java
```

Appendice 4

Keyword (parole chiave)

abstract	final	public
assert*	finally	return
boolean	float	short
break	for	static
byte	goto	strictfp
case	if	super
catch	implements	switch
char	import	synchronized
class	instanceof	this
const	int	throw
continue	interface	throws
default	long	transient
do	native	true*
double	new	try
else	null*	void
enum	package	volatile
extends	private	while
false*	protected	

* Queste non sono riportate come parole chiave nella documentazione Oracle. Qualcuno, però, le considera tali. Per maggior sicurezza, in questo testo sono trattate come parole chiave.

Indice analitico

B

C

D

E

F

G

H

I

J

L

M

R

S

T

U