

## Dal catalogo Apogeo Education

### Informatica

- Bolchini, Brandolesi, Salice, Sciuto, *Reti logiche*, seconda edizione  
Bruni, Corradini, Gervasi, *Programmazione in Java*, seconda edizione  
Cabodi, Camurati, Pasini, Patti, Vendraminetto, *Dal problema al programma. Introduzione al problem solving in linguaggio C*, seconda edizione  
Cabodi, Camurati, Pasini, Patti, Vendraminetto, *Ricorsione e problem-solving. Strategie algoritmiche in linguaggio C*  
Chianese, Moscato, Picariello, Sansone, *Sistemi di basi di dati e applicazioni*  
Collins, *Algoritmi e strutture dati in Java*  
Coppola, Mizzaro, *Laboratorio di programmazione in Java*  
Deitel, *C++ Fondamenti di programmazione*, seconda edizione  
Deitel, *C++ Tecniche avanzate di programmazione*, seconda edizione  
Della Mea, Di Gaspero, Scagnetto, *Programmazione web lato server*, seconda edizione aggiornata  
Di Noia, De Virgilio, Di Sciascio, Donini, *Semantic Web. Tra ontologie e Open Data*  
Facchinetti, Larizza, Rubini, *Programmare in C. Concetti di base e tecniche avanzate*  
Goodrich, Tamassia, Goldwasser, *Algoritmi e strutture dati in Java*  
Hanly, Koffman, *Problem solving e programmazione in C*  
Hennessy, Patterson, *Architettura degli elaboratori*  
Horstmann, Necaise, *Concetti di informatica e fondamenti di Python*  
King, *Programmazione in C*  
Laganà, Righi, Romani, *Informatica. Concetti e sperimentazioni*, seconda edizione  
Lambert, *Programmazione in Python*  
Lombardo, Valle, *Audio e multimedia*, quarta edizione  
Malik, *Programmazione in C++*  
Peterson, Davie, *Reti di calcolatori*, terza edizione  
Schneider, Gersting, *Informatica*  
Sipser, *Introduzione alla teoria della computazione*  
Sklar, *Principi di Web Design*  
Tarabella, *Musica informatica. Filosofia, storia e tecnologia della Computer Music*

# Concetti di informatica e fondamenti di Java

sesta edizione  
per Java 7 e Java 8

Cay Horstmann

Edizione italiana a cura  
di Marcello Dalpasso



**Concetti di informatica e fondamenti di Java**  
**Sesta edizione**

Titolo originale:  
**Java Concepts. Early Objects, 8<sup>th</sup> edition**

Autore:  
**Cay Horstmann**

Copyright © 2015 John Wiley & Sons, Inc. All rights reserved.  
Authorized translation from the English language edition published by John Wiley & Sons, Inc.

Traduzione e revisione: Marcello Dalpasso  
Impaginazione elettronica: Grafica editoriale – Vimercate

© Copyright 2016 by Maggioli S.p.A.  
Maggioli Editore è un marchio di Maggioli S.p.A.  
Azienda con sistema qualità certificato ISO 9001: 2008

47822 Santarcangelo di Romagna (RN) • Via del Carpino, 8  
Tel. 0541/628111 • Fax 0541/622595

[www.maggiolieditore.it](http://www.maggiolieditore.it)  
e-mail: [clienti.editore@maggioli.it](mailto:clienti.editore@maggioli.it)

Diritti di traduzione, di memorizzazione elettronica, di riproduzione  
e di adattamento, totale o parziale con qualsiasi mezzo sono riservati  
per tutti i Paesi.

Finito di stampare nel mese di luglio 2016  
nello stabilimento Maggioli S.p.a.  
Santarcangelo di Romagna (RN)

# Sommario

Presentazione della edizione italiana .....	xv
Prefazione .....	xvii
<b>Capitolo 1 – Introduzione .....</b>	<b>1</b>
Obiettivi del capitolo .....	1
1.1 Calcolatori e programmi .....	2
1.2 L'anatomia di un calcolatore .....	3
1.3 Il linguaggio di programmazione Java .....	6
1.4 L'ambiente di programmazione Java.....	8
1.5 Analisi di un semplice programma.....	12
1.6 Errori .....	16
1.7 Problem Solving: progettazione di algoritmi .....	18
1.7.1 Il concetto di algoritmo.....	18
1.7.2 Un algoritmo che risolve un problema finanziario .....	19
1.7.3 Pseudocodice .....	20
1.7.4 Dagli algoritmi ai programmi.....	21
Riepilogo degli obiettivi di apprendimento .....	27
Elementi di libreria presentati nel capitolo .....	28
Esercizi di riepilogo e approfondimento .....	28
Esercizi di programmazione .....	30
<b>Capitolo 2 – Utilizzare oggetti.....</b>	<b>33</b>
Obiettivi del capitolo .....	33
2.1 Oggetti e classi.....	34
2.1.1 Utilizzare oggetti .....	34

2.1.2	Classi.....	35
2.2	Variabili .....	36
2.2.1	Dichiarazioni di variabili.....	36
2.2.2	Tipi.....	37
2.2.3	Nomi .....	38
2.2.4	Commenti.....	39
2.2.5	Assegnazioni.....	40
2.3	Invokeare metodi .....	44
2.3.1	L'interfaccia pubblica di una classe .....	44
2.3.2	Parametri dei metodi .....	45
2.3.3	Valori restituiti.....	45
2.3.4	Dichiarazioni di metodi .....	47
2.4	Costruire oggetti .....	48
2.5	Metodi d'accesso e metodi modificatori .....	51
2.6	La documentazione API .....	53
2.6.1	Consultare la documentazione API .....	53
2.6.2	Pacchetti .....	55
2.7	Realizzare un programma di collaudo .....	56
2.8	Riferimenti a oggetti .....	63
2.9	Applicazioni grafiche .....	67
2.9.1	Finestre con cornice .....	67
2.9.2	Disegnare in un componente .....	69
2.9.3	Visualizzare un componente in un frame .....	72
2.10	Ellissi, segmenti, testo e colore .....	73
2.10.1	Ellissi e cerchi.....	73
2.10.2	Segmenti.....	74
2.10.3	Scrivere testo .....	74
2.10.4	Colori.....	75
	Riepilogo degli obiettivi di apprendimento .....	78
	Elementi di libreria presentati in questo capitolo .....	79
	Esercizi di riepilogo e approfondimento .....	79
	Esercizi di programmazione .....	81
	<b>Capitolo 3 – Realizzare classi.....</b>	<b>85</b>
	Obiettivi del capitolo .....	85
3.1	Variabili di esemplare e encapsulamento .....	86
3.1.1	Variabili di esemplare .....	86
3.1.2	I metodi della classe Counter.....	88
3.1.3	Encapsulamento .....	88
3.2	Progettare l'interfaccia pubblica di una classe .....	90
3.2.1	Definire i metodi.....	90
3.2.2	Definire i costruttori .....	92
3.2.3	Usare l'interfaccia pubblica .....	94
3.2.4	Commentare l'interfaccia pubblica.....	94
3.3	Realizzare la classe .....	98
3.3.1	Definire le variabili di esemplare .....	98
3.3.2	Definire i costruttori .....	98

3.3.3 Definire i metodi.....	100
3.4 Collaudo di unità.....	110
3.5 Problem Solving: tenere traccia dell'esecuzione .....	113
3.6 Variabili locali .....	116
3.7 Il riferimento <code>this</code> .....	120
3.8 Classi per figure complesse .....	123
Riepilogo degli obiettivi di apprendimento .....	131
Esercizi di riepilogo e approfondimento .....	132
Esercizi di programmazione .....	135
<b>Capitolo 4 – Tipi di dati fondamentali.....</b>	<b>141</b>
Obiettivi del capitolo .....	141
4.1 Numeri .....	142
4.1.1 Tipi di numeri.....	142
4.1.2 Costanti .....	144
4.2 Aritmetica.....	150
4.2.1 Operatori aritmetici .....	150
4.2.2 Incremento e decremento.....	151
4.2.3 Divisione intera e resto .....	151
4.2.4 Potenze e radici .....	152
4.2.5 Conversione e arrotondamento.....	153
4.3 Dati in ingresso e in uscita .....	159
4.3.1 Acquisire dati .....	159
4.3.2 Controllare il formato di visualizzazione .....	160
4.4 Problem Solving: prima si risolve a mano .....	171
4.5 Stringhe.....	176
4.5.1 Il tipo di dato <code>String</code> .....	176
4.5.2 Concatenazione.....	176
4.5.3 Acquisire stringhe in ingresso .....	177
4.5.4 Sequenze di escape .....	177
4.5.5 Stringhe e caratteri .....	178
4.5.6 Sottostringhe .....	179
Riepilogo degli obiettivi di apprendimento .....	184
Elementi di libreria presentati in questo capitolo .....	184
Esercizi di riepilogo e approfondimento .....	185
Esercizi di programmazione .....	189
<b>Capitolo 5 – Decisioni .....</b>	<b>195</b>
Obiettivi del capitolo .....	195
5.1 L'enunciato <code>if</code> .....	196
5.2 Confrontare valori .....	203
5.2.1 Operatori relazionali.....	203
5.2.2 Confrontare numeri in virgola mobile .....	204
5.2.3 Confrontare stringhe .....	204
5.2.4 Confrontare oggetti .....	205
5.2.5 Confrontare con <code>null</code> .....	207
5.3 Alternative multiple .....	214

5.4	Diramazioni annidate .....	219
5.5	Problem Solving: diagrammi di flusso .....	228
5.6	Problem Solving: preparare casi di prova .....	231
5.7	Variabili booleane e operatori.....	235
5.8	Applicazione: validità dei dati in ingresso .....	241
	Riepilogo degli obiettivi di apprendimento .....	245
	Elementi di libreria presentati in questo capitolo .....	246
	Esercizi di riepilogo e approfondimento .....	246
	Esercizi di programmazione .....	251
<b>Capitolo 6 – Iterazioni .....</b>		<b>257</b>
	Obiettivi del capitolo .....	257
6.1	Il ciclo while.....	258
6.2	Problem Solving: esecuzione manuale .....	266
6.3	Il ciclo for .....	271
6.4	Il ciclo do .....	280
6.5	Applicazione: elaborazione di valori sentinella .....	282
6.6	Problem Solving: storyboard.....	289
6.7	Algoritmi di uso frequente che utilizzano cicli.....	292
6.7.1	Calcolo di somma e valor medio.....	292
6.7.2	Conteggio di valori che soddisfano una condizione .....	293
6.7.3	Identificazione della prima corrispondenza .....	294
6.7.4	Richiesta ripetuta fino al raggiungimento di un obiettivo .....	294
6.7.5	Valore massimo e minimo .....	295
6.7.6	Confronto di valori adiacenti .....	295
6.8	Cicli annidati .....	304
6.9	Applicazione: numeri casuali e simulazioni .....	310
6.9.1	Generare numeri casuali .....	310
6.9.2	Il metodo Monte Carlo .....	312
6.10	Usare un debugger .....	314
	Riepilogo degli obiettivi di apprendimento .....	326
	Elementi di libreria presentati in questo capitolo .....	327
	Esercizi di riepilogo e approfondimento .....	327
	Esercizi di programmazione .....	333
<b>Capitolo 7 – Array e vettori .....</b>		<b>337</b>
	Obiettivi del capitolo .....	337
7.1	Array .....	338
7.1.1	Dichiarazione e utilizzo di array.....	338
7.1.2	Riferimenti ad array .....	342
7.1.3	Array e metodi .....	342
7.1.4	Array riempiti solo in parte .....	343
7.2	Il ciclo for esteso .....	349
7.3	Algoritmi fondamentali per l'elaborazione di array .....	351
7.3.1	Riempimento .....	351
7.3.2	Somma e valore medio .....	351
7.3.3	Valore massimo e minimo .....	351

7.3.4	Elementi con separatori .....	352
7.3.5	Ricerca lineare .....	352
7.3.6	Eliminazione di un elemento .....	353
7.3.7	Inserimento di un elemento .....	354
7.3.8	Scambio di elementi .....	355
7.3.9	Copiatura di array .....	356
7.3.10	Acquisizione di valori .....	358
7.4	Problem Solving: adattamento di algoritmi .....	362
7.5	Problem Solving: progettare algoritmi facendo esperimenti .....	372
7.6	Array bidimensionali .....	376
7.6.1	Dichiarazione di array bidimensionali .....	376
7.6.2	Accesso agli elementi .....	377
7.6.3	Individuazione degli elementi adiacenti .....	378
7.6.4	Accedere a righe e colonne .....	379
7.7	Vettori .....	386
7.7.1	Dichiarazione e utilizzo di vettori .....	387
7.7.2	Il ciclo <code>for</code> esteso usato con vettori .....	389
7.7.3	Copiatura di un vettore .....	390
7.7.4	Classi involucro (wrapper) e auto-boxing .....	390
7.7.5	Algoritmi per array usati con vettori .....	392
7.7.6	Acquisizione di valori in un vettore .....	393
7.7.7	Eliminazione di specifici valori .....	393
7.7.8	Array o vettore? .....	394
7.8	Collaudo regressivo .....	397
	Riepilogo degli obiettivi di apprendimento .....	401
	Elementi di libreria presentati in questo capitolo .....	402
	Esercizi di riepilogo e approfondimento .....	402
	Esercizi di programmazione .....	407
<b>Capitolo 8 – Progettazione di classi .....</b>	<b>415</b>	
	Obiettivi del capitolo .....	415
8.1	Individuare le classi .....	416
8.2	Progettare buoni metodi .....	417
8.2.1	Un’interfaccia pubblica coesa .....	417
8.2.2	Minimizzare le dipendenze .....	418
8.2.3	Tenere distinti accessi e modifiche .....	419
8.2.4	Minimizzare gli effetti collaterali .....	420
8.3	Problem Solving: progettare i dati di un oggetto .....	426
8.3.1	Gestione di un totale .....	427
8.3.2	Conteggio di eventi .....	428
8.3.3	Gestione di una raccolta di valori .....	429
8.3.4	Gestione delle proprietà di un oggetto .....	429
8.3.5	Oggetti con stati diversi .....	430
8.3.6	Descrizione della posizione di un oggetto .....	431
8.4	Variabili statiche e metodi statici .....	433
8.5	Problem Solving: iniziare da un problema più semplice .....	438
8.6	Pacchetti .....	443

8.6.1	Organizzare classi in pacchetti .....	444
8.6.2	Importare pacchetti .....	444
8.6.3	Nomi di pacchetto .....	445
8.6.4	Pacchetti e file di codice sorgente .....	446
8.7	Ambienti per il collaudo di unità.....	450
	Riepilogo degli obiettivi di apprendimento .....	453
	Esercizi di riepilogo e approfondimento .....	454
	Esercizi di programmazione .....	458
<b>Capitolo 9 – Ereditarietà .....</b>		<b>463</b>
	Obiettivi del capitolo .....	463
9.1	Gerarchie di ereditarietà.....	464
9.2	Realizzare sottoclassi .....	468
9.3	Sovrascrivere metodi .....	474
9.4	Polimorfismo .....	480
9.5	La superclasse universale: <code>Object</code> .....	499
9.5.1	Sovrascrivere il metodo <code>toString</code> .....	499
9.5.2	Il metodo <code>equals</code> .....	501
9.5.3	L'operatore <code>instanceof</code> .....	502
	Riepilogo degli obiettivi di apprendimento .....	508
	Esercizi di riepilogo e approfondimento .....	508
	Esercizi di programmazione .....	510
<b>Capitolo 10 – Interfacce .....</b>		<b>513</b>
	Obiettivi del capitolo .....	513
10.1	Uso di interfacce per il riutilizzo di algoritmi .....	514
10.1.1	Individuare un tipo interfaccia .....	514
10.1.2	Dichiarare un tipo interfaccia.....	515
10.1.3	Implementare un tipo interfaccia .....	517
10.1.4	Confronto tra ereditarietà e interfacce.....	519
10.2	Programmare con le interfacce .....	524
10.2.1	Conversione da classe a interfaccia .....	524
10.2.2	Invocare metodi con variabili interfaccia .....	525
10.2.3	Conversione da interfaccia a classe .....	526
10.3	L'interfaccia <code>Comparable</code> .....	529
10.4	Usare interfacce di smistamento ( <i>callback</i> ).....	535
10.5	Classi interne .....	541
10.6	Oggetti semplificati.....	544
10.7	Gestione di eventi .....	545
10.7.1	Ricezione di eventi .....	546
10.7.2	Classi interne come ricevitori di eventi.....	548
10.8	Costruire applicazioni dotate di pulsanti .....	552
10.9	Eventi di temporizzazione .....	556
10.10	Eventi del mouse .....	559
	Riepilogo degli obiettivi di apprendimento .....	566
	Elementi di libreria presentati in questo capitolo .....	568

Esercizi di riepilogo e approfondimento .....	568
Esercizi di programmazione .....	571
<b>Capitolo 11 – Ingresso/uscita e gestione delle eccezioni.....</b>	<b>575</b>
Obiettivi del capitolo .....	575
11.1 Leggere e scrivere file di testo .....	576
11.2 Acquisire e scrivere testi .....	582
11.2.1 Acquisire parole .....	582
11.2.2 Acquisire caratteri .....	583
11.2.3 Classificare caratteri .....	583
11.2.4 Acquisire righe .....	584
11.2.5 Analizzare una stringa .....	585
11.2.6 Convertire stringhe in numeri .....	586
11.2.7 Evitare errori nell’acquisizione di numeri .....	586
11.2.8 Acquisire numeri, parole e righe .....	587
11.2.9 Impaginare i dati in uscita .....	588
11.3 Argomenti sulla riga dei comandi .....	591
11.4 Gestire eccezioni.....	604
11.4.1 Lanciare eccezioni .....	604
11.4.2 Catturare eccezioni .....	604
11.4.3 Eccezioni a controllo obbligatorio .....	608
11.4.4 Chiudere ricorse .....	609
11.4.5 Progettare eccezioni .....	610
11.5 Applicazione: gestione di errori in ingresso .....	615
Riepilogo degli obiettivi di apprendimento .....	620
Elementi di libreria presentati in questo capitolo .....	621
Esercizi di riepilogo e approfondimento .....	622
Esercizi di programmazione .....	623
<b>Capitolo 12 – Ricorsione .....</b>	<b>627</b>
Obiettivi del capitolo .....	627
12.1 Numeri triangolari .....	628
12.2 Metodi ausiliari ricorsivi .....	641
12.3 L’efficienza della ricorsione .....	643
12.4 Permutazioni .....	649
12.5 Ricorsione mutua .....	654
12.6 Backtracking .....	661
Riepilogo degli obiettivi di apprendimento .....	674
Esercizi di riepilogo e approfondimento .....	674
Esercizi di programmazione .....	675
<b>Capitolo 13 – Ordinamento e ricerca.....</b>	<b>679</b>
Obiettivi del capitolo .....	679
13.1 Ordinamento per selezione .....	680
13.2 Prestazioni dell’ordinamento per selezione .....	683
13.3 Analisi delle prestazioni dell’ordinamento per selezione .....	687

13.4	Ordinamento per fusione (MergeSort) .....	692
13.5	Analisi dell'algoritmo di ordinamento per fusione.....	695
13.6	Effettuare ricerche.....	700
13.6.1	Ricerca lineare .....	700
13.6.2	Ricerca binaria .....	702
13.7	Problem Solving: stima del tempo di esecuzione di un algoritmo.....	706
13.7.1	Algoritmi lineari.....	706
13.7.2	Algoritmi quadratici .....	708
13.7.3	Lo schema triangolare.....	709
13.7.4	Algoritmi logaritmici.....	711
13.8	Ordinamento e ricerca nella libreria Java .....	713
13.8.1	Ordinamento.....	713
13.8.2	Ricerca binaria .....	714
13.8.3	Confronto di oggetti.....	714
	Riepilogo degli obiettivi di apprendimento .....	722
	Elementi di libreria presentati in questo capitolo .....	723
	Esercizi di riepilogo e approfondimento .....	723
	Esercizi di programmazione .....	727
	<b>Capitolo 14 –Java Collections Framework .....</b>	<b>729</b>
	Obiettivi del capitolo .....	729
14.1	Una panoramica del Collections Framework .....	730
14.2	Liste concatenate.....	733
14.2.1	La struttura delle liste concatenate.....	733
14.2.2	La classe <code>LinkedList</code> del Java Collections Framework .....	734
14.2.3	Iteratori per liste.....	735
14.3	Insiemi .....	739
14.3.1	Scegliere un'implementazione di insieme.....	739
14.3.2	Lavorare con insiemi.....	742
14.4	Mappe .....	745
14.5	Pile, code e code prioritarie .....	755
14.5.1	Pile .....	755
14.5.2	Code .....	756
14.5.3	Code prioritarie .....	757
14.6	Applicazioni di pile e code .....	759
14.6.1	Accoppiamento delle parentesi.....	760
14.6.2	Valutazione di espressioni RPN .....	760
14.6.3	Valutazione di espressioni algebriche .....	762
14.6.4	Backtracking .....	765
	Riepilogo degli obiettivi di apprendimento .....	768
	Elementi di libreria presentati in questo capitolo .....	769
	Esercizi di riepilogo e approfondimento .....	770
	Esercizi di programmazione .....	772

**Capitolo 15 – Programmazione generica (online)**

Appendice A – Il sottoinsieme Basic Latin di Unicode .....	777
Appendice B – Linguaggio Java: operatori .....	779
Appendice C – Linguaggio Java: parole riservate.....	781
Appendice D – Sistemi di numerazione.....	783
Glossario .....	791
Indice analitico .....	799

**Appendice E – Linguaggio Java: linee guida per la codifica (online)****Appendice F – Linguaggio Java: compendio sintattico (online)****Progetti di programmazione (online)****Risposte alle domande di auto-valutazione (online)**

Il materiale online è disponibile all'indirizzo

<http://www.apogeoeducation.com/concetti-di-informatica-e-fondamenti-di-java-6a-ed.html>



# Presentazione della edizione italiana



La sesta edizione italiana di questo fortunato testo di Cay Horstmann si allinea con l'ottava edizione del testo originale, nella quale l'Autore ha introdotto, tra tanti miglioramenti, l'adeguamento alla recente versione Java 8 del linguaggio, nonostante la maggior parte degli esempi sia perfettamente funzionante anche con le versioni 6 e 7, ancora molto utilizzate.

Il massimo impegno dell'Autore è stato questa volta rivolto alla presentazione di "casi pratici" completi, che guidano il lettore neofita alla soluzione progettuale di un problema attraverso una successione di fasi ben delineate, suddividendo in modo appropriato le responsabilità dei singoli oggetti. Le sezioni speciali di tipo "Consigli pratici" ed "Esempi completi", dedicate proprio a questo, agevoleranno senza dubbio lo studente nel suo percorso di apprendimento.

Questa edizione vede anche una rivisitazione quasi completa del materiale didattico relativo all'ereditarietà e al polimorfismo, posticipando, rispetto all'edizione precedente, la presentazione delle interfacce, argomento peraltro arricchito da sezioni dedicate alle interfacce funzionali e all'introduzione delle espressioni lambda, una novità di Java 8 particolarmente apprezzata dai programmati professionisti.

Tra le scelte precedenti che sono state confermate, vale la pena citare il fatto che la programmazione grafica era, e rimane, completamente facoltativa: un arricchimento per i corsi universitari che dedicano più crediti a questo insegnamento di base, senza porre vincoli che potrebbero risultare troppo pesanti per altri e senza costringere i docenti a intervenire per isolare in modo artificioso gli argomenti (e gli esercizi) da evitare.

Il testo rimane perfettamente adeguato sia a un percorso universitario, con la guida di un docente, sia all'apprendimento autonomo, grazie alla presenza di una vasta raccolta di esercizi con difficoltà graduata in modo esplicito, nuovamente ampliata in questa edizione.

Proprio gli esercizi costituiscono, a mio modesto parere, uno dei tanti “valori aggiuntivi” che questo volume porta in dote in un corso o, più in generale, in un percorso di studio. Innanzitutto, al termine di ogni paragrafo lo studente viene invitato a confrontarsi con domande di auto-valutazione, semplici ma assolutamente non banali, la cui risposta, spesso articolata, è disponibile nel sito web dedicato alla versione italiana del libro. Poi, al termine di ogni capitolo, vengono proposti esercizi di varia tipologia e di diverso livello, fino ad arrivare a veri e propri progetti di programmazione che potranno impegnare lo studente in modo del tutto adeguato.

*Marcello Dalpasso*

*Professore Associato di Sistemi per l'Elaborazione dell'Informazione*

*Dipartimento di Ingegneria dell'Informazione - Scuola di Ingegneria*

*Università degli Studi di Padova*

# Prefazione



Questo libro è un testo introduttivo di programmazione in Java che focalizza l'attenzione sui principi fondamentali e sull'efficacia dell'apprendimento. Il libro è stato ideato per essere utile a un'ampia gamma di studenti, adeguandosi ai loro interessi e alle loro capacità, ed è adatto al primo insegnamento sulla programmazione di calcolatori in corsi di informatica, ingegneria e altre discipline. Non è richiesta alcuna precedente esperienza nella programmazione e serve solamente una normale preparazione matematica, tipica delle scuole medie superiori.

Queste sono le caratteristiche principali del testo:

- **Si inizia subito con gli oggetti, procedendo con gradualità.**  
Nel Capitolo 2 gli studenti imparano a usare oggetti e classi della libreria standard, mentre nel Capitolo 3 apprendono le strategie per la realizzazione di classi a partire da specifiche assegnate, usando poi semplici oggetti per approfondire la conoscenza di diramazioni, cicli e array. La vera e propria progettazione orientata agli oggetti inizia nel Capitolo 8. Un approccio così graduale consente agli studenti di utilizzare oggetti anche mentre studiano la parte algoritmica di base, senza ricorrere all'insegnamento di cattive abitudini che dovrebbero essere abbandonate in seguito.
- **Guide mirate ed esempi completi portano gli studenti al successo.**  
Spesso i programmatore principianti chiedono: "Come comincio? E ora cosa devo fare?" Un'attività complessa come la programmazione non può, ovviamente, ridursi a un ricettario di istruzioni, però guide mirate e dettagliate possono essere estremamente utili per acquisire confidenza con la materia, costituendo uno schema da tenere

sotto mano durante la soluzione dei problemi assegnati. Il libro contiene un'ampia rassegna di tali guide, denominate “Consigli pratici” e focalizzate su problemi frequenti, seguite dalla presentazione di ulteriori “Esempi completi”.

- **Le strategie di soluzione dei problemi vengono rese esplicite.**  
L'illustrazione pratica, passo dopo passo, di tecniche informatiche aiutano gli studenti a individuare e valutare soluzioni diverse ai problemi di programmazione. Queste strategie, presentate nel punto in cui possono apparire più interessanti, sono per molti studenti il modo per arrivare al successo nella programmazione.
- **L'esercizio rende migliori.**  
Ovviamente gli studenti che si cimentano nella programmazione devono essere in grado di realizzare programmi non banali, ma prima di tutto devono convincersi di poterlo fare. Questo libro contiene, al termine di ogni paragrafo, un numero rilevante di domande di auto-valutazione, e i riferimenti di tipo “Per far pratica” suggeriscono esercizi di cui tentare la soluzione.
- **Un approccio visuale motiva il lettore e ne agevola il percorso.**  
Le molte illustrazioni presenti nel libro spesso descrivono passo dopo passo le operazioni eseguite da programmi complessi. Speciali riquadri, graficamente molto evidenti, presentano la sintassi del linguaggio e le molte tabelle contengono in forma compatta una grande varietà di casi tipici e speciali. Prima di concentrarsi sull'attenta lettura del testo scritto, una rapida occhiata alle immagini consente di farsi un'idea sull'argomento che si sta per studiare.
- **Concentrarsi sugli aspetti essenziali, pur rimanendo tecnicamente corretti.**  
Una trattazione encyclopedica non è di alcun aiuto per un programmatore principiante, ma non lo è nemmeno un approccio opposto, che riduca il materiale a un elenco di punti semplificati. In questo libro gli aspetti essenziali di ciascun argomento sono presentati con dimensioni digeribili, aggiungendo note che consentono di approfondire le buone pratiche di programmazione o le caratteristiche del linguaggio, una volta che il lettore sia pronto per tali informazioni aggiuntive. Non troverete semplificazioni artificiose che diano soltanto l'illusione di aver compreso un argomento.
- **È bene insistere sulle buone pratiche.**  
Il libro è letteralmente disseminato di suggerimenti utili in merito alla qualità del software e la sottolineatura degli errori maggiormente ricorrenti incoraggia lo sviluppo di buone abitudini di programmazione. Il percorso opzionale relativo al collaudo si concentra sullo sviluppo di programmi che siano facilmente collaudabili, suggerendo con forza agli studenti di collaudare in modo sistematico i propri programmi.
- **La programmazione grafica è presentata in un percorso opzionale.**  
Le forme grafiche sono splendidi esempi di oggetti e molti studenti amano scrivere programmi grafici o, comunque, dotati di interfaccia grafica per l'interazione con l'utente. Se voluto, questi argomenti possono essere integrati nell'insegnamento, usando il materiale che si trova alla fine dei Capitoli 2, 3 e 10.
- **Gli studenti si confrontano anche con esercizi di ambito scientifico ed economico.**  
Gli esercizi presenti alla fine di ogni capitolo sono stati arricchiti con problemi definiti in ambito scientifico ed economico: progettati per attrarre gli studenti, mostrano l'importanza della programmazione in vari campi applicativi.

## Le novità di questa edizione

Java 8 ha introdotto nel linguaggio molte caratteristiche interessanti e questa ottava edizione del libro (e sesta edizione italiana) è stata aggiornata per sfruttarle al meglio. Ora le interfacce hanno metodi statici e predefiniti, e le espressioni lambda rendono molto più agevole la creazione di esemplari di interfacce che abbiano un solo metodo. Il capitolo relativo alle interfacce e i paragrafi che trattano il problema dell'ordinamento dei dati sono stati aggiornati per usare queste nuove possibilità, ancorché in modo facoltativo.

Inoltre sono state integrate nel testo alcune utili caratteristiche di Java 7, come l'enunciato `try` con risorse.

## Una panoramica del libro

Il libro può essere suddiviso in tre parti in modo molto naturale, come si può vedere nella Figura 1, dove vengono evidenziate graficamente anche le propedeuticità fra i capitoli, che consentono la medesima flessibilità che caratterizzava l'edizione precedente.

### Parte A: I fondamenti (Capitoli da 1 a 7)

Il Capitolo 1 contiene una breve introduzione relativa all'informatica e alla programmazione in Java, il Capitolo 2 mostra come manipolare oggetti di classi predefinite e il Capitolo 3 insegna a progettare e realizzare semplici classi a partire da specifiche assegnate.

I capitoli che vanno dal 4 al 7 trattano i tipi di dati fondamentali, le strutture sintattiche per il controllo di flusso (diramazioni e cicli) e gli array.

### Parte B: Progettazione orientata agli oggetti (Capitoli da 8 a 11)

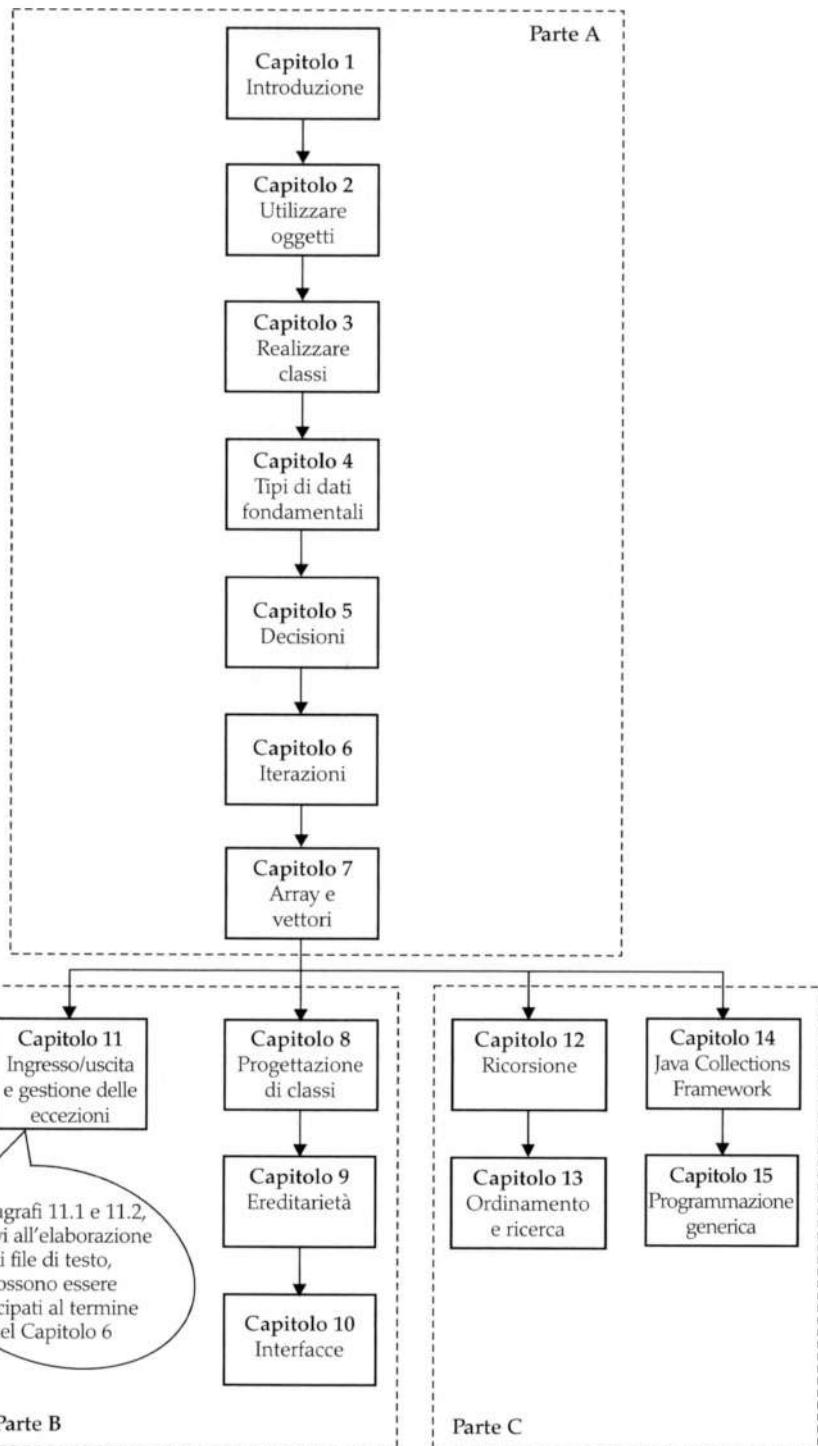
Il Capitolo 8 si occupa della progettazione di classi in modo più sistematico e introduce un sottoinsieme (molto semplificato) della notazione UML. La discussione sul polimorfismo e l'ereditarietà è suddivisa in due capitoli: il Capitolo 9 tratta, appunto, l'ereditarietà e il polimorfismo, mentre il Capitolo 10 si occupa delle interfacce. La gestione delle eccezioni (la cui gerarchia costituisce un ottimo esempio di ereditarietà) e gli strumenti basilari per l'elaborazione dei file di testo sono argomento del Capitolo 11.

### Parte C: Algoritmi e strutture di dati (Capitoli da 12 a 15)

I Capitoli 12, 13 e 14 contengono un'introduzione agli algoritmi e alle strutture di dati, con la trattazione di ricorsione, ordinamento e ricerca, liste concatenate, insiemi, mappe, pile e code. Come si può vedere nella Figura 1, tali argomenti possono essere trattati, se così si preferisce, anche dopo il Capitolo 7.

La presentazione della ricorsione, nel Capitolo 12, inizia con esempi molto semplici e arriva ad applicazioni significative, che sarebbero di difficile realizzazione senza ricorsione. Il Capitolo 13 introduce l'analisi delle prestazioni mediante la notazione O-grande e si occupa degli algoritmi di ordinamento fondamentali, con prestazioni quadratiche, oltre all'ordinamento per fusione.

**Figura 1**  
Dipendenze fra i capitoli



Il Capitolo 14 presenta le più importanti strutture per la memorizzazione di dati nel contesto della libreria standard di Java, nella sua sezione denominata Java Collections Framework: vengono delineate le astrazioni più importanti della libreria (come le liste, gli iteratori, gli insiemi e le mappe), oltre alle caratteristiche delle prestazioni delle diverse implementazioni di raccolte di dati.

Infine, il Capitolo 15, disponibile sul sito web dedicato al libro, presenta le caratteristiche del linguaggio Java che consentono la programmazione generica: si tratta di un capitolo adatto a studenti che vogliono realizzare proprie classi generiche o propri metodi generici.

## Appendici

Le prime appendici contengono materiale di riferimento sulla codifica dei caratteri, sugli operatori e sulle parole riservate del linguaggio Java, per proseguire con la rappresentazione binaria dei numeri e le operazioni sui bit. Altre appendici, disponibili nel sito web dedicato al libro, riportano un utile compendio sintattico del linguaggio Java e linee guida per la scrittura del codice sorgente.

## Sito web dedicato al libro

All'indirizzo <http://www.apogeeducation.com/concetti-di-informatica-e-fondamenti-di-java-6a-ed.html> si trova il sito web dedicato a questo libro, dove è possibile scaricare il codice sorgente degli esempi utilizzati nel testo, insieme ad altre risorse per studenti e docenti, tra le quali: le risposte alle domande di auto-valutazione, ulteriori progetti di programmazione relativi a ciascun capitolo del libro, un capitolo dedicato alla programmazione generica e alcune appendici.

## Una panoramica degli ausili didattici

Questa nuova edizione è basata sugli elementi pedagogici già utilizzati nell'edizione precedente, con l'aggiunta di ulteriori ausili per il lettore, rendendo il testo adeguato sia ai principianti sia a chi voglia apprendere Java come secondo linguaggio di programmazione.

L'inizio di ciascun capitolo offre la consueta panoramica sugli obiettivi del capitolo stesso e le motivazioni introduttive.

Note a margine evidenziano i punti in cui vengono introdotti nuovi concetti, costituiscono uno schema delle idee chiave e sono poi riassunte alla fine di ogni capitolo, suddivise per obiettivi di apprendimento: possono essere molto utili per un rapido ripasso.

I riquadri sintattici forniscono una rapida panoramica dei costrutti del linguaggio: già presenti nelle precedenti edizioni, sono stati arricchiti da annotazioni che evidenziano gli errori più frequenti e le migliori consuetudini.

Ogni paragrafo si conclude con esercizi di auto-valutazione, che lo studente può usare per verificare di aver compreso i nuovi argomenti. Le relative risposte si trovano nel sito web dedicato al libro.

Molte tabelle agevolano il compito dei principianti, fornendo esempi concreti, per un veloce ripasso delle caratteristiche principali del linguaggio e dei relativi errori.

frequenti. Un aiuto analogo viene da figure che illustrano, in fasi successive, il flusso di esecuzione di parti salienti di un programma, affiancandosi al codice sorgente di molti casi concreti.

I paragrafi esplicitamente dedicati alla “soluzione dei problemi” (*problem solving*) insegnano tecniche per elaborare nuove idee e per valutare le soluzioni individuate, spesso usando carta e penna o altri strumenti concreti. L’obiettivo è anche quello di porre l’enfasi sul fatto che molte delle attività di pianificazione e di risoluzione dei problemi possono essere svolte senza un computer.

Esistono, poi, diverse tipologie di sezioni speciali inserite nel testo, evidenziate in modo particolare per non interrompere il flusso del materiale principale.

- I **Consigli pratici**, ispirati dalle guide “HOW-TO” di Linux, hanno lo scopo di rispondere a domande frequenti degli studenti, del tipo “Cosa devo fare ora?”, fornendo istruzioni sull’esecuzione dei compiti più comuni, passo dopo passo, concentrandosi in particolar modo sulla pianificazione dell’attività di programmazione e sul relativo collaudo.
- Gli **Esempi completi** applicano i Consigli pratici a un esempio diverso, illustrando come quei consigli siano effettivamente assai utili per pianificare, realizzare e collaborare la soluzione di un altro problema di programmazione.
- Gli **Errori comuni** descrivono i tipi di errori compiuti spesso dagli studenti, con una spiegazione del motivo dell’errore e di come rimediарvi.
- I **Suggerimenti per la programmazione** illustrano buone abitudini di programmazione e spingono gli studenti a essere più produttivi.
- Gli **Argomenti avanzati** trattano materiale non essenziale o più complesso, oltre a fornire ulteriori spiegazioni relativamente ad alcuni argomenti già trattati.
- Le **Note per Java 8** presentano caratteristiche presenti soltanto a partire dalla nuova versione Java 8 del linguaggio.
- Le sezioni **Computer e società** forniscono informazioni storiche e sociali sull’informatica.

## Ringraziamenti

I miei ringraziamenti vanno a Bryan Gambrel, Don Fowley, Jenny Welter, Jessy Moor, Jennifer Lartz, Billy Ray e Tim Lindner di John Wiley & Sons, oltre a Vickie Piercey del gruppo Publishing Services, per l’aiuto che hanno dato a questo progetto. Un ringraziamento speciale e profondo va a Cindy Johnson per l’incredibile cura con cui ha lavorato e per i suoi preziosi consigli.

Sono grato a Jose Cordova, *The University of Louisiana at Monroe*, Suzanne Dietrich, *Arizona State University, West Campus*, Mike Domaratzki, *University of Manitoba*, Guy Helmer, *Iowa State University*, Peter Lutz, *Rochester Institute of Technology*, Carolyn Schauble, *Colorado State University*, Brent Seales, *University of Kentucky*, e Brent Wilson, *George Fox University*, per il loro eccellente contributo alla realizzazione del materiale di supporto.

Devo, poi, ringraziare le numerosissime persone che hanno rivisto il manoscritto di questa edizione, dando utili consigli e portando alla mia attenzione un numero imbarazzante di errori e omissioni. Tra tutti, citerò:

Robin Carr, *Drexel University*

Gerald Cohen, *The Richard Stockton College of New Jersey*

Aaron Keen, *California Polytechnic State University, San Luis Obispo*

Aurelia Smith, *Columbus State University*

Aakash Taneja, *The Richard Stockton College of New Jersey*

Craig Tanis, *University of Tennessee at Chattanooga*

Katherine Winters, *University of Tennessee at Chattanooga*

Ogni nuova edizione si basa sui suggerimenti e sulle esperienze dei revisori e degli utenti delle edizioni precedenti. Sono, quindi, grato a tutti quanti per il loro prezioso contributo.



# 1

## Introduzione



### Obiettivi del capitolo

---

- Conoscere i calcolatori e comprendere il significato dell'attività di programmazione
- Compilare e mettere in esecuzione il primo programma Java
- Riconoscere gli errori che si manifestano durante la compilazione e durante l'esecuzione
- Descrivere algoritmi facendo uso di pseudocodice

Proprio come quando si analizza un problema e, dopo aver elaborato un progetto, si mettono insieme gli strumenti necessari per risolverlo, in questo capitolo acquisirete le informazioni necessarie per iniziare l'apprendimento della programmazione. Dopo una breve panoramica sugli elementi hardware e software di un computer, nonché sulla programmazione in generale, vedrete come scrivere e come eseguire il vostro primo programma Java, imparando a diagnosticare e a correggere gli errori di programmazione. Infine, progettando programmi per il calcolatore, imparerete a usare lo pseudocodice per descrivere algoritmi, che sono una descrizione del metodo di soluzione di un problema, passo dopo passo.

## 1.1 Calcolatori e programmi

I calcolatori eseguono istruzioni estremamente elementari a velocità molto elevata.

Probabilmente avete già usato un computer (o calcolatore), per lavoro o per svago: molte persone lo utilizzano per attività quotidiane, come calcolare il saldo di un conto corrente bancario o scrivere un elaborato di fine semestre. I computer sono ottimi per tutto questo, perché possono gestire operazioni ripetitive, come sommare numeri o inserire parole in una pagina, senza annoiarsi o stancarsi.

La flessibilità che caratterizza i calcolatori è un fenomeno davvero affascinante: la stessa macchina può calcolare il saldo di un conto corrente, stampare una relazione o eseguire un gioco. Al contrario, altre macchine svolgono una gamma di attività decisamente più limitata: un'automobile viaggia, un tostapane tosta il pane. I computer sono in grado di svolgere compiti così diversi perché eseguono programmi diversi, ciascuno dei quali istruisce il calcolatore per risolvere un ben determinato problema.

Di per sé, un computer è una macchina che immagazzina dati (numeri, parole, immagini), interagisce con dispositivi (lo schermo, gli altoparlanti, la stampante) ed esegue programmi. Un **programma per calcolatore** spiega con grande dettaglio al calcolatore quale sia la sequenza di passi che devono essere messi in atto per risolvere un problema. Chiamiamo **hardware** l'insieme costituito dalla parte fisica del calcolatore e dai suoi dispositivi periferici, mentre i programmi eseguiti sono il **software**.

Gli attuali programmi per computer sono talmente sofisticati che è difficile credere che siano composti interamente da operazioni elementari estremamente semplici. Ecco qualche esempio di queste operazioni:

- accendi un punto rosso in una determinata posizione dello schermo;
- somma questi due numeri;
- se questo valore è negativo, prosegui il programma con quella specifica istruzione.

Gli utenti dei computer hanno l'illusione di un'interazione fluida con la macchina per il semplice motivo che un programma contiene un numero elevatissimo di tali istruzioni elementari, che vengono eseguite a una grandissima velocità.

La progettazione e la realizzazione di programmi per computer sono due attività che prendono il nome di **programmazione**. Studiando questo libro scoprirete come programmare un calcolatore, cioè come istruirlo perché risolva specifici problemi.

La programmazione di un gioco sofisticato che preveda animazioni o di un elaboratore di testi che consenta l'inserimento di figure e l'utilizzo di caratteri con stili diversi è un compito complesso che richiede una squadra di molti programmati altamente specializzati. I vostri primi esercizi di programmazione saranno decisamente più elementari, ma i concetti e le competenze che apprenderete costituiscono comunque una base fondante significativa e non dovete demoralizzarvi quando scoprirete che i vostri primi programmi non possono competere con il software professionale con cui siete soliti confrontarvi quotidianamente. In effetti, scoprirete che anche le attività di programmazione più semplici sono molto stimolanti: è un'esperienza assai gratificante osservare il computer mentre svolge con precisione e rapidità un'attività che avrebbe richiesto ore di fatica, così come constatare che piccole modifiche apportate a un programma possono produrre miglioramenti immediati. Percepirete il computer come un'estensione dei vostri poteri intellettuali.

Un programma per computer è una sequenza di istruzioni e decisioni.

La programmazione consiste nella progettazione e nella realizzazione di programmi per computer.



## Auto-valutazione

1. Cosa si utilizza per riprodurre musica mediante un computer?
2. Perché un riproduttore di CD è meno versatile di un computer?
3. Quali elementi di programmazione deve conoscere un utente di calcolatori per poter utilizzare un videogioco per computer?

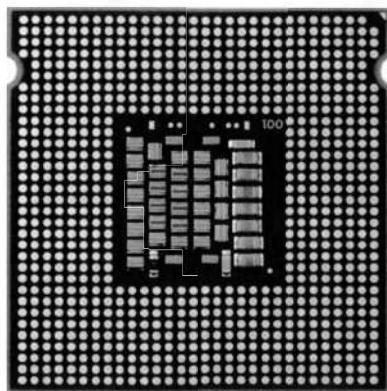
## 1.2 L'anatomia di un calcolatore

Per comprendere il processo di programmazione, è necessario conoscere almeno sommariamente gli elementi costitutivi di un computer, pertanto daremo uno sguardo a un personal computer (le macchine di maggiori dimensioni sono dotate di componenti più veloci, più grandi o più potenti, ma hanno sostanzialmente la stessa struttura).

Nel cuore del calcolatore si trova l'**unità centrale di elaborazione (CPU, central processing unit)**, di cui vedete un esempio in Figura 1), i cui collegamenti interni sono estremamente complessi. Ad esempio, il processore Intel Core (una CPU per personal computer molto diffusa nel momento in cui il libro è stato scritto) contiene parecchie centinaia di milioni di elementi strutturali, detti transistori (*transistor*).

**Figura 1**

Unità centrale  
di elaborazione (CPU)



La CPU si occupa del controllo dell'esecuzione dei programmi e dell'elaborazione dei dati, cioè individua ed esegue le istruzioni del programma: effettua le operazioni aritmetiche, come addizioni, sottrazioni, moltiplicazioni e divisioni, e recupera i dati dalla memoria esterna o dai dispositivi periferici, per poi memorizzarvi i risultati delle elaborazioni.

I dispositivi di memorizzazione  
sono la memoria (principale)  
e la memoria secondaria.

Esistono due tipi di memoria (*memory*, o “spazio di archiviazione”, *storage*). La *memoria principale*, detta anche semplicemente “memoria”, è costituita da circuiti elettronici che sono in grado di memorizzare dati quando sono alimentati elettricamente. La *memoria secondaria*, che generalmente è un **disco rigido (hard disk, in Figura 2)** oppure un dispositivo allo stato solido (SSD, *solid-state drive*), consente un'archiviazione dei dati meno costosa e più lenta, che, però, viene mantenuta anche in assenza di elettricità. Un disco rigido è formato da piatti rotanti, rivestiti di materiale magnetico, mentre un dispositivo di memorizzazione allo stato solido usa componenti elettronici che sono in grado di

preservare l'informazione memorizzata anche in assenza di alimentazione elettrica e non contengono parti meccaniche in movimento.

Per interagire con un utente umano, un computer ha bisogno di dispositivi periferici. Il calcolatore comunica informazioni verso l'esterno, cioè "in uscita" (*output*), mediante uno schermo di visualizzazione, altoparlanti o stampanti, mentre l'utente può fornire informazioni al computer, cioè "in ingresso" (*input*), tramite una tastiera o un dispositivo di puntamento, quale un mouse.

Alcuni computer sono unità autosufficienti, mentre altri sono connessi tra loro tramite **reti**. Attraverso le connessioni alla rete, il computer può leggere programmi da una postazione centralizzata oppure inviare dati ad altri computer: per l'utente di un computer connesso alla rete, non sempre può essere semplice distinguere i dati che risiedono sulla propria macchina da quelli che vengono trasmessi attraverso la rete stessa.

**Figura 2**

Disco rigido



La Figura 3 presenta uno schema dell'architettura di un personal computer. Le istruzioni dei programmi e i relativi dati (quali testi, numeri, sequenze audio o video) sono conservati nella memoria secondaria o in qualche punto della rete. Quando viene avviato un programma, le sue istruzioni vengono copiate nella memoria, dove la CPU le può leggere: la CPU legge ed esegue un'istruzione per volta. A seconda delle direttive espresse da tali istruzioni, la CPU legge dati, li modifica e li scrive nuovamente nella memoria principale o nella memoria secondaria; inoltre, alcune istruzioni del programma indurranno la CPU a visualizzare punti sullo schermo o a mettere in funzione l'altoparlante. Poiché queste azioni si ripetono molte volte e a grande velocità, l'utente umano percepisce immagini e suoni. Alcune istruzioni ricevono i comandi dell'utente tramite la tastiera, il mouse, il microfono o il sensore a sfioramento: il programma esamina la natura di questi comandi ed esegue le istruzioni conseguenti.

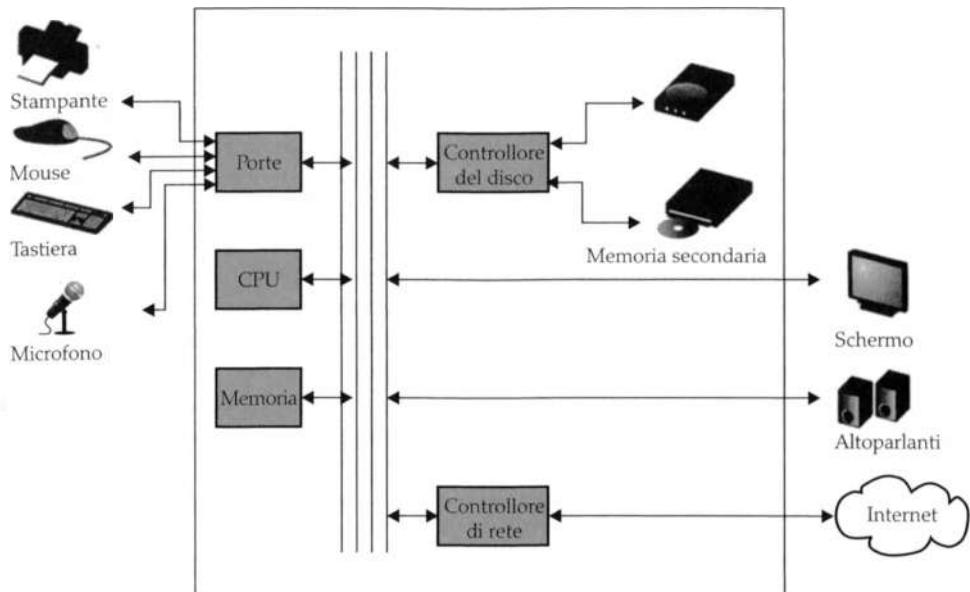


### Auto-valutazione

4. Dove è memorizzato un programma che non sia in esecuzione?
5. Quale parte del computer esegue le operazioni aritmetiche, come l'addizione e la moltiplicazione?

**Figura 3**

Diagramma schematico di un personal computer



6. Uno *smartphone* è, in effetti, un computer, analogo ai calcolatori da tavolo. Quali componenti di uno smartphone possono essere messi in corrispondenza con i dispositivi evidenziati nella Figura 3?



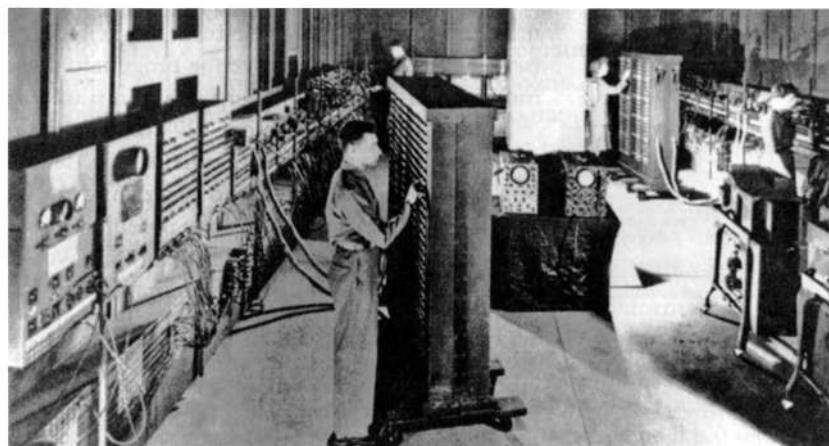
## Computer e società 1.1

### I computer sono ovunque

Quando, negli Anni Quaranta del secolo scorso, vennero inventati i primi calcolatori, ciascuno di essi occupava una stanza intera. L'ENIAC (*Electronic Numerical Integrator And Computer*, integratore numerico e calcolatore elettronico) venne completato nel 1946 presso la University of Pennsylvania e fu usato dall'esercito statunitense per il calcolo delle traiettorie dei proiettili. In effetti, anche oggi le apparecchiature di calcolo utilizzate dai motori di ricerca, dai negozi di Internet e dai *social network* riempiono interi edifici, detti *data center*, cioè centri per l'elaborazione dei dati. All'altro estremo di questo spettro continuo di possibilità, tro-

viamo computer in tutto ciò che ci circonda. I telefoni cellulari contengono un computer, così come le carte di credito e le tessere di abbonamento

per i trasporti pubblici. Una moderna automobile contiene diversi calcolatori, per controllare il motore, i freni, le luci e la radio.



L'ENIAC

L'avvento del calcolo ubiquo o diffuso (*ubiquitous computing*) ha cambiato molti aspetti della nostra vita. Nelle fabbriche lavoravano persone che svolgevano attività di montaggio ripetitive, che oggi vengono portate a termine da robot controllati da computer, il cui funzionamento è controllato da poche persone, che sanno come lavorare con tali calcolatori. Oggigiorno si usano computer per leggere libri, per ascoltare musica e per guardare film, e i calcolatori vengono utilizzati anche per produrre tali contenuti: il libro che state leggendo non

avrebbe potuto essere scritto senza l'ausilio di computer.

La conoscenza dei calcolatori e della loro programmazione è diventata essenziale per la carriera di molte persone. Gli ingegneri progettano automobili controllate da computer e apparati medicali che sono in grado di salvare vite. Gli informatici sviluppano programmi che aiutano le persone a conoscersi e a mettersi insieme per fornire supporto a iniziative sociali: ad esempio, alcuni attivisti usano i *social network* per condividere filmati che mostrano abusi commessi da regimi

repressivi e queste informazioni sono davvero utili per influenzare la pubblica opinione.

Mentre i calcolatori, piccoli e grandi, diventano sempre più presenti nella vita di tutti i giorni, diventa analogamente più importante che ciascuno di noi sappia come funzionano e come si possano utilizzare. Usando questo testo per imparare a programmare i calcolatori, raggiungerete un buon livello di comprensione dei fondamenti dell'informatica, cosa che vi consentirà di diventare cittadini più consapevoli e, forse, informatici professionisti.

### 1.3 Il linguaggio di programmazione Java

Scrivere un programma per computer, pertanto, significa fornire alla CPU una sequenza di istruzioni che possano essere eseguite. Un programma per computer, quindi, è costituito da un gran numero di semplici istruzioni, la cui elencazione dettagliata è noiosa, rendendo probabile la produzione di numerosi errori. Per tale motivo sono stati ideati i **linguaggi di programmazione di alto livello**, mediante i quali si possono specificare le azioni che devono essere svolte da un programma, dopodiché sarà un **compilatore** a tradurre le istruzioni di alto livello nelle più dettagliate istruzioni necessarie alla CPU, che vengono chiamate **codice macchina**. Negli anni sono stati progettati molti diversi linguaggi di programmazione di alto livello, per gli scopi più disparati.

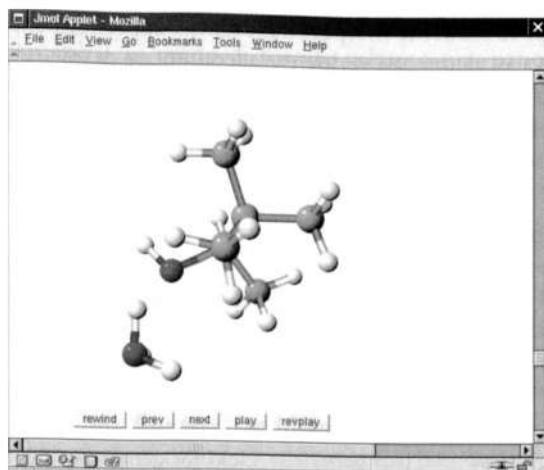
Nel 1991, un gruppo di lavoro all'interno di Sun Microsystems, guidato da James Gosling e Patrick Naughton, progettò un linguaggio, chiamato in codice "Green", da utilizzare all'interno degli elettrodomestici, come gli apparecchi "intelligenti" per televisori (*set-top box*). Il linguaggio venne progettato per essere semplice, sicuro e utilizzabile con molti tipi di processori diversi, ma non si trovò mai alcun cliente per questa tecnologia.

Gosling racconta che nel 1994 la squadra si rese conto che avrebbe potuto "scrivere un *browser* davvero eccellente; nel filone delle applicazioni client/server, era una delle poche cose che aveva bisogno di alcuni degli inconsueti risultati ottenuti: neutralità rispetto all'architettura, esecuzione in tempo reale, affidabilità, sicurezza". Java fu presentato a una folla entusiasta durante l'evento SunWorld del 1995, unitamente a un browser che poteva eseguire **applet**: piccoli programmi, scritti in linguaggio Java, che potevano trovarsi all'interno di qualunque pagina di un sito di Internet e che venivano eseguiti durante la visualizzazione della pagina stessa, producendo risultati come quello mostrato nella figura visibile nella pagina seguente.

Da quel giorno in avanti, il linguaggio Java si è imposto a un ritmo fenomenale. I programmatore l'hanno adottato perché è più semplice del suo rivale che più gli assomiglia,

Java venne progettato per programmare semplici elettrodomestici, ma il suo primo utilizzo di grande successo fu la scrittura di applet per Internet.

Un applet per visualizzare molecole



il linguaggio C++. Inoltre, Java ha una ricca **libreria** (nota del traduttore: nonostante la traduzione del termine inglese *library* sia *biblioteca*, useremo la traduzione *libreria* perché ormai consolidata nell'informatica italiana), che permette di scrivere programmi portabili (cioè "trasferibili") da un sistema operativo all'altro, una funzionalità attesa con ansia da quanti volevano essere indipendenti dai sistemi proprietari e decisamente osteggiata dai loro fornitori. Le diverse versioni della libreria, spaziando da una *micro edition* a una *enterprise edition*, rendono possibile il lavoro dei programmatore in situazioni che vanno dai più piccoli dispositivi integrati, come le *smart card*, ai più grandi server di Internet.

Java fu progettato per essere sicuro e portabile, a vantaggio sia degli utenti di Internet sia degli studenti.

Poiché Java è stato progettato per Internet, ha due qualità che lo rendono molto adatto per i principianti: sicurezza e portabilità. L'ambiente Java consente a chiunque di eseguire programmi all'interno del proprio browser senza alcun timore: le caratteristiche di sicurezza previste dal linguaggio Java garantiscono l'interruzione dell'esecuzione di un programma nel momento in cui questo tenti di compiere azioni potenzialmente non sicure. Un ambiente di esecuzione così sicuro è molto utile anche per chi sta imparando il linguaggio: eseguendo un programma che contiene un errore che possa produrre un comportamento non sicuro, si assiste alla sua terminazione prematura e si ottiene una dettagliata segnalazione d'errore.

L'altro vantaggio di Java è la portabilità: il medesimo programma Java verrà eseguito in ambiente Windows, UNIX, Linux o Macintosh senza bisogno di alcuna modifica. Per consentire la portabilità, il compilatore Java non traduce i programmi Java direttamente in istruzioni per la CPU: i programmi Java compilati contengono, invece, istruzioni eseguibili dalla **macchina virtuale Java** (*Java virtual machine*, JVM), un programma che simula il funzionamento di una CPU reale. La portabilità è un ulteriore vantaggio per lo studente: non è necessario imparare a scrivere programmi per piattaforme diverse.

Allo stato attuale, Java si è imposto stabilmente come uno dei più importanti linguaggi per la programmazione in ambito generale e per l'apprendimento dell'informatica. Tuttavia, sebbene Java sia un buon linguaggio per principianti, non è perfetto, per tre ragioni.

Dal momento che Java non è stato progettato specificatamente per gli studenti, non è stato pensato per essere veramente semplice da utilizzare nella scrittura dei programmi più elementari ed è necessaria una certa dose di tecnicismi per scrivere anche il più semplice dei programmi: questo non è un problema per un programmatore professionista, ma è uno

I programmi Java vengono distribuiti sotto forma di istruzioni per una macchina virtuale, risultando così indipendenti dalla piattaforma.

svantaggio per lo studente inesperto. Durante l'apprendimento della programmazione in linguaggio Java, in alcuni casi vi dovrete accontentare di spiegazioni preliminari, attendendo un successivo capitolo per conoscere tutti i particolari.

Java è stato esteso più volte, come si può vedere nella Tabella 1: si assume, in questo testo, la disponibilità di Java nella versione 7 o successiva.

**Tabella 1**  
Versioni di Java

Versione	Anno	Principali novità
1.0	1996	
1.1	1997	Classi interne
1.2	1998	Swing, Collections
1.3	2000	Miglioramento delle prestazioni
1.4	2002	Asserzioni, XML
5	2004	Classi generiche, ciclo <code>for</code> generico, auto-incapsulamento, enumerazioni e annotazioni
6	2006	Miglioramenti della libreria
7	2011	Piccole modifiche al linguaggio e miglioramenti della libreria
8	2014	Espresioni lambda, flussi, nuovi oggetti per il tempo e le date

Java ha una libreria molto vasta:  
focalizzate la vostra attenzione  
sull'apprendimento di quelle sezioni  
della libreria di cui avete bisogno  
per i vostri progetti  
di programmazione.

Infine, non potete sperare di imparare tutto di Java frequentando un unico corso. Di per sé, il linguaggio Java è relativamente semplice, ma è corredata da un'enorme raccolta di **pacchetti di libreria** (*library package*), che sono necessari per scrivere programmi utili. Si tratta di pacchetti per la visualizzazione grafica, per la progettazione di interfacce di interazione con l'utente del programma, per la crittografia, per la connessione in rete, per l'audio, per la memorizzazione di basi di dati e per molti altri scopi. Anche i programmatore di Java più esperti non possono sperare di conoscere il contenuto di tutti i pacchetti: si limitano a utilizzare quelli che servono per ciascun progetto specifico.

Usando questo libro apprenderete una grande quantità di nozioni sul linguaggio Java e sui pacchetti più importanti della sua libreria, ma ricordate che l'obiettivo principale del testo non è quello di farvi imparare a memoria le minuzie di Java, bensì di insegnarvi a ragionare sulla programmazione.



### Auto-valutazione

7. Quali sono i due principali vantaggi del linguaggio Java?
8. Quanto tempo occorre per imparare a utilizzare l'intera libreria di Java?

### Per far pratica

A questo punto si consiglia di svolgere l'esercizio R.1.5, al termine del capitolo.

## 1.4 L'ambiente di programmazione Java

Molti studenti vivono come un problema il fatto di dover usare, come programmatore, strumenti software molto diversi da quelli a cui sono abituati: semplicemente, è necessario prendere confidenza con l'ambiente di programmazione. Dato che le piattaforme

Dedicate un po' di tempo all'apprendimento dell'ambiente di programmazione che dovrete utilizzare.

Un editor è un programma per inserire e modificare un testo, come un programma Java.

disponibili (cioè i computer e i loro sistemi operativi) sono molto diverse tra loro, qui siamo in grado di fornire soltanto una panoramica generica dei passi da seguire: è decisamente opportuno cercare di partecipare a un'attività di laboratorio che fornisca istruzioni operative o, in alternativa, chiedere l'aiuto di un amico esperto.

### Fase 1 Mettere in esecuzione l'ambiente di sviluppo Java.

I diversi sistemi operativi richiedono azioni significativamente differenti per svolgere questo primo compito. Molti computer dispongono di un **ambiente di sviluppo integrato** (*integrated development environment*, IDE), con il quale è possibile scrivere programmi e collaudarli. In altri, invece, bisogna innanzitutto eseguire un **editor**, cioè un programma che svolge una funzione analoga a un elaboratore di testi (*word processor*), con il quale si scrivono le istruzioni del programma in linguaggio Java, poi è necessario aprire una **finestra per l'esecuzione di comandi** o "finestra di terminale" (*console window*), dove digitare i comandi che eseguono il programma realizzato. Dovete scoprire da soli come compiere questi primi passi nel vostro computer.

### Fase 2 Scrivere un semplice programma.

Per convenzione consolidata, nell'apprendere un nuovo linguaggio di programmazione si inizia con un programma che visualizza un semplice saluto: "Hello, World!". Seguiamo la tradizione: ecco il programma "Hello, World!" in Java.

```
public class HelloPrinter
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World!");
    }
}
```

Nel paragrafo successivo analizzeremo questo programma.

Indipendentemente dall'ambiente di programmazione utilizzato, l'attività di progettazione di un programma inizia scrivendone gli enunciati all'interno della finestra di un editor.

Create un nuovo file e chiamatelo `HelloPrinter.java`, usando il metodo più adeguato al vostro ambiente di programmazione (se, oltre al nome del file, vi viene chiesto di fornire un nome per il "progetto", chiamatelo `hello`). Copiate le istruzioni del programma *esattamente* come le trovate scritte nel libro, oppure cercate la copia digitale dei programmi contenuti nel libro, messa a disposizione dalla casa editrice, e inserite nell'editor il programma usando un'operazione di "copia e incolla".

Java distingue tra maiuscolo e minuscolo, quindi bisogna fare altrettanto, con attenzione.

Scrivendo il programma, occorre fare attenzione ai diversi simboli utilizzati, ricordando che Java *distingue fra maiuscolo e minuscolo*, quindi le lettere, maiuscole e minuscole, vanno inserite esattamente come appaiono nel testo: non si può scrivere `MAIN` oppure `PrintLn`. Se non si sta attenti, si manifestano dei problemi (come riportato nella sezione Errori comuni 1.2).

**Fase 3** Eseguire il programma.

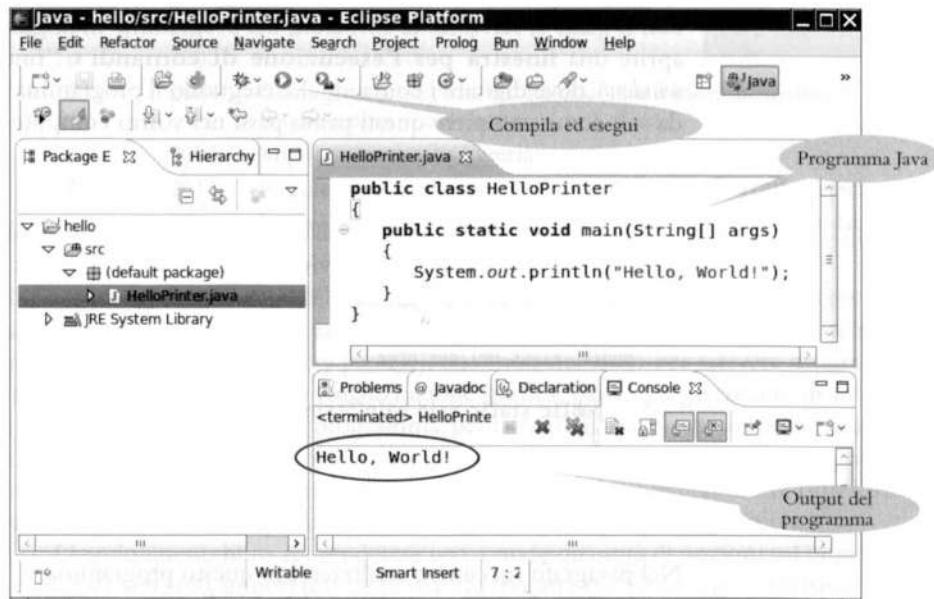
Le azioni richieste per eseguire il programma dipendono in gran parte dall'ambiente di programmazione che utilizzate: potrebbe essere necessario selezionare un pulsante con il mouse oppure scrivere qualche comando. Ciò fatto, da qualche parte sullo schermo comparirà il messaggio

Hello, World!

La posizione esatta del messaggio dipende, di nuovo, dall'ambiente di programmazione, come si può vedere nei due esempi delle Figure 4 e 5.

**Figura 4**

Esecuzione del programma HelloPrinter in un ambiente di sviluppo integrato



**Figura 5**

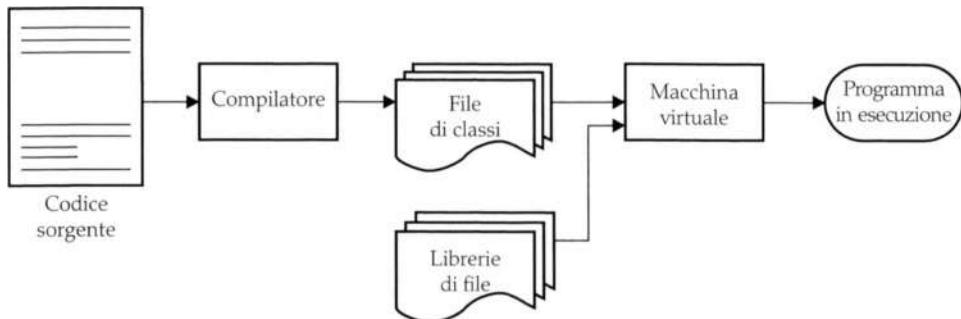
Esecuzione del programma HelloPrinter in una finestra di terminale



Il compilatore Java traduce il codice sorgente in file di classi, contenenti istruzioni per la macchina virtuale Java.

Prima di poter eseguire il programma, il compilatore Java traduce il **file sorgente** (*source file*), cioè gli enunciati che avete scritto, in **file di classi** (*class file*), contenenti istruzioni per la macchina virtuale Java. Dopo che il compilatore ha tradotto il **codice sorgente** (*source code*) in istruzioni per la macchina virtuale, questa le può eseguire. Durante l'esecuzione, la macchina virtuale accede a una libreria di codice precompilato, che contiene, tra le altre cose, l'implementazione delle classi `System` e `PrintStream`, necessarie per

**Figura 6**  
Dal codice sorgente al programma in esecuzione



visualizzare i messaggi prodotti in uscita dal programma. La Figura 6 riassume il processo di creazione ed esecuzione di un programma Java. In alcuni ambienti di programmazione il compilatore e la macchina virtuale sono sostanzialmente invisibili al programmatore e vengono chiamati in causa automaticamente ogni volta che si richiede l'esecuzione di un programma Java, mentre in altri occorre eseguire in modo esplicito sia il compilatore sia la macchina virtuale.

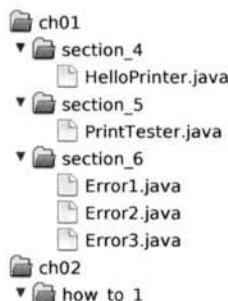
#### Fase 4 Organizzare il lavoro.

Nel vostro lavoro di programmatore scrivete programmi, provate ad eseguirli e cercate di migliorarli. Durante queste fasi, i programmi sono memorizzati in **file**, a loro volta memorizzati in **cartelle** (*folder* o *directory*). Una cartella, poi, può contenere file e altre cartelle, le quali, a loro volta, possono contenere ulteriori file e cartelle, come si può vedere nella Figura 7. Questa struttura gerarchica di archiviazione può essere anche molto vasta e non è necessario che ne conosciate tutte le diramazioni, ma è importante che creiate cartelle per organizzare il vostro lavoro. Una valida strategia consiste nella creazione di una cartella dedicata ai programmi dell'intero corso e, al suo interno, di una cartella diversa per ciascun programma realizzato.

Alcuni ambienti di programmazione conservano i programmi in una cartella predefinita e non c'è bisogno di definirla esplicitamente: in tal caso, è comunque importante che scopriate dove si trovano i vostri file.

Accertatevi di aver capito dove si trovano i vostri file all'interno della gerarchia di cartelle: tale informazione è essenziale, ad esempio, per poter inviare file al docente e per farne una copia di sicurezza (come evidenziato nella sezione Suggerimenti per la programmazione 1.1).

**Figura 7**  
Una gerarchia di cartelle





## Auto-valutazione

9. Dove si trova, nel vostro computer, il file `HelloPrinter.java`?
10. Cosa si fa per proteggersi dalla perdita di dati quando si lavora su progetti di programmazione?

## Per far pratica

A questo punto si consiglia di svolgere l'esercizio R1.6, al termine del capitolo.



## Suggerimenti per la programmazione 1.1

### Copie di sicurezza (backup)

Prima che si verifichi  
un disastro irrecuperabile,  
pianificate una strategia  
di backup del vostro lavoro.

Trascorrerete molte ore scrivendo programmi Java e cercando di migliorarli. È facile cancellare un file accidentalmente e a volte i file vanno perduti a causa di un guasto del computer. Riscrivere il contenuto di un file perduto è molto frustrante e fa perdere tempo, per cui è di importanza veramente cruciale imparare a salvaguardare i propri file e prendere l'abitudine di farlo *prima* che sia troppo tardi. Fare copie di sicurezza (o di *backup*) dei propri file, mettendoli in salvo in una chiavetta o da qualche parte nella rete Internet è una strategia semplice e comoda. Ecco alcuni punti da tenere presente:

- *Fate backup spesso.* Salvare un file richiede solo pochi secondi e, se dovete perdere molte ore per ricreare un lavoro che avreste potuto salvare facilmente, vi detesterete. Il consiglio è di mettere in sicurezza il vostro lavoro ogni trenta minuti circa.
- *Utilizzate i supporti di backup a rotazione.* Usate supporti fisici diversi per i backup, a rotazione: per prima cosa eseguite il salvataggio su un primo supporto, poi sul secondo, quindi sul terzo, per poi, infine, riutilizzare il primo. In questo modo avrete sempre a disposizione i tre salvataggi più recenti: anche se uno si rileverà difettoso, potrete usare uno dei rimanenti.
- *Fate attenzione alla direzione del backup.* L'azione di backup consiste nel copiare file da un posto a un altro: è importante farlo nella direzione corretta, copiando i file dalla cartella di lavoro alla cartella di backup. Se lo fate nel modo sbagliato, sovrascriverete un file più recente con una versione più vecchia.
- *Controllate i backup di tanto in tanto.* Controllate accuratamente che le vostre copie di backup siano dove pensate. Non c'è nulla di più frustrante della scoperta che, quando se ne ha bisogno, i backup non ci siano.
- *Rilassatevi prima di ripristinare.* Quando perdete un file e avete bisogno di ripristinarlo dal supporto di backup, probabilmente vi trovate in uno stato d'animo nervoso e poco sereno: fate un respiro profondo e riflettete sul processo di recupero prima di iniziare. Non è raro che un utente in preda all'agitazione cancelli il backup, nel tentativo di ripristinare un file danneggiato.

## 1.5 Analisi di un semplice programma

In questo paragrafo analizzeremo in dettaglio il nostro primo programma Java, di cui riportiamo qui il codice sorgente:

## File HelloPrinter.java

```

1 public class HelloPrinter
2 {
3     public static void main(String[] args)
4     {
5         // visualizza un saluto nella finestra di console
6
7         System.out.println("Hello, World!");
8     }
9 }
```

La prima riga

```
public class HelloPrinter
```

contiene la dichiarazione di una **classe** (*class*) di nome `HelloPrinter`.

Ogni programma Java è costituito da una o più classi e nei Capitoli 2 e 3 ne parleremo con maggiore dettaglio.

La parola `public` indica che la classe è utilizzabile “pubblicamente”; più avanti incontrerete le caratteristiche `private`.

In Java, ciascun file sorgente può contenere al massimo una classe pubblica, il cui nome deve corrispondere al nome del file che contiene la classe. Per esempio, la classe `HelloPrinter` deve essere contenuta nel file `HelloPrinter.java`. Il costrutto sintattico

```

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

In Java, le classi sono i blocchi fondamentali per la costruzione di programmi.

In ogni applicazione Java esiste una classe che ha il metodo `main`. Quando l'applicazione viene eseguita, vengono eseguite le istruzioni di tale metodo `main`.

Ciascuna classe contiene dichiarazioni di metodi e ogni metodo è costituito da una sequenza di istruzioni.

definisce un **metodo**, chiamato `main` (cioè “principale”). Un metodo è una raccolta di istruzioni di programmazione che descrivono come svolgere un determinato compito. Ciascuna applicazione Java deve avere un **metodo main**. La maggior parte dei programmi Java contiene altri metodi oltre a `main` e nel Capitolo 3 vedrete come scriverli.

La parola `static` verrà spiegata con grande dettaglio nel Capitolo 8, mentre il significato di `String[] args` sarà illustrato nel Capitolo 11. A questo punto del vostro apprendimento, dovreste considerare semplicemente

```

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

come una porzione della “infrastruttura” necessaria per scrivere qualsiasi programma Java. In questo primo programma, tutte le istruzioni si trovano all'interno del metodo `main` della classe.

Il metodo `main` contiene istruzioni, dette **enunciati** (*statement*). Ciascun enunciato termina con un carattere “punto e virgola”. Quando un programma viene eseguito, gli enunciati presenti nel metodo `main` sono eseguiti uno dopo l'altro, in ordine.

## Sintassi di Java

### 1.1 Programma Java

Ogni programma Java contiene un metodo `main` avente esattamente questa intestazione.

Quando viene eseguito un programma, vengono eseguiti uno dopo l'altro gli enunciati presenti all'interno del metodo `main`.

Le parentesi graffe aperte e chiuse devono corrispondere a coppie.

Ogni programma contiene almeno una classe, il cui nome va scelto in modo che descriva l'azione svolta dal programma.

```
public class HelloPrinter
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World!");
    }
}
```

Per scrivere programmi diversi, si può sostituire questo enunciato.

Ogni enunciato termina con un "punto e virgola" (sezione Errori comuni 1.1).

Nel nostro semplice programma, il metodo `main` ha un solo enunciato:

```
System.out.println("Hello, World!");
```

Questo enunciato visualizza (o "stampa") una riga di testo, vale a dire "Hello,World!". In questo enunciato vediamo che viene *invocato* (o "chiamato") un altro metodo, identificato (per motivi che ora non spiegheremo) dal nome `System.out.println`, in verità un po' lungo.

Non c'è bisogno che realizziamo questo metodo, perché i programmatore che hanno scritto la libreria di Java l'hanno già fatto per noi: vogliamo soltanto che tale metodo svolga il compito per il quale è stato progettato, cioè visualizzare un valore.

Quando, in Java, si invoca un metodo, occorre specificare:

- Per invocare un metodo se ne specificano il nome e i parametri (o argomenti).
- Il metodo che si vuole invocare (in questo caso, `System.out.println`).
  - Le altre informazioni eventualmente necessarie perché il metodo possa svolgere il suo compito (in questo caso, "Hello, World!"). Dal punto di vista tecnico, questa informazione viene chiamata **argomento** (*argument*, o parametro). Gli argomenti di un metodo sono racchiusi da una coppia di parentesi tonde e argomenti multipli sono separati da virgolette.

Una sequenza di caratteri racchiusa fra virgolette è detta **stringa**:

"Hello, World!"

Una stringa è una sequenza di caratteri racchiusa tra virgolette.

È necessario inserire il contenuto della stringa all'interno delle virgolette, in modo che il compilatore sappia che intendete scrivere letteralmente "Hello, World!". Esiste un motivo per questa regola: supponiamo di dover visualizzare la parola `main`. Se si racchiude la parola fra virgolette, "`main`", il compilatore capisce che intendete la sequenza di caratteri `m a i n`, non il metodo chiamato `main`. La regola stabilisce che dovete racchiudere semplicemente tutte le stringhe di testo fra virgolette, affinché il compilatore le consideri testo normale e non cerchi di interpretarle come istruzioni del programma.

Si possono visualizzare anche valori numerici. Per esempio, l'enunciato seguente valuta l'espressione `3 + 4` e visualizza il numero 7:

```
System.out.println(3 + 4);
```

Il metodo `System.out.println` visualizza una stringa o un numero, poi inizia una riga nuova. Per esempio, la sequenza di enunciati seguente:

```
System.out.println("Hello");
System.out.println("World!");
```

visualizza due righe di testo, in questo modo:

```
Hello
World!
```

Esiste un secondo metodo, chiamato `System.out.print`, che potete utilizzare per visualizzare un elemento senza che subito dopo venga iniziata una nuova riga. Per esempio, questi due enunciati:

```
System.out.print("00");
System.out.println(3 + 4);
```

visualizzano la singola riga seguente:

```
007
```



## Auto-valutazione

11. Come modifichereste il programma `HelloPrinter` in modo che rivolga un saluto a voi?
12. Come modifichereste il programma `HelloPrinter` in modo che visualizzi la parola "Hello" verticalmente?
13. Il programma continuerebbe a funzionare se la riga 7 venisse sostituita con l'enunciato seguente?

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

14. Cosa viene visualizzato dal seguente insieme di enunciati?

```
System.out.print("My lucky number is");
System.out.println(3 + 4 + 5);
```

15. Cosa viene visualizzato dagli enunciati seguenti?

```
System.out.println("Hello");
System.out.println("");
System.out.println("World");
```

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi R1.7, R1.8, E1.5 e E1.8, al termine del capitolo.



## Errori comuni 1.1

### Dimenticare i punti e virgola

In Java, ciascun enunciato deve terminare con un punto e virgola. Dimenticare di scriverlo è un errore frequente che disorienta il compilatore, che si basa sul punto e virgola per stabilire dove termina un enunciato e inizia il successivo: il compilatore, infatti, non usa le interruzioni di riga o le parentesi chiuse per identificare la fine degli enunciati. Per esempio, il compilatore tratta questi due enunciati

```
System.out.println("Hello")
System.out.println("World!");
```

come se costituissero un enunciato unico, cioè come se avessimo scritto così:

```
System.out.println("Hello") System.out.println("World!");
```

Di conseguenza, il compilatore non è in grado di comprendere l'enunciato, perché non si aspetta la parola `System` che segue la parentesi chiusa dopo la stringa `"Hello"`.

Il rimedio è semplice: scorrere tutti gli enunciati per verificare che terminino con un punto e virgola, proprio come si controlla che ciascuna frase in linguaggio naturale termini con il punto. Non bisogna, però, mettere un punto e virgola al termine delle righe che contengono `public class Hello` oppure `public static void main`: non sono enunciati.

## 1.6 Errori

Facciamo qualche esperimento con il programma `HelloPrinter`. Vediamo che cosa succede con errori di battitura di questo tipo:

```
System.ou.println("Hello, World!");
System.out.println("Hello, Word!");
```

Un errore di sintassi è una violazione  
delle regole del linguaggio  
di programmazione identificata  
dal compilatore.

Nel primo caso il compilatore protesterà, dicendo che non ha la più pallida idea di che cosa intendiate con `ou`. L'esatta formulazione del messaggio d'errore dipende dall'ambiente di programmazione, ma potrebbe assomigliare a "Cannot find symbol ou" (cioè "non sono in grado di trovare il simbolo ou"). Si tratta di un **errore in fase di compilazione** (*compile-time error*) o **errore di sintassi**: in base alle regole del linguaggio c'è qualcosa di sbagliato e il compilatore se ne accorge. Quando il compilatore individua errori, si rifiuta di tradurre il programma in istruzioni per la macchina virtuale Java e, di conseguenza, non c'è un programma da eseguire: bisogna rimediare all'errore e compilare di nuovo. Di fatto, il compilatore è piuttosto esigente e non è raro passare attraverso numerosi cicli di correzione degli errori di sintassi, prima di concludere con successo la compilazione per la prima volta.

Se il compilatore rileva un errore, non si limita a fermarsi e a rinunciare, ma tenta di segnalare tutti gli errori che riesce a trovare, in modo che possano essere corretti tutti in una volta sola.

Talvolta, tuttavia, un errore lo porta fuori strada. Osservate, ad esempio, quello che succede se vi dimenticate la virgolette che delimitano una stringa, scrivendo: `System.out.`

`println(Hello, World!).` Il compilatore non protesta per le virgolette mancanti, bensì segnala “Cannot find symbol Hello”. Sfortunatamente il compilatore non è molto furbo e non è in grado di capire che volevate usare una stringa: sta a voi capire che, per correggere l'errore, dovete racchiudere la stringa tra virgolette.

L'errore presente nella seconda riga è di natura diversa. Il programma verrà compilato e potrà essere eseguito, ma, erroneamente, visualizzerà:

```
Hello, Word!
```

Si ha un errore logico quando un programma esegue un'azione non prevista dal programmatore.

Questo è un **errore in fase di esecuzione** (*run-time error*): il programma è sintatticamente corretto e fa qualcosa, ma non quello che dovrebbe fare. Dal momento che gli errori in fase di esecuzione sono provocati da errori nella progettazione della logica sottostante al funzionamento del programma, vengono spesso chiamati **errori logici**.

In questo caso specifico, l'errore in fase di esecuzione non produce un messaggio d'errore: semplicemente, produce un risultato errato. Altri errori logici, invece, sono tanto gravi da generare un'**eccezione**: un messaggio d'errore emesso dalla macchina virtuale Java. Ad esempio, se il vostro programma contiene questo enunciato

```
System.out.println(1 / 0);
```

otterrete, durante la sua esecuzione, il messaggio d'errore “Division by zero” (*divisione per zero*).

Durante lo sviluppo dei programmi gli errori sono inevitabili. Quando un programma supera le poche righe, occorre una concentrazione sovrumanica per digitarlo correttamente, senza incorrere in alcuna svista. Vi sorprenderete a dimenticare punti e virgola e virgolette più spesso di quanto vi piacerebbe ammettere, ma il compilatore scoverrà questi errori per voi.

Gli errori logici sono più fastidiosi. Il compilatore non li rileva (di fatto, il compilatore convertirà allegramente qualsiasi programma, se la sintassi è corretta), ma il programma risultante farà qualcosa di sbagliato: è responsabilità dell'autore del programma collaudarlo e scoprire eventuali errori logici.



## Auto-valutazione

16. Immaginate di dimenticare le virgolette che delimitano la stringa `Hello, World!` nel programma `HelloPrinter.java`. È un errore di sintassi o un errore logico?
17. Immaginate di scrivere, nel programma `HelloPrinter.java`, `printline` invece di `println`. È un errore di sintassi o un errore logico?
18. Immaginate di scrivere, nel programma `HelloPrinter.java`, `hello` invece di `main`. È un errore di sintassi o un errore logico?
19. Utilizzando il computer, vi sarà capitato di usare un programma che, poi, “è morto” (andato in *crash*, cioè terminato bruscamente e spontaneamente) o “si è bloccato” (cioè ha smesso di rispondere alle sollecitazioni in ingresso). Si tratta di un comportamento catalogabile come errore di sintassi o come errore logico?
20. Perché non si può collaudare un programma per scoprire errori logici se presenta errori di sintassi?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R1.9, R1.10 e R1.11, al termine del capitolo.



## Errori comuni 1.2

### Errori di ortografia

Se si sbaglia accidentalmente l'ortografia di una parola, possono succedere strane cose e non sempre è facile capire dai messaggi di errore che cosa è andato storto. Ecco un buon esempio che illustra come banali errori di ortografia possano essere fastidiosi:

```
public class HelloPrinter
{
    public static void Main(String[] args)
    {
        System.out.println("Hello, World!");
    }
}
```

Questa classe dichiara il metodo `Main`. Il compilatore non lo identifica come metodo `main`, perché `Main` inizia con una lettera maiuscola e il linguaggio Java è sensibile alla differenza tra lettere maiuscole e minuscole: le lettere maiuscole e minuscole sono considerate completamente diverse fra loro e, per il compilatore, `Main` non corrisponde a `main` più di quanto non vi corrisponda `rain`. Il metodo `Main` verrà compilato senza proteste ma, quando la macchina virtuale Java leggerà il file compilato, si lamenterà dell'assenza del metodo `main` e rifiuterà di eseguire il programma. Naturalmente, il messaggio “missing main method” (*il metodo main è assente*) dovrebbe fornire un indizio utile per cercare l'errore.

Se un messaggio di errore sembra indicare che il compilatore o la macchina virtuale siano fuori strada, è il caso di controllare l'ortografia e le lettere maiuscole e minuscole. Se si sbaglia a scrivere il nome di un simbolo (per esempio, `ou` al posto di `out`), il compilatore genererà il messaggio d'errore “`Cannot find symbol ou`”: questo messaggio è, solitamente, un buon indizio di un possibile errore di ortografia.

## 1.7 Problem Solving: progettazione di algoritmi

Presto imparerete ad eseguire calcoli e a prendere decisioni mediante la programmazione in Java. Prima, però, di prendere in esame, nei capitoli successivi, gli strumenti che consentono di eseguire calcoli, vediamo come si possano descrivere le fasi che sono necessarie per trovare la soluzione di un problema.

### 1.7.1 Il concetto di algoritmo

Può darsi che vi sia capitato di vedere annunci pubblicitari che esortano a pagare per un servizio computerizzato che trova “l'anima gemella”, analizzando profili personali e individuandovi corrispondenze. Provate a pensare a come questo possa funzionare: dovete compilare un questionario e inviarlo, così come faranno altre persone; i dati così raccolti vengono, poi, elaborati al computer da un programma. Ha senso ipotizzare che

il computer sia in grado di individuare le persone che meglio corrispondono al vostro profilo? Immaginate che sia vostro fratello più giovane, invece del computer, ad avere sulla propria scrivania tutti i questionari compilati: che istruzioni gli dareste? Non potete pensare di dirgli, semplicemente “Trova la persona più carina che ama pattinare e navigare in Internet”, perché non esiste un metodo oggettivo per valutare la bellezza ed è molto probabile che l’opinione di vostro fratello (così come quella di un computer che analizzi fotografie digitalizzate di potenziali partner) sia diversa dalla vostra. Se non siete in grado di fornire a qualcuno istruzioni scritte che consentano di risolvere un problema, non c’è modo che il computer possa trovare magicamente una soluzione: il computer può fare soltanto ciò che gli dite di fare, anche se è in grado di farlo più velocemente e senza annoiarsi o affaticarsi.

Proprio per questo motivo, non c’è alcuna garanzia che un servizio computerizzato che trova “l’anima gemella” possa fornire il risultato migliore: sarà, però, in grado di presentare a un utente un insieme di potenziali partner che ne condividono gli interessi principali. Quest’ultimo è un problema che un computer può decisamente risolvere.

Per fornire una risposta a un problema che preveda il calcolo di una risposta, un programma per calcolatore deve seguire una sequenza di passi che:

- Non sia ambigua
- Sia eseguibile
- Arrivi a compimento (in un tempo finito)

Un algoritmo che risolve un problema  
è una sequenza di passi non ambigua,  
eseguibile e che termina  
in un tempo finito.

Un passo di una sequenza è *non ambiguo* quando contiene istruzioni precise in merito a cosa si debba fare e specifica con certezza quale sia il passo successivo, senza lasciare spazio alle opinioni personali e all’inventiva. Un passo è, poi, *eseguibile* quando può essere effettivamente realizzato. Ad esempio, un calcolatore è in grado di elencare tutte le persone che condividono i vostri interessi, ma non può predire quale di queste sarà il vostro partner per tutta la vita. Infine, una sequenza di passi arriva a compimento in un tempo finito se, prima o poi, termina: un programma che continua a funzionare senza mai produrre una risposta è, ovviamente, inutile.

Una sequenza di passi che non sia ambigua, che sia eseguibile e che termini in un tempo finito prende il nome di **algoritmo**. Anche se non esiste alcun algoritmo che sia in grado di individuare un partner perfetto, per molti altri problemi esistono algoritmi risolutivi: il prossimo paragrafo ne fornisce un esempio.

## 1.7.2 Un algoritmo che risolve un problema finanziario

Esaminiamo il seguente problema, relativo a un investimento finanziario:

Vengono depositati 10000 dollari in un conto bancario che fornisce il 5 per cento di interesse annuo. Quanti anni occorrono perché il saldo del conto arrivi al doppio dell’importo iniziale?

Sareste in grado di risolvere il problema facendo i calcoli manualmente? Dovreste senza dubbio riuscirci, ad esempio calcolando ripetutamente il saldo del conto, in questo modo:

Anno	Interesse	Saldo
0		10 000
1	$10\ 000.00 \times 0.05 = 500.00$	$10\ 000.00 + 500.00 = 10\ 500.00$
2	$10\ 500.00 \times 0.05 = 525.00$	$10\ 500.00 + 525.00 = 11\ 025.00$
3	$11\ 025.00 \times 0.05 = 551.25$	$11\ 025.00 + 551.25 = 11\ 576.25$
4	$11\ 576.25 \times 0.05 = 578.81$	$11\ 576.25 + 578.81 = 12\ 155.06$

Basta proseguire finché il saldo non è almeno pari a 20 000 dollari: a quel punto, la risposta è l'ultimo numero scritto nella colonna relativa all'anno.

Lo svolgimento di calcoli di questo tipo è, ovviamente, assai noioso, sia per voi sia per il vostro fratello più giovane, mentre i calcolatori si comportano ottimamente nell'esecuzione veloce di calcoli ripetitivi, senza fare errori. Ciò che è importante per il calcolatore è una descrizione dei passi necessari per trovare la soluzione del problema: ogni passo deve essere espresso in modo chiaro e non ambiguo, senza che ci sia bisogno di "indovinare". Ecco una descrizione di questo tipo:

Iniziare con anno uguale a 0 e saldo uguale a 10 000.

Anno	Interessi	Saldo
0		10 000

Ripetere i passi seguenti finché il saldo è minore di 20000

Aggiungere 1 al valore dell'anno.

Calcolare l'interesse come saldo per 0.05 (cioè il 5%).

Aggiungere l'interesse al saldo.

Anno	Interessi	Saldo
0		10 000
1	500.00	10 500
:		:
14	942.82	19 799.32
(15)	989.96	20 789.28

Riportare il valore finale dell'anno come risposta.

Questi passi non sono ancora descritti in un linguaggio comprensibile al calcolatore, ma imparerete ben presto a formularli in Java. Questa descrizione informale è detta **pseudocodice** e il paragrafo successivo prenderà in esame proprio le regole per scrivere algoritmi mediante pseudocodice.

### 1.7.3 Pseudocodice

Lo pseudocodice è una descrizione informale di una sequenza di passi che risolvono un problema.

Non esistono regole stringenti per lo pseudocodice, perché non è destinato alla lettura da parte di programmi, bensì di persone. Ecco i tipi di enunciati in pseudocodice che useremo nel libro:

- Per descrivere come viene impostato o modificato un valore, usiamo enunciati come questi:

costo totale = prezzo di acquisto  $\times$  costo operativo

Moltiplicare il valore del saldo per 1.05.

Eliminare dalla parola il primo e l'ultimo carattere.

- Decisioni e ripetizioni sono descritte in questo modo:

Se costo totale 1 < costo totale 2

Finché il saldo è minore di 20000

Per ogni immagine presente nella sequenza

Usiamo i rientri del testo verso destra (“indentazione”) per indicare quali enunciati siano oggetto di selezione o di ripetizione.

Per ogni automobile

costo operativo = 10  $\times$  costo annuale del carburante

costo totale = prezzo di acquisto  $\times$  costo operativo

L’indentazione usata in questo esempio indica che entrambi gli enunciati devono essere eseguiti “per ogni automobile”.

- Specifichiamo i risultati con enunciati simili a questi:

Scegliere automobile1.

Riportare come risposta il valore finale dell’anno.

#### 1.7.4 Dagli algoritmi ai programmi

Nel Paragrafo 1.7.2 abbiamo descritto, mediante pseudocodice, un metodo per scoprire il tempo necessario al raddoppio di un investimento. Proviamo a verificare con cura che quello pseudocodice rappresenti un algoritmo, cioè che sia una descrizione non ambigua, che sia eseguibile e che arrivi a conclusione in un tempo finito.

Il nostro pseudocodice non è ambiguo: semplicemente, descrive come debbano essere aggiornati i valori a ogni passo. Lo pseudocodice è eseguibile perché usa un tasso d’interesse fisso: se avessimo scritto di utilizzare, invece di un tasso fisso del 5%, l’effettivo tasso d’interesse osservato negli anni a venire, le istruzioni non sarebbero state eseguibili, perché non c’è modo di prevedere i tassi d’interesse futuri. Per verificare che la sequenza di passi arriva a conclusione è necessario fare un piccolo ragionamento: a ogni passo il saldo aumenta di almeno 500 dollari, per cui, prima o poi, deve raggiungere il valore di 20000 dollari.

Abbiamo, quindi, individuato un algoritmo che risolve il nostro problema finanziario, per cui sappiamo di poter trovare la soluzione anche programmando un calcolatore: l’esistenza di un algoritmo è prerequisito essenziale per la programmazione. Prima di iniziare a programmare, dovete individuare e descrivere un algoritmo adeguato al compito che volete risolvere. Nei prossimi capitoli imparerete a esprimere algoritmi usando il linguaggio Java.

**Figura 8**  
Procedimento per lo sviluppo di un programma



### Auto-valutazione

21. Nell'ipotesi che il tasso di interesse sia il 20 per cento, quanti anni servono per il raddoppio dell'investimento?
22. Il vostro operatore telefonico vi addebita \$29.95 per i primi 300 minuti di conversazione e \$0.45 per ogni ulteriore minuto, a cui va aggiunto il 12.5 per cento di tasse e contributi. Individuate un algoritmo per calcolare il costo mensile a partire dal tempo totale di conversazione, in minuti.
23. Esaminate lo pseudocodice seguente, che descrive una procedura per individuare la fotografia più bella all'interno di una sequenza:

Scegliere la prima foto e dichiarare che tale foto è la "migliore fino ad ora".

Per ogni foto della sequenza

Se è più bella della foto "migliore fino ad ora"

Eliminare la foto "migliore fino ad ora".

Dichiarare che la foto in esame è la "migliore fino ad ora".

La foto "migliore fino ad ora" è la più bella foto della sequenza.

Si tratta effettivamente di un algoritmo in grado di individuare la fotografia più bella?

24. Nell'ipotesi che, nella situazione descritta nella domanda precedente, ogni foto abbia un'etichetta che ne indica il prezzo, individuate un algoritmo che trovi la foto più costosa.
25. Immaginando di avere una sequenza casuale di biglie nere e bianche e di volerle disporre in modo che quelle nere siano consecutive, come quelle bianche, analizzate l'algoritmo seguente:

Ripetere finché l'ordine voluto non è stato raggiunto

Trovare la prima biglia nera che sia preceduta da una biglia bianca.

Scambiarle tra loro.

Come si comporta l'algoritmo con la sequenza O●O●●? Descrivete il funzionamento dell'algoritmo passo dopo passo fino al momento in cui termina.

26. Immaginando di avere una sequenza casuale di biglie colorate, analizzate l'algoritmo seguente:

Ripetere finché l'ordine voluto non è stato raggiunto

Trovare la prima biglia che sia preceduta da una biglia di colore diverso.

Scambiarle tra loro.

Perché questo non è un algoritmo?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R1.16, E1.4 e P1.1, al termine del capitolo.



## Consigli pratici 1.1

### Descrizione di un algoritmo mediante pseudocodice

Questa è la prima sezione di tipo “Consigli pratici” (*How to*) che trovate in questo libro e che, come vedrete, descrive passo dopo passo il procedimento che occorre seguire per portare a termine alcuni compiti specifici e importanti nell’attività di sviluppo di programmi per il calcolatore.

Prima di iniziare a scrivere un programma in Java occorre sviluppare un algoritmo, cioè un metodo che consenta di giungere alla soluzione di uno specifico problema. L'algoritmo si può descrivere mediante pseudocodice: una sequenza di passi ben precisi, espressi in linguaggio naturale. Per illustrare il procedimento, progetteremo un algoritmo per risolvere il problema seguente.

**Problema.** Dovete scegliere, per l’acquisto, tra due automobili, una delle quali è più efficiente dell’altra in merito al consumo di carburante, ma è più costosa. Vi sono noti il prezzo e l’efficienza (in miglia per gallone, mpg) di entrambe le vetture. Pensate di usare l’automobile per dieci anni, percorrendo 15000 miglia all’anno, con un prezzo del carburante di \$4 al gallone. Pagherete l’acquisto in contanti e non prendete in considerazione costi finanziari. Qual è l’acquisto migliore?

**Fase 1** Determinare i dati disponibili (*ingressi*) e i risultati da produrre (*uscite*).

Nel nostro esempio abbiamo questi valori di “ingresso”:

- prezzo 1 e efficienza 1, rispettivamente il prezzo di acquisto e l’efficienza (in mpg) della prima automobile
- prezzo 2 e efficienza 2, rispettivamente il prezzo di acquisto e l’efficienza (in mpg) della seconda automobile

Vogliamo semplicemente sapere quale automobile sia meglio acquistare: questo è il risultato cercato, cioè l’uscita da produrre.

**Fase 2** Scomporre il problema in compiti più semplici.

Dobbiamo sapere il costo totale di ciascuna automobile: facciamo i calcoli separatamente, per la prima e per la seconda. Noto il costo totale di ciascuna, potremo decidere quale sia la migliore.

Il costo totale, per ciascuna auto, è **prezzo + costo di esercizio**.

Nell'ipotesi che, per dieci anni, ci sia un utilizzo costante, con un prezzo costante del carburante, il costo (totale) di esercizio dipende dal costo di esercizio annuale.

Il costo di esercizio è  $10 \times$  spesa annuale.

La spesa annuale per carburante è **prezzo al gallone  $\times$  consumo annuale**.

Il consumo annuale è pari a **miglia all'anno/efficienza**. Ad esempio, guidando per 15 000 miglia con un'efficienza di 15 miglia/gallone, l'automobile consuma 1000 galloni.

**Fase 3** Descrivere ciascun sotto-problema mediante pseudocodice.

Nella descrizione precedente, i vari passi vanno elencati in modo che ogni valore intermedio venga calcolato prima che serva per altri calcoli. Ad esempio, il calcolo

**costo totale = prezzo + costo di esercizio**

deve essere eseguito dopo aver calcolato il **costo di esercizio**.

Ecco, infine, l'algoritmo completo che consente di decidere quale automobile costituisca l'acquisto migliore.

Per ogni automobile, calcolare il costo totale, in questo modo:

consumo annuale = miglia all'anno / efficienza

spesa annuale = prezzo al gallone  $\times$  consumo annuale

costo di esercizio =  $10 \times$  spesa annuale

costo totale = prezzo + costo di esercizio

Se costo totale 1 < costo totale 2

Scegliere automobile 1

Altrimenti

Scegliere automobile 2

**Fase 4** Collaudare lo pseudocodice in alcuni casi specifici.

Useremo, come esempio, questi valori:

Automobile 1: \$25 000, 50 miglia/gallone

Automobile 2: \$20 000, 30 miglia/gallone

Ecco i calcoli che forniscono il costo totale della prima automobile

consumo annuale = miglia all'anno / efficienza =  $15\ 000 / 50 = 300$

spesa annuale = prezzo al gallone  $\times$  consumo annuale =  $4 \times 300 = 1200$

$$\text{costo di esercizio} = 10 \times \text{spesa annuale} = 10 \times 1200 = 12\,000$$

$$\text{costo totale} = \text{prezzo} + \text{costo di esercizio} = 25\,000 + 12\,000 = 37\,000$$

Analogamente, il costo totale per la seconda automobile risulta essere \$40 000, per cui l'algoritmo produce come decisione la scelta della Automobile 1.

Il seguente “Esempio completo” mette all’opera i concetti illustrati nel capitolo e i passi delineati nella sezione Consigli pratici 1.1, risolvendo un nuovo problema: vedrete come si può sviluppare un algoritmo per pavimentare una stanza con piastrelle di colori diversi, realizzando uno schema a scacchiera. La sezione va letta con l’obiettivo di rivedere i concetti appresi e ricevere ulteriore aiuto per affrontare problemi nuovi: nei capitoli successivi troverete sezioni analoghe.



## Esempi completi 1.1

### Sviluppare un algoritmo per posare piastrelle su un pavimento

**Problema.** Dovete ricoprire il pavimento rettangolare di un bagno con piastrelle, alternativamente bianche e nere, quadrate, con lato di 4 pollici. Le dimensioni del pavimento, misurate in pollici, sono entrambe multiple di 4.

**Fase 1** Determinare i dati disponibili (*ingressi*) e i risultati da produrre (*uscite*).

I dati disponibili sono le dimensioni del pavimento (*lunghezza* × *larghezza*), misurate in pollici. Il risultato da produrre è un pavimento ricoperto di piastrelle.

**Fase 2** Scomporre il problema in compiti più semplici.

Un sotto-problema che si evidenzia in modo naturale è la posa di una fila di piastrelle. Se siete in grado di risolvere questo problema, allora potete portare a termine il compito iniziale posando una fila accanto all’altra, iniziando da un muro, finché non raggiungete il muro opposto.

Come si posa una fila di piastrelle? Iniziate posando una piastrella vicino a un muro. Se è bianca, posatele accanto una piastrella nera; se, invece, è nera, posatele accanto una piastrella bianca. Continuate finché non raggiungete il muro opposto. La fila conterrà un numero di piastrelle pari a *larghezza* / 4.

**Fase 3** Descrivere ciascun sotto-problema mediante pseudocodice.

Scrivendo lo pseudocodice, vogliamo essere ancora più precisi nell’enunciare dove vadano posizionate esattamente le piastrelle.

Posare una piastrella nera nell’angolo nord-ovest del pavimento.

Finché il pavimento non è completamente ricoperto, ripetere questi passi:

Ripetere questo passo ((*larghezza* / 4) – 1) volte:

Posare una piastrella a est di quella posata precedentemente:

se era bianca, prenderne una nera, altrimenti una bianca.

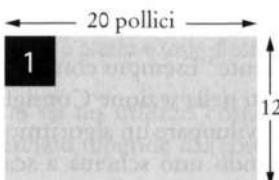
Identificare la piastrella iniziale della fila appena posata:

se a sud c’è spazio, posare là una piastrella di colore diverso.

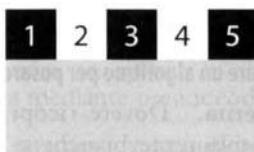
**Fase 4** Collaudare lo pseudocodice in alcuni casi specifici.

Immaginate di voler pavimentare una superficie che misura  $20 \times 12$  pollici.

Il primo passo consiste nella posa di una piastrella nera nell'angolo nord-ovest.



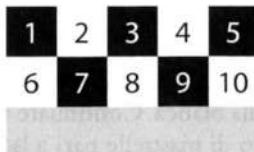
Poi, posare alternativamente quattro piastrelle, fino al raggiungimento del muro a est ( $\text{larghezza} / 4 - 1 = 20 / 4 - 1 = 4$ ).



Verso sud c'è ancora spazio. Individuare la piastrella che si trova all'inizio della riga appena completata: è nera, quindi posare una piastrella bianca a sud di essa.



Completare la riga.



Verso sud c'è ancora spazio. Individuare la piastrella che si trova all'inizio della riga appena completata: è bianca, quindi posare una piastrella nera a sud di essa.



Completare la riga.



A questo punto il pavimento è completamente ricoperto: il problema è risolto.

## Riepilogo degli obiettivi di apprendimento

### Definizione di "programma per calcolatore" e di "programmazione"

- I calcolatori eseguono istruzioni estremamente elementari a velocità molto elevata.
- Un programma per computer è una sequenza di istruzioni e decisioni.
- La programmazione consiste nella progettazione e nella realizzazione di programmi per computer.

### I componenti di un computer

- L'unità centrale di elaborazione (CPU) controlla l'esecuzione dei programmi e l'elaborazione dei dati.
- I dispositivi di memorizzazione sono la memoria (principale) e la memoria secondaria.

### Il processo di traduzione da linguaggi di alto livello a codice macchina

- Java venne progettato per programmare semplici elettrodomestici, ma il suo primo utilizzo di grande successo fu la scrittura di *applet* per Internet.
- Java fu progettato per essere sicuro e portatile, a vantaggio sia degli utenti di Internet sia degli studenti.
- I programmi Java vengono distribuiti sotto forma di istruzioni per una macchina virtuale, risultando così indipendenti dalla piattaforma.
- Java ha una libreria molto vasta: focalizzate la vostra attenzione sull'apprendimento di quelle sezioni della libreria di cui avete bisogno per i vostri progetti di programmazione.

### L'ambiente per la programmazione in Java

- Dedicate un po' di tempo all'apprendimento dell'ambiente di programmazione che dovete utilizzare.
- Un editor è un programma per inserire e modificare un testo, come un programma Java.
- Java distingue tra maiuscolo e minuscolo, quindi bisogna fare altrettanto, con attenzione.
- Il compilatore Java traduce il codice sorgente in file di classi, contenenti istruzioni per la macchina virtuale Java.
- Prima che si verifichi un disastro irrecuperabile, pianificate una strategia di backup del vostro lavoro.

### I blocchi costitutivi di un semplice programma

- In Java, le classi sono i blocchi fondamentali per la costruzione di programmi.
- In ogni applicazione Java esiste una classe che ha il metodo `main`. Quando l'applicazione viene eseguita, vengono eseguite le istruzioni di tale metodo `main`.
- Ciascuna classe contiene dichiarazioni di metodi e ogni metodo è costituito da una sequenza di istruzioni.
- Per invocare un metodo se ne specificano il nome e i parametri (o argomenti).
- Una stringa è una sequenza di caratteri racchiusa tra virgolette.

### Catalogazione degli errori: errori di sintassi ed errori logici

- Un errore di sintassi è una violazione delle regole del linguaggio di programmazione identificata dal compilatore.
- Si ha un errore logico quando un programma esegue un'azione non prevista dal programmatore.

### Scrittura di semplici algoritmi mediante pseudocodice

- Un algoritmo che risolve un problema è una sequenza di passi non ambigua, eseguibile e che termina in un tempo finito.
- Lo pseudocodice è una descrizione informale di una sequenza di passi che risolvono un problema.

## Elementi di libreria presentati nel capitolo

```
java.io.PrintStream
    print
    println
java.lang.System
    out
```

## Esercizi di riepilogo e approfondimento

- ★ **R1.1.** Spiegate la differenza che intercorre fra usare un programma al calcolatore e programmare un computer.
- ★ **R1.2.** Quali parti di un computer possono contenere il codice di un programma? E quali i dati da elaborare o elaborati?
- ★ **R1.3.** Quali parti di un computer hanno il compito di fornire informazioni all'utente? E quali acquisiscono dati forniti dall'utente?
- ★★ **R1.4.** Un tostapane è un dispositivo avente un'unica funzione, mentre un computer può essere programmato per svolgere molti compiti diversi. Il vostro telefono cellulare è un dispositivo che ha un'unica funzione oppure è un computer programmabile? La risposta dipende dal modello di telefono.
- ★★ **R1.5.** Illustrate due vantaggi derivanti dall'uso di codice Java rispetto al codice macchina.
- ★★ **R1.6.** Nel vostro computer personale, oppure in uno di quelli del laboratorio di informatica, trovate la posizione esatta (nome della cartella):
  - del file di esempio `HelloPrinter.java` che avete scritto con l'editor;
  - dell'esecutore di programmi Java, `java.exe` oppure semplicemente `java`;
  - del file `rt.jar`, che contiene la libreria di esecuzione (*run-time library*).
- ★★ **R1.7.** Cosa visualizza questo programma?

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.println("39 + 3");
        System.out.println(39 + 3);
    }
}
```

- \*\* **R1.8.** Cosa visualizza questo programma? Fate attenzione alle spaziature.

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.print("Hello");
        System.out.println("World");
    }
}
```

- \*\* **R1.9.** Quale errore di sintassi è presente in questo programma?

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.println("Hello", "World!");
    }
}
```

- \*\* **R1.10.** Scrivete tre versioni del programma `HelloPrinter.java`, con errori di sintassi diversi. Scrivetene una versione con un errore logico.

- \* **R1.11.** In che modo si scoprono gli errori di sintassi? E gli errori logici?

- \*\*\* **R1.12.** La mensa dell'università vende ai suoi clienti una tessera che dà diritto a un pasto gratis ogni volta che, durante un determinato periodo di tempo, si sono acquistati un certo numero di pasti. I dettagli dell'offerta sono variabili nel tempo. Descrivete un algoritmo che consenta di determinare se una particolare offerta sia economicamente vantaggiosa. Di quali dati avete bisogno?

- \*\* **R1.13.** Scrivete un algoritmo per rispondere a questa domanda: un conto bancario contiene inizialmente \$10000; gli interessi vengono calcolati mensilmente, al tasso annuo del 6% (cioè 0.5% al mese); ogni mese vengono prelevati \$500 per pagare le spese del convitto universitario; dopo quanti anni il conto è vuoto?

- \*\*\* **R1.14.** Considerate di nuovo l'esercizio precedente e supponete che i valori che vi figurano (\$10000, 6%, \$500) siano a discrezione dell'utente del programma. Esistono valori che impediscono all'algoritmo che avete sviluppato di terminare? In caso affermativo, modificate l'algoritmo in modo da avere la certezza che termini sempre.

- \*\*\* **R1.15.** Per fare il preventivo del costo di tinteggiatura di una casa, l'imbianchino deve conoscere l'area della sua superficie esterna. Sviluppate un algoritmo che calcoli tale valore, avendo come dati in ingresso l'altezza della casa e le sue due dimensioni orizzontali (chiamiamole lunghezza e larghezza), oltre al numero di finestre e porte e alle relative dimensioni (nell'ipotesi che tutte le finestre abbiano le stesse dimensioni, al pari di tutte le porte).

- \*\* **R1.16.** Nel problema descritto nei Consigli pratici 1.1, per confrontare automobili candidate all'acquisto si sono fatte ipotesi sul prezzo del carburante e sull'utilizzo annuale dell'automobile. In linea teorica, sarebbe meglio sapere quale vettura sia la migliore senza dover fare tali ipotesi. Perché un programma eseguito al calcolatore non è in grado di risolvere tale problema?

- \*\* **R1.17.** Immaginate di voler affidare al vostro fratello minore il compito di effettuare la copia di sicurezza (*backup*) del vostro lavoro. Scrivete un insieme di istruzioni dettagliate per svolgere tale compito, spiegando quanto spesso occorra farlo e quali file debbano essere copiati, indicando esplicitamente la cartella d'origine e quella di destinazione dell'azione di copiatura. Spiegate, infine, in che modo debba essere verificata la corretta esecuzione del backup.

- ★ **R1.18.** Scrivete lo pseudocodice di un algoritmo che descriva la preparazione della colazione domenicale a casa vostra.
- ★★ **R1.19.** Gli antichi Babilonesi conoscevano un algoritmo per calcolare la radice quadrata del numero  $a$ . Si parte con un valore iniziale,  $g = a/2$ , poi si calcola il valore medio tra  $g$  e  $a/g$ , che sarà il nuovo valore di  $g$ ; si ripete finché due valori consecutivi assunti da  $g$  differiscono di una quantità desiderata e a quel punto  $g$  sarà il valore cercato. Scrivete lo pseudocodice per questo algoritmo.

## Esercizi di programmazione

- ★ **E1.1.** Scrivete un programma che visualizzi un messaggio di saluto a vostra scelta, magari in una lingua diversa dall'inglese.
- ★★ **E1.2.** Scrivete un programma che calcoli e visualizzi la somma dei primi 10 numeri interi positivi:  $1 + 2 + \dots + 10$ .
- ★★ **E1.3.** Scrivete un programma che calcoli e visualizzi il prodotto dei primi 10 numeri interi positivi:  $1 \times 2 \times \dots \times 10$  (in Java, la moltiplicazione si indica con l'asterisco, \*).
- ★★ **E1.4.** Scrivete un programma che calcoli e visualizzi il saldo di un conto bancario dopo il primo, secondo e terzo anno. Il conto ha un saldo iniziale di \$1000 e vi vengono accreditati annualmente interessi pari al 5% del saldo.
- ★ **E1.5.** Scrivete un programma che visualizzi il vostro nome all'interno di un rettangolo. Fate quanto possibile per comporre i lati del rettangolo con i caratteri | - +, come nell'esempio seguente:

```
+---+
|Dave|
+---+
```

- ★★ **E1.6.** Scrivete un programma che scriva il vostro nome con lettere molto grandi, come nell'esempio seguente:

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

- ★★ **E1.7.** Scrivete un programma che visualizzi il vostro nome usando l'alfabeto Morse, come nell'esempio seguente, usando una diversa invocazione di `System.out.print` per ciascuna lettera:

```
.... - - - . - -
```

- ★★ **E1.8.** Scrivete un programma che, usando caratteri, visualizzi un viso simile a questo (ma diverso):

```
/////
+''''''+
(|_o_o_|)
| ^ |
| '-' |
+---+
```

- ★★ **E1.9.** Scrivete un programma che visualizzi un'imitazione di un quadro di Piet Mondrian (se non conoscete l'artista, fate una ricerca in Internet). Per rappresentare aree di colori diversi usate sequenze di caratteri diversi, come @@ oppure :::, usando i caratteri - e | per comporre, rispettivamente, linee orizzontali e verticali.
- ★★ **E1.10.** Scrivete un programma che visualizzi una casa, identica a questa:

```

+
+
+
+----+
| . .
| | |
+---+-

```

- ★★★ **E1.11.** Scrivete un programma che visualizzi un animale mentre pronuncia un saluto, simile a questo (ma diverso):

```

^_/
( ' ) / ---- \
( - ) < Hello \
| | | \ Junior |
( _ ) ----- /

```

- ★ **E1.12.** Scrivete un programma che, su tre righe consecutive, visualizzi tre stringhe, ad esempio i nomi dei vostri migliori amici o i vostri film preferiti.
- ★ **E1.13.** Scrivete un programma che visualizzi una poesia di vostro gradimento. Se non ne avete una preferita, cercate in Internet “Emily Dickinson” o “e e cummings”.
- ★★ **E1.14.** Scrivete un programma che visualizzi la bandiera degli Stati Uniti d’America usando soltanto i caratteri \* e =.
- ★★ **E1.15.** Copiate ed eseguite il programma seguente, poi modificate lo in modo che visualizzi il messaggio “Hello, *vostro nome!*”.

```

import javax.swing.JOptionPane;

public class DialogViewer
{
    public static void main(String[] args)
    {
        JOptionPane.showMessageDialog(null, "Hello, World!");
    }
}

```

- ★★ **E1.16.** Copiate ed eseguite il programma seguente, poi modificate lo in modo che visualizzi il messaggio “Hello, *nome!*”, essendo *nome* ciò che viene scritto dall’utente nella finestra di dialogo che compare durante l’esecuzione del programma.

```

import javax.swing.JOptionPane;

public class DialogViewer
{
    public static void main(String[] args)
    {
        String name = JOptionPane.showInputDialog("What is your name?");
        System.out.println(name);
    }
}

```

- \*\*\* **E1.17.** Modificate il programma dell'esercizio precedente in modo che il dialogo con l'utente prosegua con il messaggio "My name is Hal! What would you like me to do?" Dopotiché ignorate quanto scritto dall'utente e visualizzate in ogni caso il messaggio seguente (sostituendo Dave con il nome fornito inizialmente dall'utente):

I'm sorry, Dave. I'm afraid I can't do that.

- \*\* **E1.18.** Copiate ed eseguite il programma seguente, poi modificalo in modo che visualizzi un diverso saluto e una diversa immagine.

```
import java.net.URL;
import javax.swing.ImageIcon;
import javax.swing.JOptionPane;

public class Test
{
    public static void main(String[] args) throws Exception
    {
        URL imageLocation = new URL(
            "http://horstmann.com/java4everyone/duke.gif");
        JOptionPane.showMessageDialog(null, "Hello", "Title",
            JOptionPane.PLAIN_MESSAGE, new ImageIcon(imageLocation));
    }
}
```

- \* **E1.19 (economia).** Scrivete un programma che visualizzi l'elenco delle date dei compleanni dei vostri amici, su due colonne: nella prima colonna ci siano i nomi e nella seconda le date.
- \* **E1.20 (economia).** Negli Stati Uniti d'America non esistono tasse federali sugli acquisti, per cui ciascuno stato può imporre le proprie. Cercate in Internet le percentuali della tassazione sugli acquisti applicata da cinque stati, poi scrivere un programma che le visualizzi con questo formato:

Sales Tax Rates	
<hr/>	
Alaska:	0%
Hawaii:	4%
...	

- \* **E1.21 (economia).** Conoscere più di una lingua è un'abilità molto importante nell'attuale mercato del lavoro e una delle attività più elementari consiste nel porgere un saluto. Scrivete un programma che visualizzi una tabella con due colonne, riportando frasi di saluto: nella prima colonna la frase in inglese e nella seconda la stessa frase in un'altra lingua, a vostra scelta. Se non parlate altre lingue oltre all'inglese, usate un traduttore in Internet o chiedete a un amico.

#### Elenco di frasi da tradurre

Good morning.

It is a pleasure to meet you.

Please call me tomorrow.

Have a nice day!

**Sul sito web dedicato al libro si trova una raccolta di progetti di programmazione più complessi.**

# 2

## Utilizzare oggetti



### Obiettivi del capitolo

- Imparare a utilizzare variabili
- Capire i concetti di classe e di oggetto
- Saper invocare metodi
- Usare parametri e valori restituiti dai metodi
- Essere in grado di consultare la documentazione dell'API di Java
- Realizzare programmi di collaudo
- Capire la differenza tra oggetti e riferimenti a oggetti
- Scrivere programmi che visualizzano semplici forme grafiche

La maggior parte dei programmi utili non manipola soltanto numeri e stringhe, ma gestisce dati più complessi e più aderenti alla realtà, come conti bancari, informazioni sugli impiegati e forme grafiche.

Il linguaggio Java è perfettamente appropriato per progettare e per gestire dati di questo tipo, detti *oggetti*: in Java, per descrivere il comportamento di oggetti, si definiscono *classi*. In questo capitolo imparerete a manipolare oggetti appartenenti a classi già realizzate da altri: sarete così pronti per il capitolo successivo, nel quale imparerete a progettare vostre classi.

## 2.1 Oggetti e classi

Per scrivere un programma per calcolatore si mettono insieme un certo numero di “blocchi elementari” e anche in Java si fa così: si costruiscono programmi usando oggetti. Ogni oggetto ha uno specifico comportamento e può essere manipolato per raggiungere determinati obiettivi.

Come analogia, pensate a un muratore, che costruisce una casa a partire da alcuni elementi disponibili: porte, finestre, muri, tubi, una caldaia, uno scaldabagno, e così via. Ciascuno di questi elementi svolge una funzione specifica e tutti insieme collaborano per raggiungere un obiettivo prefissato. Notate, anche, come il muratore non debba preoccuparsi in alcun modo di come si costruisca una finestra o uno scaldabagno: questi elementi sono già disponibili e il lavoro del muratore consiste soltanto nella loro integrazione all'interno della casa.

Naturalmente i programmi per computer sono più astratti di una casa e gli oggetti che li costituiscono sono meno tangibili di una finestra o di uno scaldabagno, ma l'analogia è valida: un programmatore genera un programma funzionante a partire da elementi, gli oggetti, aventi la funzionalità richiesta. In questo capitolo acquisirete le informazioni basilari relative all'utilizzo di oggetti progettati da altri programmatore.

### 2.1.1 Utilizzare oggetti

Gli oggetti sono entità di un programma che si possono manipolare invocando metodi.

Un metodo è una sequenza di istruzioni che accedono ai dati di un oggetto.

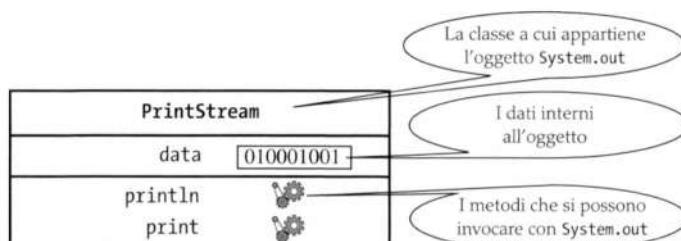
Un **oggetto** (*object*) è un'entità che si può manipolare mediante l'invocazione dei suoi **metodi** (*method*). Un metodo è costituito da una sequenza di istruzioni che possono accedere ai dati interni dell'oggetto: quando invocate (o, come anche si dice, “chiamate”) un metodo, non sapete con precisione quali siano le sue istruzioni, né sapete come sia organizzato internamente l'oggetto, ma il comportamento del metodo è ben definito e questo è ciò che ci interessa quando lo usiamo.

Per esempio, nel Capitolo 1 avete visto che `System.out` si riferisce a un oggetto, che può essere manipolato invocando il metodo `println`. Quando viene chiamato il metodo `println`, all'interno dell'oggetto si svolgono alcune attività, il cui effetto finale è che l'oggetto fa comparire il testo nella finestra di console. Non sapete come ciò avvenga, ma questo non è un problema: ciò che importa è che il metodo porti a termine il lavoro che avete richiesto.

La Figura 1 mostra una rappresentazione schematica dell'oggetto `System.out`: i dati interni sono rappresentati da una sequenza di valori binari (zero e uno), mentre ciascun metodo è raffigurato da un insieme di ingranaggi, che simboleggiano una parte strumentale che porta a termine il compito a essa assegnato.

**Figura 1**

Rappresentazione dell'oggetto `System.out`



In generale, potete pensare a un oggetto come a un'entità che è in grado di compiere delle azioni su vostro ordine quando ne invocate i metodi: come venga svolto il lavoro richiesto non è importante per il programmatore che usa l'oggetto.

Nella parte restante di questo capitolo vedrete altri oggetti e i metodi che li caratterizzano.

## 2.1.2 Classi

Nel Capitolo 1 avete visto due oggetti:

- `System.out`
- "Hello, World!"

Una classe descrive un insieme  
di oggetti aventi  
lo stesso comportamento.

Ciascuno di questi oggetti appartiene a una diversa **classe** (*class*): l'oggetto `System.out` appartiene alla classe `PrintStream`, mentre l'oggetto "Hello, World!" appartiene alla classe `String`. Una classe specifica i metodi che possono essere applicati ai suoi oggetti. Ovviamente esistono molti altri oggetti di tipo `String`, come "Goodbye" o "Mississippi", e tutti hanno qualcosa in comune: con tutte le stringhe potete invocare gli stessi metodi, alcuni dei quali verranno presentati nel Paragrafo 2.3.

Come vedrete nel Capitolo 11, si possono costruire oggetti della classe `PrintStream` diversi da `System.out`. Si tratta di oggetti che scrivono dati in file o in altre destinazioni, diverse dalla finestra di console, ma, comunque, tutti gli oggetti di `PrintStream` condividono lo stesso comportamento: con ciascuno di essi è possibile invocare i metodi `print` e `println`, ottenendo come effetto l'invio alla destinazione prescelta dei valori da visualizzare.

Come è ovvio che sia, gli oggetti della classe `PrintStream` hanno un comportamento completamente diverso da quello degli oggetti della classe `String`: non si può invocare il metodo `println` con una stringa, perché questa non saprebbe come inviare se stessa a una finestra di console o a un file.

Quindi, appare evidente come classi diverse si assumano responsabilità differenti: una stringa conosce le lettere che la costituiscono, ma non sa come visualizzarle in modo che un utente umano le possa leggere, né è in grado di memorizzarle in un file.



### Auto-valutazione

1. In Java, gli oggetti sono raggruppati in classi sulla base del loro comportamento. Una finestra e uno scaldabagno apparterrebbero alla medesima classe o a classi diverse? Perché?
2. Alcune lampadine funzionano mediante un filamento luminoso, altre usano un gas fluorescente. Se considerate una lampadina come un oggetto Java dotato del metodo "illumina", avreste bisogno di sapere di quale tipo di lampadina si tratta?
3. Cosa succede quando si tenta di invocare il metodo seguente?

```
"Hello, World".println(System.out)
```

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R2.1 e R2.2, al termine del capitolo.

## 2.2 Variabili

Prima di proseguire con l'argomento principale di questo capitolo, che è il comportamento degli oggetti, dobbiamo fare una panoramica presentando un po' di terminologia di base relativa alla programmazione. Nei paragrafi che seguono imparerete i concetti di variabile, di tipo e di assegnazione.

### 2.2.1 Dichiarazioni di variabili

Quando un programma manipola oggetti, solitamente c'è bisogno di memorizzare gli oggetti stessi e i valori restituiti dai loro metodi, in modo da poterli utilizzare anche in seguito. Per memorizzare valori in un programma Java si usano variabili e l'enunciato seguente dichiara una variabile di nome `width`:

```
int width = 20;
```

Una variabile è un zona di memoria dotata di un nome.

Quando si dichiara una variabile di solito se ne specifica anche un valore iniziale.

Una **variabile** (*variable*) è una zona di memoria usata all'interno di un programma per computer: ciascuna variabile ha un nome e contiene un valore.

Una variabile è simile a uno stallone in un parcheggio: ha un nome che lo identifica (detto "identificatore", come "J 053") e può contenere un veicolo. Una variabile ha un nome (come `width`) e può contenere un valore (come 20).

Quando si dichiara una variabile solitamente se ne specifica anche un valore iniziale, cioè la si **inizializza** con un valore che, appunto, viene memorizzato nella variabile non appena questa viene creata. Riprendiamo in esame questa dichiarazione di variabile:

```
int width = 20;
```

## Sintassi di Java

### 2.1 Dichiarazione di variabile

#### Sintassi

*nomeTipo nomeVariabile = valore;*

oppure

*nomeTipo nomeVariabile;*

#### Esempio

Il tipo specifica ciò che si può fare con il valore memorizzato nella variabile.

String greeting = "Hello, Dave!";

La dichiarazione di una variabile termina con un punto e virgola.

Usate un nome di variabile significativo.

Fornire un valore iniziale è facoltativo, ma solitamente è una buona idea.

La variabile `width` viene inizializzata con il valore 20.

Così come lo stallone di un parcheggio è spesso riservato a un certo tipo di veicoli (come i veicoli elettrici o i motocicli), allo stesso modo una variabile, in Java, memorizza dati di un tipo specifico. Java fornisce supporto a parecchi tipi di dati: numeri, stringhe di testo, file, date e molti altri. Ogni volta che si dichiara una variabile bisogna specificarne il tipo (come evidenziato nella sezione Sintassi di Java 2.1).

**Quando si dichiara una variabile si specifica anche il tipo dei suoi valori.**

La variabile `width` ha valori di tipo *numero intero*, cioè numeri senza parte frazionaria. In Java questo tipo di dato si chiama `int` e, come si può notare, il tipo va specificato prima del nome della variabile:

```
int width = 20;
```

Dopo aver dichiarato e inizializzato una variabile, la si può utilizzare. Ad esempio:

```
int width = 20;
System.out.println(width);
int area = width * width;
```

La Tabella 1 mostra alcuni esempi di dichiarazione di variabile.

**Tabella 1**

Dichiarazioni di variabili in Java

Dichiarazione di variabile	Commento
<code>int width = 20;</code>	Dichiara una variabile intera e le assegna il valore iniziale 20.
<code>int perimeter = 4 * width;</code>	Il valore iniziale può dipendere da altre variabili (che, ovviamente, devono essere state dichiarate in precedenza).
<code>String greeting = "Hi!";</code>	Questa variabile è di tipo <code>String</code> e viene inizializzata con la stringa "Hi".
 <code>height = 30;</code>	<b>Errore:</b> manca il tipo. Questo enunciato non è una dichiarazione di variabile ma un assegnamento di un nuovo valore a una variabile esistente (Paragrafo 2.2.5).
 <code>int height = "20";</code>	<b>Errore:</b> non si può usare una stringa per inizializzare una variabile di tipo numerico (ci sono le virgolette).
<code>int width;</code>	Dichiara una variabile di tipo "numero intero" senza inizializzarla, un'azione che può essere fonte di errori (sezione Errori comuni 2.1).
<code>int width, height;</code>	Dichiara due variabili intere in un solo enunciato. In questo libro dichiareremo ogni variabile in un enunciato a sé stante.

## 2.2.2 Tipi

Il tipo `int` si usa per numeri che non possono avere una parte frazionaria.

In Java esistono diversi tipi di numeri. Si usa il tipo `int` per numeri interi, cioè privi di parte frazionaria. Supponiamo, ad esempio, di dover contare il numero di automobili presenti in un parcheggio: il conteggio è certamente un numero intero, perché non ci può essere una parte di automobile.

Il tipo `double` si usa per i numeri in virgola mobile.

Quando serve una parte frazionaria (come nel numero 22.5) usiamo i **numeri in virgola mobile** (*floating-point number*). In Java, `double` è il tipo di dato più utilizzato per i numeri in virgola mobile e questo è un esempio di dichiarazione di variabile:

```
double milesPerGallon = 22.5;
```

I numeri si possono combinare con operatori aritmetici, come +, - e \*.

I numeri possono essere combinati usando gli operatori + e -, come in `width + 10` oppure `width - 1`. Per moltiplicare due numeri si usa l'operatore \*: ad esempio,  $2 \times width$  si scrive `2 * width`. Per le divisioni si usa l'operatore /, come in `width / 2`.

Come in matematica, gli operatori \* e / hanno la precedenza rispetto agli operatori + e -, per cui `width + height * 2` esprime la somma tra `width` e il prodotto `height * 2`. Volendo moltiplicare la somma per due, bisogna usare le parentesi (tonde): `(width + height) * 2`.

Non tutti i dati sono numerici. Ad esempio, il valore "Hello" è di tipo `String` e quando si definisce una variabile che contiene una stringa bisogna usare tale tipo di dato:

```
String greeting = "Hello";
```

Un tipo di dato specifica le operazioni che si possono compiere con valori di quel tipo.

I tipi sono importanti perché indicano ciò che si vuol fare con una determinata variabile. Ad esempio, consideriamo la variabile `width`. È di tipo `int`, quindi si può moltiplicare il valore contenuto in essa per un altro numero. Il tipo di `greeting`, invece, è `String` e una stringa non può essere moltiplicata per un numero (nel Paragrafo 2.3.1 vedrete cosa si può fare con le stringhe).

### 2.2.3 Nomi

Quando si dichiara una variabile bisogna scegliere un nome che ne illustri lo scopo. Ad esempio, è preferibile usare un nome descrittivo, come `milesPerGallon`, piuttosto che un nome criptico, magari un acronimo come `mpg`.

In Java occorre seguire alcune semplici regole per i nomi di variabili, di metodi e di classi:

- I nomi devono iniziare con una lettera o con un segno di sottolineatura (\_), mentre i caratteri che seguono possono essere lettere, cifre numeriche o caratteri di sottolineatura (tecnicamente è ammesso anche il simbolo \$, ma non dovreste usarlo, perché il suo utilizzo è previsto per i nomi che vengono generati automaticamente da strumenti di ausilio alla programmazione).
- Non si possono usare altri simboli, come ? o %, né spazi. Le lettere maiuscole possono essere utilizzate per segnalare i confini tra le parole, come in `milesPerGallon` (questa convenzione viene chiamata ironicamente "a cammello", perché le lettere maiuscole svettano come le sue gobbe).
- Nei nomi le lettere maiuscole sono distinte dalle lettere minuscole, per cui `milesPerGallon` e `milespergallon` sono nomi diversi.
- Infine, le **parole riservate**, come `double` o `class`, non possono essere usate come nomi; tali parole sono riservate in modo esclusivo allo speciale significato che hanno nel linguaggio Java (l'Appendice C riporta l'elenco di tutte le parole riservate in Java).

Per convenzione, i nomi delle variabili devono iniziare con una lettera minuscola.

Poi, è convenzione assai diffusa tra i programmati Java che i nomi di variabili e metodi inizino con una lettera minuscola (come `milesPerGallon`) e che i nomi di classi inizino con una lettera maiuscola (come `HelloPrinter`): in questo modo è facile distinguerli.

La Tabella 2 presenta esempi di nomi di variabile leciti e invalidi in Java.

**Tabella 2**

Nomi di variabili in Java

Nome di variabile	Commento
<code>distance_1</code>	I nomi sono costituiti da lettere, numeri e caratteri di sottolineatura.
<code>x</code>	In matematica si usano nomi di variabili brevi, come <code>x</code> o <code>y</code> . Anche in Java è lecito fare così, ma è una pratica poco comune, perché rende poco comprensibili i programmi (sezione Suggerimenti per la programmazione 2.1).
 <code>CanVolume</code>	<b>Attenzione:</b> nei nomi, le lettere maiuscole e minuscole sono diverse. Questo nome di variabile è diverso da <code>canVolume</code> e viola la convenzione che prevede che i nomi delle variabili inizino con una lettera minuscola.
 <code>6pack</code>	<b>Errore:</b> un nome non può iniziare con una cifra.
 <code>can volume</code>	<b>Errore:</b> un nome non può contenere spazi.
 <code>doubles</code>	<b>Errore:</b> non si può usare una parola riservata del linguaggio come nome.
 <code>miles/gal</code>	<b>Errore:</b> nei nomi non si possono usare simboli come <code>/</code> .

## 2.2.4 Commenti

Quando i programmi iniziano a diventare complessi e articolati, è bene aggiungere **commenti**, cioè spiegazioni del codice destinate a lettori umani. Ecco, ad esempio, un commento che motiva il valore usato per inizializzare una variabile:

```
double milesPerGallon = 35.5; // Consumo medio delle auto statunitensi nel 2013
```

Questo commento spiega a un lettore umano il significato del valore 35.5. Il compilatore non elabora i commenti in alcun modo: ignora qualsiasi cosa scritta dall'inizio del commento, che è rappresentato dalla coppia di caratteri consecutivi `//`, fino alla fine della riga.

Scrivere commenti è una buona pratica di programmazione, perché aiuta i programmati che leggeranno il vostro codice a comprendere i vostri obiettivi; inoltre, troverete utili i commenti anche per voi stessi, quando revisionerete i vostri programmi dopo qualche tempo.

Per commenti brevi si usa la sintassi `//`, mentre i commenti più lunghi vengono solitamente delimitati da `/*` e `*/`: di nuovo, il compilatore ignora tali delimitatori e tutto ciò che contengono. Ad esempio:

```
/*
```

Nella maggior parte dei Paesi l'efficienza delle auto è misurata in litri per centinaia di chilometri percorsi e in effetti è forse più utile: è una misura

I commenti si usano per fornire spiegazioni ai lettori umani in merito al codice. Il compilatore ignora i commenti.

che dice quanto carburante occorre acquistare per percorrere una determinata distanza. Ecco la formula di conversione.

```
/*
double fuelEfficiency = 235.214583 / milesPerGallon;
```

## 2.2.5 Assegnazioni

Per modificare il valore di una variabile si usa l'operatore di assegnazione (=).

Usando l'operatore di assegnazione o assegnamento (=) è possibile modificare il valore di una variabile. Ad esempio, considerate la seguente dichiarazione di variabile:

```
int width = 10; ①
```

Se volete modificare il valore della variabile, assegnatele semplicemente il nuovo valore, in questo modo:

```
width = 20; ②
```

L'assegnazione provoca la sostituzione del valore originario della variabile, come si può vedere nella Figura 2.

**Figura 2**

Assegnazione di un nuovo valore a una variabile

① width =   
 ② width =

Usare una variabile a cui non sia mai stato assegnato un valore è un errore. Ad esempio, la sequenza di enunciati

```
int height;
int width = height; // ERRORE: la variabile height è priva di valore iniziale
```

## Sintassi di Java

## 2.2 Assegnazione

### Sintassi

*nomeVariabile* = *valore*;

### Esempio

Questo è una dichiarazione di variabile.

Il valore di questa variabile viene modificato.

double width = 20;

...

width = 30;

...

width = width + 10;

Questo è un enunciato di assegnazione.

Il nuovo valore della variabile.

Lo stesso nome può figurare a sinistra e a destra dell'assegnazione (Figura 4).

contiene un errore: quando si utilizza una variabile a cui non è mai stato assegnato un valore, il compilatore protesta per “variabile non inizializzata” (*uninitialized variable*), una situazione evidenziata nella Figura 3.

**Figura 3**

Una variabile priva di valore iniziale

Non è stato assegnato alcun valore  
height =

Tutte le variabili devono essere inizializzate prima di essere utilizzate.

L'operatore di assegnazione non esprime un'uguaglianza matematica.

La soluzione consiste nell'assegnare un valore alla variabile prima di utilizzarla:

```
int height = 20
int width = height; // così è corretto
```

A destra del simbolo `=` ci può anche essere un'espressione matematica, come in questo caso:

```
width = height + 10;
```

Ciò significa: “calcola il valore di `height + 10` e memorizzalo nella variabile `width`”.

Nel linguaggio di programmazione Java, l'operatore `=` indica un'*azione* atta a sostituire il valore memorizzato in una variabile, un significato assai diverso da quello del simbolo matematico `=`, che enuncia un'uguaglianza. Ad esempio, in Java l'enunciato seguente è perfettamente lecito:

```
width = width + 10;
```

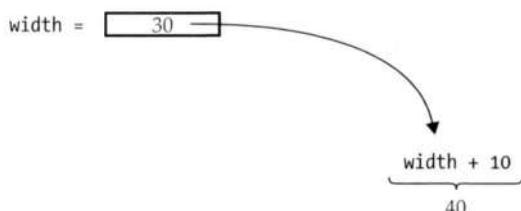
Ciò significa: “calcola il valore di `width + 10` e memorizzalo nella variabile `width`” (come rappresentato nella Figura 4).

In Java, quindi, il fatto che la variabile `width` compaia sia a sinistra sia a destra del simbolo `=` non costituisce un problema, mentre, ovviamente, l'equazione matematica `width = width + 10` non ha soluzione.

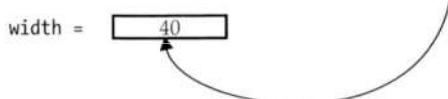
**Figura 4**

Esecuzione dell'enunciato  
`width = width + 10`

① Calcola il valore della parte destra



② Memorizza il valore nella variabile





## Auto-valutazione

4. Cosa c'è di sbagliato nella seguente dichiarazione di variabile?  

```
int miles per gallon = 39.4
```
5. Dichiarate e inizializzate due variabili, `unitPrice` e `quantity`, per memorizzare, rispettivamente, il prezzo unitario di un articolo e il numero di articoli dello stesso tipo che compongono un acquisto. Usate valori iniziali ragionevoli.
6. Usate le variabili dichiarate nella domanda precedente per visualizzare il costo totale dell'acquisto.
7. Di che tipo sono i valori 0 e "0"?
8. Che tipo di numero usereste per memorizzare l'area di un cerchio?
9. Quali dei seguenti identificatori sono validi?

```
Greeting1
g
void
101dalmatians
Hello, World
<greeting>
```

10. Dichiarate una variabile adatta a memorizzare il vostro nome, usando un identificatore "a forma di cammello".
11. L'espressione `12 = 12` è valida nel linguaggio Java?
12. Come si modifica il valore della variabile `greeting` in modo che diventi "`Hello, Nina!`"?
13. Come spieghereste l'assegnazione usando l'analogia degli stalli nei parcheggi?

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi R2.4, R2.5 e R2.7, al termine del capitolo.



## Errori comuni 2.1

### Usare variabili non dichiarate o non inizializzate

Prima di usare una variabile per la prima volta è necessario dichiararla. Ad esempio, la sequenza di enunciati seguente non è valida:

```
int perimeter = 4 * width; // ERRORE: width non è stata dichiarata
int width = 20;
```

Nei programmi gli enunciati vengono compilati seguendo l'ordine in cui sono scritti. Quando il compilatore raggiunge il primo dei due enunciati, non sa che la variabile `width` verrà dichiarata nella riga successiva, quindi segnala un errore. La soluzione consiste nello scambio delle dichiarazioni, in modo che ciascuna variabile venga dichiarata prima di essere utilizzata.

Un errore simile è l'utilizzo di una variabile rimasta non inizializzata:

```
int width;
int perimeter = 4 * width; // ERRORE: width non è stata inizializzata
```

Il compilatore Java segnalerà l'utilizzo di una variabile (`width`) a cui non è ancora stato assegnato alcun valore. La soluzione consiste nell'assegnare un valore alla variabile prima di utilizzarla.



## Errori comuni 2.2

---

### Fare confusione tra enunciati di dichiarazione di variabile e di assegnazione

Supponiamo di aver dichiarato una variabile, in questo modo:

```
int width = 20;
```

Per modificare il valore della variabile, usiamo un enunciato di assegnazione:

```
width = 30;
```

Usare, invece, un'altra dichiarazione di variabile è un errore assai frequente:

```
int width = 30; // ERRORE, inizia con int e, quindi, è una dichiarazione
```

Ma esiste già una variabile di nome `width`, per cui il compilatore protesterà, segnalando che state cercando di dichiarare una nuova variabile avente lo stesso nome.



## Suggerimenti per la programmazione 2.1

---

### Scegliete nomi significativi per le variabili

In algebra, i nomi di variabili sono solitamente di una sola lettera, come `p` o `A`, eventualmente con un pedice, come `p1`. Potreste, per analogia, cedere alla tentazione di risparmiare il numero di caratteri da digitare sulla tastiera, usando nei vostri programmi Java nomi corti per le variabili:

```
int a = w * h;
```

Confrontate, però, l'enunciato precedente con questo:

```
int area = width * height;
```

Il vantaggio è assolutamente evidente: leggendo `width`, invece di `w`, è molto più facile capire che si tratta di una larghezza (`width`, in inglese).

Nella programmazione reale, quando i programmi sono scritti da più persone, l'uso di nomi significativi e descrittivi per le variabili assume particolare importanza. Per voi può essere ovvio che `w` sta per "larghezza", ma sarà altrettanto ovvio per chi dovrà aggiornare il vostro codice fra qualche anno? E voi stessi, tra un mese, vi ricorderete del significato di quella variabile `w`?

## 2.3 Invocare metodi

Un programma svolge il proprio lavoro invocando metodi con i propri oggetti. In questo paragrafo vedremo come fornire valori a un metodo e come acquisire il risultato prodotto dall'esecuzione di un metodo.

### 2.3.1 L'interfaccia pubblica di una classe

Un oggetto si usa invocandone i metodi, che sono condivisi da tutti gli oggetti della stessa classe. Ad esempio, la classe `PrintStream` mette a disposizione metodi per i propri oggetti (come `println` e `print`). Analogamente, la classe `String` mette a disposizione metodi che si possono applicare a oggetti di tipo `String`. Uno di essi è il metodo `length`, che conta il numero di caratteri presenti in una stringa e può essere applicato a qualsiasi oggetto di tipo `String`. Ad esempio, la sequenza di enunciati

```
String greeting = "Hello, World!";
int numberOfCharacters = greeting.length();
```

assegna a `numberOfCharacters` il numero di caratteri presenti nell'oggetto `"Hello, World!"` di tipo `String`. Dopo l'esecuzione delle istruzioni del metodo `length`, a `numberOfCharacters` viene assegnato il valore 13 (le virgolette non fanno parte della stringa e, quindi, il metodo `length` non le conta).

Il metodo `length`, diversamente dal metodo `println`, non richiede dati da elaborare (dati “in ingresso”) all’interno delle parentesi tonde, però fornisce un valore “in uscita”, il conteggio dei caratteri. Va osservato che la generazione di tale valore di conteggio non produce alcun effetto visibile durante l'esecuzione del programma: viene semplicemente reso disponibile per successive elaborazioni all'interno del programma stesso.

Esaminiamo ora un altro metodo della classe `String`. Quando a un oggetto di tipo `String` viene applicato il metodo `toUpperCase`, questo crea un nuovo oggetto di tipo `String` che contiene gli stessi caratteri dell'oggetto originale, con le lettere minuscole convertite in maiuscole. Ad esempio, la sequenza di enunciati

```
String river = "Mississippi";
String bigRiver = river.toUpperCase();
```

assegna a `bigRiver` l'oggetto `"MISSISSIPPI"`, di tipo `String`.

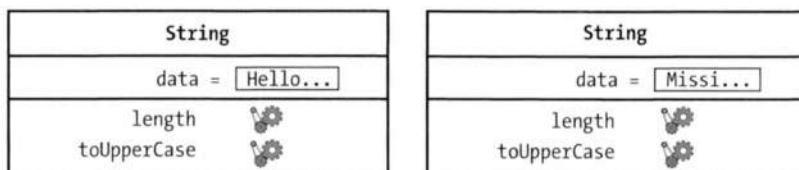
La classe `String` definisce molti altri metodi oltre a `length` e `toUpperCase`, alcuni dei quali saranno presentati nel Capitolo 4. Tutti insieme, i metodi di una classe costituiscono la sua **interfaccia pubblica** (*public interface*) determinando ciò che può essere fatto con gli oggetti della classe. Una classe definisce anche una **realizzazione** (o implementazione) **privata**, che descrive i dati interni ai propri oggetti e le istruzioni per l'esecuzione dei propri metodi: tali dettagli non sono noti ai programmatore che usano oggetti, invocandone metodi.

La Figura 5 mostra due oggetti della classe `String`: ciascun oggetto memorizza i propri dati (rappresentati come rettangoli che contengono caratteri) ed entrambi gli oggetti forniscono supporto al medesimo insieme di metodi, che costituiscono l'interfaccia pubblica specificata dalla classe `String`.

L'interfaccia pubblica di una classe  
specificà cosa si può fare  
con i suoi oggetti, mentre  
l'implementazione nasconde  
descrive come si portano  
a termine tali azioni.

**Figura 5**

Rappresentazione di due oggetti di tipo String



### 2.3.2 Parametri dei metodi

Un argomento o parametro è un dato fornito durante l'invocazione di un metodo.

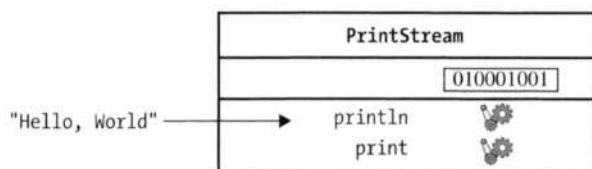
La maggior parte dei metodi necessita di valori in ingresso che specificino il tipo di elaborazione che deve essere svolta. Ad esempio, quando si invoca il metodo `println` si fornisce la stringa che deve essere visualizzata. In informatica, i dati in ingresso a un metodo vengono tecnicamente chiamati **parametri** o **argomenti**: diciamo, quindi, che nell'invocazione seguente la stringa `greeting` è un parametro

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

La Figura 6 illustra il trasferimento del parametro al metodo.

**Figura 6**

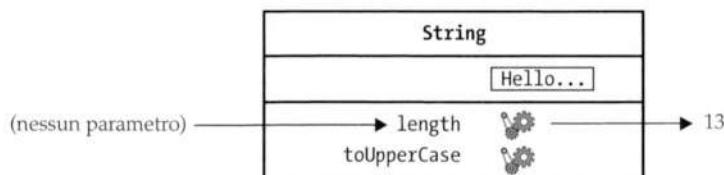
Passaggio di un parametro al metodo `println`



Alcuni metodi hanno bisogno di più parametri, mentre altri metodi non ne hanno affatto. Un esempio di questi ultimi è il metodo `length` della classe `String`, come si può vedere nella Figura 7: tutte le informazioni necessarie al metodo `length` per lo svolgimento del proprio compito, cioè per contare i caratteri presenti nella stringa, sono memorizzate nell'oggetto con cui viene invocato.

**Figura 7**

Invocazione del metodo `length` con un oggetto di tipo String



### 2.3.3 Valori restituiti

Alcuni metodi, come `println`, svolgono un'azione, mentre altri calcolano e restituiscono un valore. Ad esempio, il metodo `length` *restituisce un valore*: il numero di caratteri contenuti nella stringa. Tale valore restituito può essere memorizzato in una variabile:

```
int numberOfCharacters = greeting.length();
```

Il valore restituito da un metodo è un risultato calcolato dal metodo stesso.

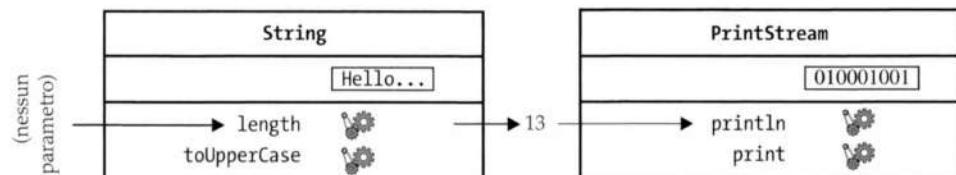
oppure utilizzato direttamente come parametro di un altro metodo:

```
System.out.println(greeting.length());
```

L'invocazione `greeting.length()` restituisce un valore, il numero intero 13, che diventa parametro del metodo `println`, come evidenziato nella Figura 8.

**Figura 8**

Il valore restituito da un metodo può essere utilizzato come parametro per un altro metodo



Non tutti i metodi restituiscono valori. Un esempio è il metodo `println`, che interagisce con il sistema operativo per visualizzare caratteri in una finestra, ma non restituisce alcun valore al metodo che l'ha invocato.

Analizziamo ora un'invocazione di metodo un po' più complessa: invochiamo il metodo `replace` della classe `String`. Il metodo `replace` esegue operazioni di ricerca e sostituzione, analogamente a quanto avviene in un *word processor*. Ad esempio, l'invocazione

```
river.replace("issipp", "our")
```

costruisce una nuova stringa, ottenuta sostituendo con "our" tutte le occorrenze di "issipp" in "Mississippi" (in questo caso avviene una sola sostituzione). Il metodo restituisce quindi l'oggetto "Missouri", di tipo `String`, che può essere memorizzato in una variabile

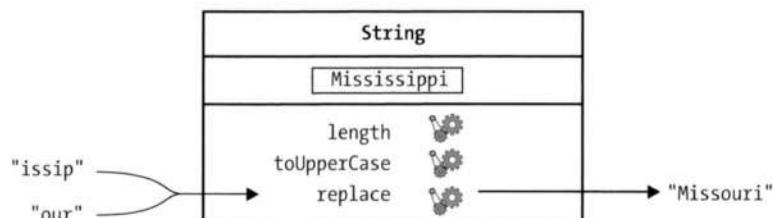
```
river = river.replace("issipp", "our");
```

oppure passato come parametro a un altro metodo:

```
System.out.println(river.replace("issipp", "our"));
```

**Figura 9**

Invocazione del metodo `replace`



Come si vede nella Figura 9, il metodo, in seguito a questa invocazione

- agisce sulla stringa "Mississippi" contenuta nella variabile `river`
- riceve due parametri: le stringhe "issipp" e "our"
- restituisce un valore: la stringa "Missouri"

**Tabella 3**  
Parametri di metodi e valori restituiti

Esempio	Commento
<code>System.out.println(greeting)</code>	<code>greeting</code> è il parametro del metodo <code>println</code> .
<code>greeting.replace("e", "3")</code>	Il metodo <code>replace</code> ha due parametri, in questo caso "e" e "3".
<code>greeting.length()</code>	Il metodo <code>length</code> non ha parametri.
<code>int n = greeting.length();</code>	Il metodo <code>length</code> restituisce un numero intero.
<code>System.out.println(n);</code>	Il metodo <code>println</code> non restituisce alcun valore.
<code>System.out.println(greeting.length());</code>	Il valore restituito da un metodo può diventare il parametro di un altro metodo.

### 2.3.4 Dichiarazioni di metodi

Quando in una classe si dichiara un metodo, vengono specificati i tipi dei parametri e del valore restituito. Ad esempio, la classe `String` dichiara il metodo `length` in questo modo:

```
public int length()
```

Come effetto di questa dichiarazione, il metodo non ha parametri e restituisce un valore di tipo `int` (per il momento considereremo "pubblici" tutti i metodi, mentre nel Capitolo 9 vedremo metodi con accesso più ristretto).

Il metodo `replace` è, invece, dichiarato in questo modo:

```
public String replace(String target, String replacement)
```

Per invocare il metodo `replace` bisogna, quindi, fornire due parametri, `target` e `replacement`, entrambi di tipo `String`; il valore restituito è anch'esso una stringa.

Quando un metodo non restituisce alcun valore, il tipo del valore restituito viene dichiarato mediante la parola chiave `void`. Ad esempio, la classe `PrintStream` definisce il metodo `println` in questo modo:

```
public void println(String output)
```

A volte una classe dichiara due metodi aventi lo stesso nome, con parametri di tipi diversi. Ad esempio, la classe `PrintStream` dichiara un secondo metodo, sempre di nome `println`, in questo modo:

```
public void println(int output)
```

Tale metodo viene usato per visualizzare un numero intero. Diciamo, in questo caso, che il nome `println` è **sovraffunzione** (nel senso di "carico di più significati", in inglese *overloaded*), dal momento che si riferisce a più metodi.



#### Auto-valutazione

14. In che modo si può calcolare la lunghezza della stringa "Mississippi"?
15. In che modo si può visualizzare la versione maiuscola della stringa "Hello, World!"?
16. Si può invocare `river.println()`? Perché?

17. Quali sono i parametri usati nell'invocazione `river.replace("p", "s")`?
18. Qual è il risultato dell'invocazione `river.replace("p", "s")`?
19. Qual è il risultato dell'invocazione `greeting.replace("World", "Dave").length()`?
20. Come viene dichiarato il metodo `toUpperCase` nella classe `String`?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R2.8, R2.9 e R2.10, al termine del capitolo.



## Suggerimenti per la programmazione 2.2

### Imparate facendo esperimenti

Quando imparate un metodo nuovo, scrivete un breve programma e provatelo. Ad esempio, questo è il momento giusto per aprire l'ambiente di sviluppo Java ed eseguire questo programma:

```
public class ReplaceDemo
{
    public static void main(String[] args)
    {
        String river = "Mississippi";
        System.out.println(river.replace("issipp", "our"));
    }
}
```

In questo modo potete vedere con i vostri occhi l'attività svolta dal metodo `replace`, dopodiché potete proseguire facendo altri esperimenti: il metodo `replace` sostituisce ogni occorrenza del primo parametro, o soltanto la prima? Provate:

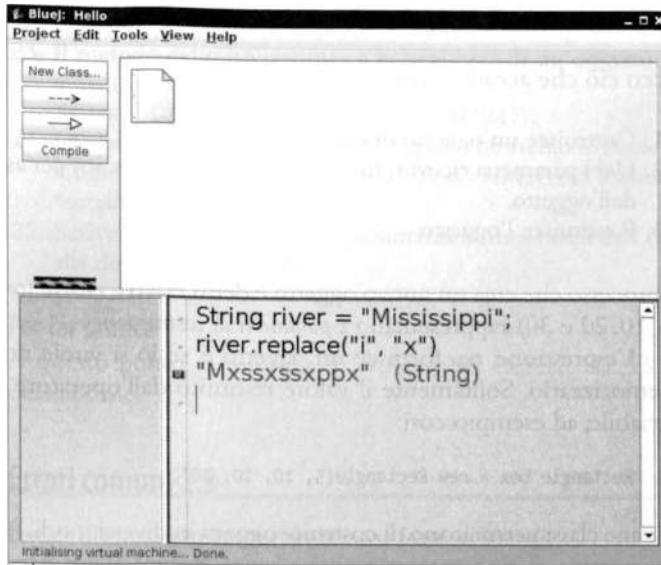
```
System.out.println(river.replace("i", "x"));
```

Predisponete il vostro ambiente di lavoro in modo che questo tipo di esperimenti possano essere condotti in modo semplice e naturale. Ad esempio, tenete a portata di mano lo schema vuoto del più semplice programma Java, in modo da poter generare un nuovo esempio facendo semplicemente un “copia e incolla”. In alternativa, sfruttate le potenzialità del vostro ambiente di sviluppo integrato, che molto probabilmente sarà in grado di “aggiungere” in modo automatico la definizione della classe e del metodo `main`, dopo che ne avrete scritto soltanto gli enunciati: scoprite se anche il vostro è in grado di farlo. Alcuni ambienti, come si può vedere nella Figura 10, sono addirittura in grado di comprendere semplici enunciati Java scritti in una finestra, visualizzandone gli effetti in un'altra, senza neanche dover scrivere un metodo `main` e senza dover invocare esplicitamente il metodo `System.out.println`.

## 2.4 Costruire oggetti

In generale, quando in un vostro programma volete usare un oggetto, dovete specificarne le proprietà iniziali *costruendolo*.

**Figura 10**  
Il pannello del codice nell'ambiente BlueJ



Per apprendere come si costruiscano oggetti, non possiamo limitarci agli oggetti di tipo `String` e all'oggetto `System.out`, per cui studiamo un'altra classe, la classe `Rectangle` della libreria di Java. Gli oggetti di tipo `Rectangle` descrivono forme rettangolari, che possono essere utili in varie situazioni: potete usare rettangoli per comporre un diagramma a barre, oppure potete realizzare semplici giochi che spostino rettangoli all'interno di una finestra.

**Figura 11**  
Oggetti di tipo `Rectangle`

Rectangle	Rectangle	Rectangle
x = 5	x = 35	x = 45
y = 10	y = 30	y = 0
width = 20	width = 20	width = 30
height = 30	height = 20	height = 20

Noteate che un oggetto di tipo `Rectangle` non è una forma rettangolare, ma un oggetto che contiene un insieme di numeri che *descrivono* il rettangolo (Figura 11): ciascun rettangolo è descritto dalle coordinate `x` e `y` del suo vertice superiore sinistro, dalla sua larghezza (`width`) e dalla sua altezza (`height`).

Capire questa distinzione è davvero molto importante. All'interno di un computer, un oggetto di tipo `Rectangle` è una zona di memoria che contiene quattro numeri, ad esempio `x = 5`, `y = 10`, `width = 20` e `height = 30`. Nella mente del programmatore che usa tale oggetto, invece, esso descrive una figura geometrica.

Per creare un nuovo rettangolo occorre specificarne i valori `x`, `y`, `width` e `height`, poi *invocare l'operatore new*, indicando il nome della classe e i parametri che sono necessari per costruire un nuovo oggetto di quel tipo. Per esempio, in questo modo si può costruire un rettangolo con le coordinate del vertice superiore sinistro uguali a (5, 10), con larghezza 20 e altezza 30:

Per costruire nuovi oggetti  
si usa l'operatore `new`,  
seguito dal nome di una classe  
e da parametri opportuni.

```
new Rectangle(5, 10, 20, 30)
```

Ecco ciò che accade, in dettaglio. L'operatore new:

1. Costruisce un oggetto di tipo Rectangle.
2. Usa i parametri ricevuti (in questo caso, 5, 10, 20 e 30) per assegnare valori iniziali ai dati dell'oggetto.
3. Restituisce l'oggetto.

Il processo che crea un nuovo oggetto è detto **costruzione** (*construction*) e i quattro valori (5, 10, 20 e 30) rappresentano i *parametri di costruzione*.

L'espressione new fornisce un oggetto e, se lo si vuole utilizzare in seguito, bisogna memorizzarlo. Solitamente il valore restituito dall'operatore new viene assegnato a una variabile, ad esempio così:

```
Rectangle box = new Rectangle(5, 10, 20, 30);
```

Alcune classi permettono di costruire oggetti in diversi modi. Per esempio, potete ottenere un oggetto di tipo Rectangle anche senza fornire alcun parametro di costruzione (ma dovete sempre inserire le parentesi):

```
new Rectangle()
```

Questa espressione costruisce un rettangolo (piuttosto inutile) avente larghezza 0, altezza 0 e il vertice superiore sinistro posizionato nell'origine (0, 0) del sistema di coordinate.



### Auto-valutazione

21. Come si costruisce un quadrato con lato di lunghezza 20 e centrato nel punto di coordinate (100, 100)?

## Sintassi di Java

### 2.3 Costruzione di oggetti

#### Sintassi

```
new NomeClasse(parametri)
```

#### Esempio

L'espressione new restituisce un oggetto.

#### Parametri di costruzione

```
Rectangle box = new Rectangle(5, 10, 20, 30);
```

Di solito l'oggetto costruito viene memorizzato in una variabile.

```
System.out.println(new Rectangle());
```

L'oggetto costruito può anche essere passato come parametro a un metodo.

Occorrono le parentesi anche se non vi sono parametri.

22. Inizializzate le variabili `box` e `box2` con due rettangoli che si tocchino.
23. Il metodo `getWidth` restituisce la larghezza di un oggetto di tipo `Rectangle`. Cosa viene visualizzato dal seguente enunciato?  
`System.out.println(new Rectangle().getWidth());`
24. La classe `PrintStream` ha un costruttore che richiede come parametro il nome di un file. Come si costruisce un oggetto di tipo `PrintStream` usando il parametro di costruzione `"output.txt"`?
25. Scrivete un enunciato che memorizzi in una variabile l'oggetto costruito nella risposta alla domanda precedente.

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R2.13, R2.16 e R2.18, al termine del capitolo.



## Errori comuni 2.3

### Cercare di invocare un costruttore come se fosse un metodo

I costruttori non sono metodi. Possono essere utilizzati soltanto con l'operatore `new` e non per riportare un oggetto esistente al suo stato iniziale (cioè per “re-inizializzare un oggetto”):

```
box.Rectangle(20, 35, 20, 30); // Errore: non si può re-inizializzare un oggetto
```

La soluzione è semplice: create un nuovo oggetto e sovrascrivete quello che si trova in `box`.

```
box = new Rectangle(20, 35, 20, 30); // così va bene
```

## 2.5 Metodi d'accesso e metodi modificatori

Un metodo d'accesso non modifica i dati interni al suo parametro implicito, mentre un metodo modificatore lo fa.

In questo paragrafo presentiamo un'utile classificazione per i metodi di una classe. Un metodo che accede a un oggetto e restituisce alcune informazioni a esso relative, senza modificare l'oggetto stesso, viene chiamato **metodo d'accesso** (*accessor method*). Diversamente, un metodo che abbia lo scopo di modificare i dati interni di un oggetto viene chiamato **metodo modificatore** (*mutator method*).

Ad esempio, il metodo `length` della classe `String` è un metodo d'accesso: restituisce un'informazione relativa alla stringa (la sua lunghezza) ma non apporta alcuna modifica alla stringa stessa mentre ne conta i caratteri.

La classe `Rectangle` ha diversi metodi d'accesso: i metodi `getX`, `getY`, `getWidth` e `getHeight` restituiscono, rispettivamente, i valori delle coordinate `x` e `y` del vertice superiore sinistro del rettangolo, la sua larghezza e la sua altezza. Ad esempio:

```
double width = box.getWidth();
```

Esaminiamo ora un metodo modificatore. I programmi che manipolano rettangoli hanno spesso bisogno di spostarli, ad esempio per visualizzare animazioni. La classe `Rectangle` ha un metodo, `translate`, che serve proprio a questo: sposta un rettangolo di una specificata

quantità nelle direzioni  $x$  e  $y$  (in matematica si usa il termine “traslazione” per indicare uno spostamento rigido in un piano). La seguente invocazione

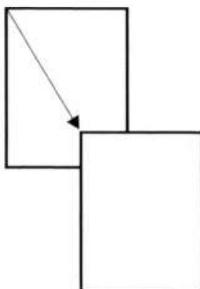
```
box.translate(15, 25);
```

sposta il rettangolo `box` di 15 unità nella direzione  $x$  e di 25 unità nella direzione  $y$ , come si può vedere nella Figura 12. Lo spostamento di un rettangolo non cambia la sua larghezza o la sua altezza, mentre modifica la posizione del suo vertice superiore sinistro: alla fine, il vertice superiore sinistro si trova nel punto (20, 35).

Questo metodo è un metodo modificatore perché modifica l’oggetto con cui viene invocato.

**Figura 12**

Uso del metodo `translate` per spostare un rettangolo



### Auto-valutazione

26. Cosa visualizza questa sequenza di enunciati?

```
Rectangle box = new Rectangle(5, 10, 20, 30);
System.out.println("Before: " + box.getX());
box.translate(25, 40);
System.out.println("After: " + box.getX());
```

27. Cosa visualizza questa sequenza di enunciati?

```
Rectangle box = new Rectangle(5, 10, 20, 30);
System.out.println("Before: " + box.getWidth());
box.translate(25, 40);
System.out.println("After: " + box.getWidth());
```

28. Cosa visualizza questa sequenza di enunciati?

```
String greeting = "Hello";
System.out.println(greeting.toUpperCase());
System.out.println(greeting);
```

29. Il metodo `toUpperCase` della classe `String` è un metodo d’accesso o un metodo modificatore?

30. Quale invocazione del metodo `translate` è necessaria per spostare il rettangolo dichiarato con `Rectangle box = new Rectangle(5, 10, 20, 30)` in modo che il suo vertice superiore sinistro si porti nel punto(0, 0), origine delle coordinate?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R.2.19, E2.7 e E2.9, al termine del capitolo.

## 2.6 La documentazione API

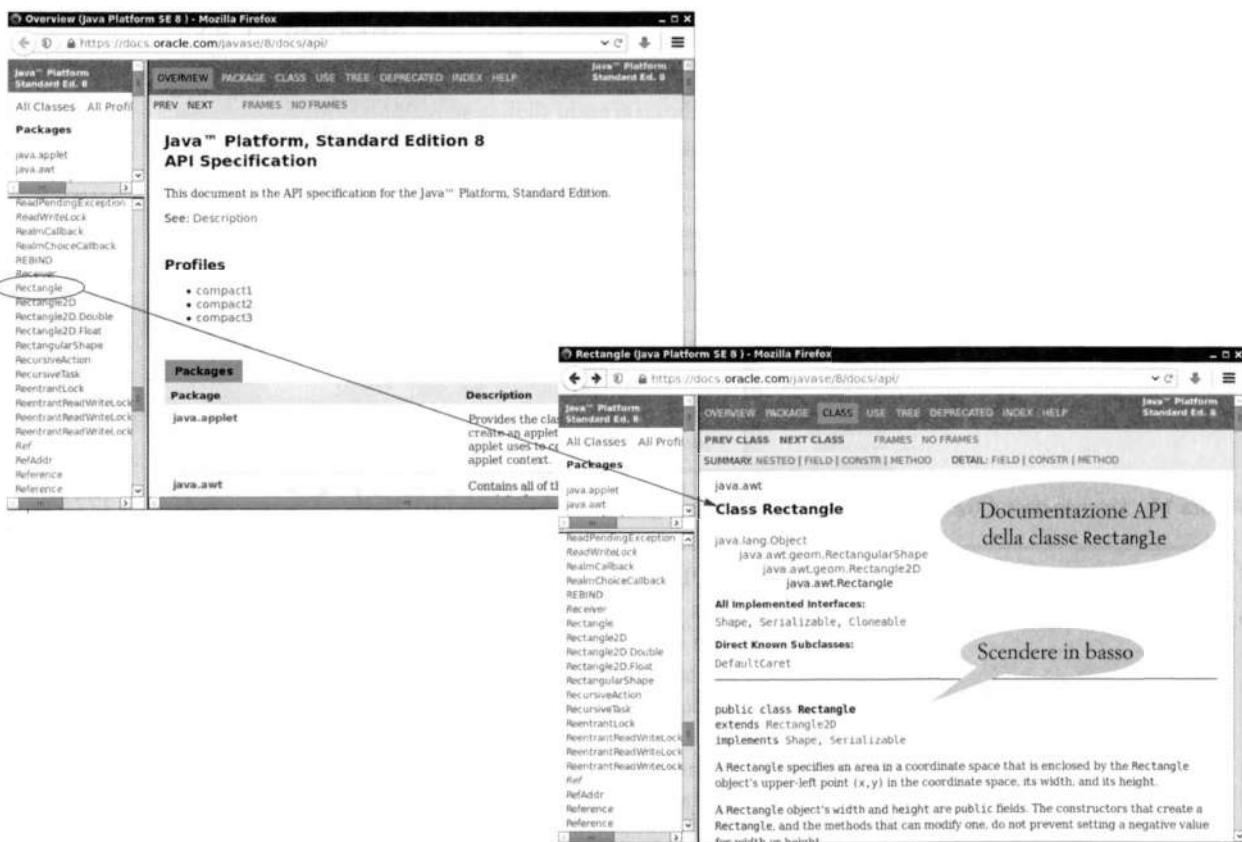
La documentazione API  
(Application Programming Interface)  
elena le classi e i metodi  
della libreria di Java.

Le classi e i metodi della libreria di Java sono elencati nella **documentazione API** (Application Programming Interface, cioè “interfaccia per la programmazione di applicazioni”). I *programmatori di applicazioni* usano le classi Java per realizzare programmi per computer, cioè *applicazioni*. Come voi! Al contrario, i programmatori che hanno progettato e realizzato le classi della libreria, come `PrintStream` e `Rectangle`, sono *programmatori di sistema*.

La documentazione API si può trovare sul Web, all’indirizzo <http://docs.oracle.com/javase/8/docs/api/index.html>.

### 2.6.1 Consultare la documentazione API

La documentazione API (ne vedete un esempio nella Figura 13) illustra tutte le classi della libreria di Java, che sono migliaia e spesso sono estremamente specifiche, per cui soltanto poche sono di interesse generale per gli apprendisti programmatori.



**Figura 13** La documentazione API per la libreria standard di Java

Cercate nel riquadro di sinistra il collegamento Rectangle, eventualmente utilizzando la funzione di ricerca del browser. Selezionate tale collegamento e vedrete visualizzata nel riquadro di destra la documentazione di tutte le caratteristiche della classe Rectangle, come nella parte destra della Figura 13.



Figura 14 L'elenco riassuntivo dei metodi della classe Rectangle

La documentazione API di ciascuna classe inizia con una sezione che ne descrive le finalità, poi si trova una tabella riassuntiva con l'elenco dei suoi costruttori e metodi, come nella parte sinistra della Figura 14: selezionando il nome di un metodo si giunge a una sua descrizione più approfondita, come mostrato nella parte destra della stessa figura.

La descrizione dettagliata di un metodo contiene:

- l'azione svolta dal metodo; ①
- i tipi e i nomi delle variabili a cui vengono assegnati i parametri con cui il metodo viene invocato; ②
- il valore restituito dal metodo (oppure la parola riservata `void`, se il metodo non restituisce un valore).

La classe `Rectangle` ha parecchi metodi: anche se questo può mettere inizialmente un po' in soggezione il programmatore principiante, si tratta della vera potenza della libreria standard. Se mai vi capiterà di fare elaborazioni con rettangoli, è molto probabile che ci siano già tutti i metodi di cui avrete bisogno.

Supponiamo, ad esempio, di voler modificare la larghezza o l'altezza di un rettangolo. Consultando la documentazione API, troverete un metodo `setSize`, avente questa descrizione: "Imposta la dimensione di questo `Rectangle` al valore specificato come larghezza e come altezza". Il metodo richiede due parametri, così descritti:

- `width` – la nuova larghezza di questo `Rectangle`
- `height` – la nuova altezza di questo `Rectangle`

Usiamo ora queste informazioni per modificare l'oggetto `box` in modo che diventi un quadrato con lato di lunghezza 40. Il nome del metodo è `setSize` e forniamo due parametri, la nuova lunghezza e la nuova altezza:

```
box.setSize(40, 40);
```

## 2.6.2 Pacchetti

La documentazione API fornisce, per ciascuna classe, anche un'altra informazione importante. Le classi della libreria standard sono organizzate in **pacchetti** (*package*), ciascuno dei quali è una raccolta di classi aventi funzioni tra loro correlate. La classe `Rectangle` appartiene al pacchetto `java.awt`, che contiene molte classi utili per disegnare finestre e forme grafiche (`awt` è l'abbreviazione di Abstract Windowing Toolkit, cioè "insieme di strumenti astratti per la realizzazione di finestre grafiche"). Nella parte destra della Figura 13 il nome del pacchetto, `java.awt`, è visibile proprio al di sopra del nome della classe.

Per usare la classe `Rectangle` del pacchetto `java.awt` occorre *importare* il pacchetto stesso, inserendo semplicemente la riga seguente all'inizio del programma:

```
import java.awt.Rectangle;
```

## Sintassi di Java

### 2.4 Importazione di una classe da un pacchetto

#### Sintassi

```
import nomePacchetto.NomeClasse;
```

#### Esempio

Nome del pacchetto	Nome della classe
<pre>import java.awt.Rectangle;</pre>	

Gli enunciati di importazione devono trovarsi all'inizio del file sorgente.

Il nome del pacchetto si trova nella documentazione API.

Perché non abbiamo avuto bisogno di importare le classi `System` e `String`? Perché tali classi si trovano nel pacchetto `java.lang`, le cui classi vengono importate automaticamente: non c'è mai bisogno di importarle in modo esplicito.



### Auto-valutazione

31. Consultate la documentazione API della classe `String`. Quale metodo usereste per ottenere la stringa "hello, world!" a partire dalla stringa "Hello, World!"?
32. Consultate la descrizione del metodo `trim` nella documentazione API della classe `String`. Qual è il risultato dell'applicazione del metodo `trim` alla stringa " Hello, Space ! " ? (Notate gli spazi presenti nella stringa)
33. Consultate la documentazione API della classe `Rectangle`. In cosa differiscono le azioni svolte dai metodi `void translate(int x, int y)` e `void setLocation(int x, int y)`?
34. La classe `Random` viene dichiarata nel pacchetto `java.util`. Cosa bisogna fare per usare tale classe in un programma?
35. In quale pacchetto si trova la classe `BigInteger`? Cercatelo nella documentazione API.

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R2.20, E2.5 e E2.12, al termine del capitolo.



### Suggerimenti per la programmazione 2.3

#### Non usate la memoria: usate la documentazione!

La libreria di Java contiene migliaia di classi, con i relativi metodi, ma non è affatto necessario né utile cercare di memorizzarli: al contrario, cercate di acquisire dimestichezza con la consultazione della documentazione API. Dato che tale documentazione sarà il vostro strumento di lavoro quotidiano, è preferibile che la scarichiate e la installiate sul vostro computer, soprattutto se non siete permanentemente collegati a Internet. La documentazione può essere scaricata all'indirizzo <http://www.oracle.com/technetwork/java/javase/downloads/index.html>.

## 2.7 Realizzare un programma di collaudo

Un programma di collaudo verifica che i metodi si comportino come previsto.

In questo paragrafo illustreremo le fasi necessarie per la progettazione di un programma di collaudo, che ha come obiettivo la verifica della corretta realizzazione di uno o più metodi, invocandoli e verificando che restituiscano i valori previsti: un'attività veramente molto importante.

Svilupperemo qui un semplice programma che collauda un metodo della classe `Rectangle` eseguendo queste fasi:

1. Definisce una classe di collaudo.
2. Definisce in essa il metodo `main`.
3. Costruisce uno o più oggetti all'interno del metodo `main`.
4. Applica metodi agli oggetti.

5. Visualizza i risultati delle invocazioni dei metodi.
6. Visualizza i valori previsti da tali invocazioni.

Il programma di collaudo che usiamo come esempio, di cui presentiamo qui le parti salienti (inserite nel metodo `main` della classe `MoveTester`), verifica la funzionalità del metodo `translate`:

```
Rectangle box = new Rectangle(5, 10, 20, 30);

// sposta il rettangolo
box.translate(15, 25);

// visualizza informazioni relative al rettangolo spostato
System.out.print("x: ");
System.out.println(box.getX()); // risultato ottenuto
System.out.println("Expected: 20"); // risultato previsto
```

Visualizziamo il valore restituito dal metodo `getX`, seguito da un messaggio che riporta il valore che ci si aspetta di ottenere.

La determinazione a priori dei risultati attesi è un'attività essenziale nel collaudo.

Questo è un aspetto molto importante: prima di eseguire un programma di collaudo, dovete pensare con calma e capire quali risultati vi aspettate che vengano prodotti. Questa attenta riflessione vi aiuterà a capire come si dovrebbe comportare il vostro programma e può essere di grande aiuto nell'individuazione di errori fin dalle prime fasi di sviluppo. Trovare e correggere tempestivamente gli errori presenti in un programma è una strategia estremamente efficace, che porta a grandi risparmi di tempo.

Nel nostro caso, il rettangolo viene costruito con l'angolo superiore sinistro nella posizione (5, 10), poi viene spostato di 15 pixel nella direzione *x*, per cui ci aspettiamo che dopo lo spostamento la coordinata *x* dell'angolo superiore sinistro abbia il valore  $5 + 15 = 20$ .

Ecco un programma completo che collauda lo spostamento di un rettangolo.

### File MoveTester.java

```
1 import java.awt.Rectangle;
2
3 public class MoveTester
4 {
5     public static void main(String[] args)
6     {
7         Rectangle box = new Rectangle(5, 10, 20, 30);
8
9         // sposta il rettangolo
10        box.translate(15, 25);
11
12        // visualizza informazioni sul rettangolo traslato
13        System.out.print("x: ");
14        System.out.println(box.getX());
15        System.out.println("Expected: 20");
16
17        System.out.print("y: ");
18        System.out.println(box.getY());
19        System.out.println("Expected: 35");
20    }
21 }
```

### Esecuzione del programma

```
x: 20
Expected: 20
y: 35
Expected: 35
```



### Auto-valutazione

36. Se avessimo invocato `box.translate(25,15)` invece di `box.translate(15, 25)`, quali sarebbero stati i valori visualizzati?
37. Perché non c'è bisogno che il programma `MoveTester` visualizzi la larghezza e l'altezza del rettangolo?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi E2.1, E2.8 e E2.14, al termine del capitolo.



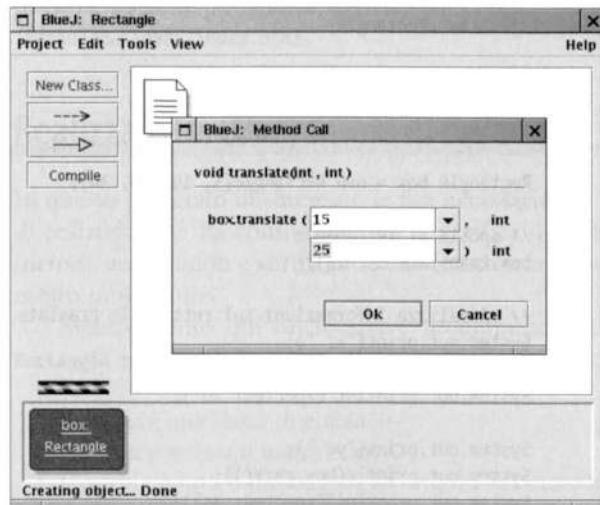
## Argomenti avanzati 2.1

### Collaudare classi in un ambiente interattivo

Alcuni ambienti di sviluppo sono specificamente progettati per aiutare gli studenti a esplorare gli oggetti senza dover scrivere classi di collaudo e possono essere molto utili nell'apprendimento del comportamento degli oggetti e nella diffusione del pensiero orientato agli oggetti. L'ambiente BlueJ visualizza gli oggetti come elementi su un tavolo da lavoro.

Potete costruire nuovi oggetti, posizionarli sul tavolo da lavoro, invocarne metodi e osservare i valori da essi restituiti, senza dover scrivere una sola riga di codice. BlueJ può essere scaricato gratuitamente all'indirizzo <http://www.bluej.org>. Un altro eccellente ambiente per l'esplorazione interattiva di oggetti è Dr. Java (<http://drjava.sourceforge.net>).

Collaudo dell'invocazione  
di un metodo con BlueJ





## Esempi completi 2.1

### Quanti giorni sono trascorsi dalla vostra nascita?

In molti programmi è necessario elaborare date, come "15 febbraio 2010": nel pacchetto dei file scaricabili per questo libro, la cartella `worked_example_1` del Capitolo 2 contiene il codice sorgente di una classe `Day` progettata per elaborare date di calendario.

La classe `Day` è a conoscenza di tutti i dettagli e le stranezze del nostro calendario, per cui sa che il mese di gennaio ha 31 giorni, mentre febbraio ne può avere 28 o 29. Con il calendario giuliano, introdotto da Giulio Cesare nel primo secolo avanti Cristo, venne istituita la regola che prevede la presenza di un anno bisestile ogni quattro anni. Nel 1582, Papa Gregorio XIII ordinò l'adozione del calendario utilizzato oggi in tutto il mondo, chiamato calendario gregoriano: in esso, la regola per la determinazione degli anni bisestili è stata complicata e resa più aderente alla realtà astronomica, specificando che gli anni divisibili per 100 non sono bisestili, a meno che non siano divisibili per 400. Di conseguenza, l'anno 1900 non fu bisestile, diversamente dal 2000, che lo fu. Tutti questi dettagli vengono gestiti dai meccanismi interni della classe `Day`.

La classe `Day` consente di rispondere a domande di questo tipo:

- Quanti giorni mancano da oggi alla fine dell'anno?
- Trascorsi 100 giorni da oggi, che giorno sarà?

**Problema.** Dovete scrivere un programma che calcoli il numero di giorni trascorsi dalla vostra nascita. Dovreste *evitare* di guardare i dettagli realizzativi interni della classe `Day`: usate, invece, la relativa documentazione API, consultabile nel file `index.html` presente nella sotto-cartella `api`.

Ecco come si costruisce un oggetto di tipo `Day` a partire da tre valori (anno, mese e giorno):

```
Day jamesGoslingBirthday = new Day(1955, 5, 19);
```

C'è, poi, un metodo che consente di aggiungere giorni a una certa data, in questo modo:

```
Day later = jamesGoslingBirthday.addDays(100);
```

Per ispezionare il risultato prodotto, si possono usare i metodi `getYear`/`getMonth`/`getDate`:

```
System.out.println(later.getYear());
System.out.println(later.getMonth());
System.out.println(later.getDate());
```

Queste funzionalità, però, non risolvono il nostro problema (a meno che non si voglia sostituire il valore 100 con altri valori, provando e riprovando fino ad ottenere la data odierna): bisogna usare il metodo `daysFrom`. In base alla documentazione, dobbiamo fornire come parametro un'altra data, per cui il metodo va invocato in questo modo:

```
int daysAlive = day1.daysFrom(day2);
```

Nella situazione che ci interessa, uno degli oggetti di tipo `Day` è `jamesGoslingBirthday`, mentre l'altro deve rappresentare la data odierna. Quest'ultimo si può ottenere usando il costruttore privo di parametri:

```
Day today = new Day();
```

A questo punto, ci si prospettano due diversi modi per invocare il metodo `daysFrom`, scrivendo

```
int daysAlive = jamesGoslingBirthday.daysFrom(today);
```

oppure

```
int daysAlive = today.daysFrom(jamesGoslingBirthday);
```

Qual è la scelta giusta? Fortunatamente gli autori della classe `Day` hanno previsto questa domanda e il dettagliato commento fornito a corredo del metodo `daysFrom` contiene questa frase:

Returns: the number of days that this day is away from the other  
(larger than 0 if this day comes later than other)

Dicono, cioè, che il metodo “restituisce il numero di giorni che separano questo giorno dall’altro (un valore positivo se questo giorno viene dopo l’altro)”. Noi vogliamo un risultato positivo, quindi la forma corretta è la seconda.

Ecco, infine, il programma che risolve il nostro problema:

### File DaysAlivePrinter.java

```
1 public class DaysAlivePrinter
2 {
3     public static void main(String[] args)
4     {
5         Day jamesGoslingsBirthday = new Day(1955, 5, 19);
6         Day today = new Day();
7         System.out.print("Today: ");
8         System.out.println(today.toString());
9         int daysAlive = today.daysFrom(jamesGoslingsBirthday);
10        System.out.print("Days alive: ");
11        System.out.println(daysAlive);
12    }
13 }
```

### Esecuzione del programma:

```
Today: 2015-02-09
Days alive: 21826
```



## Esempi completi 2.2

### Lavorare con immagini

Nel pacchetto dei file scaricabili per questo libro, la cartella `worked_example_2` del Capitolo 2 contiene il codice sorgente di una classe, `Picture`, che consente di modificare e visualizzare file contenenti immagini.

Ad esempio, il programma seguente ha il semplice scopo di visualizzare un'immagine:

```
public class PictureDemo
{
    public static void main(String[] args)
    {
        Picture pic = new Picture();
        pic.load("queen-mary.png");
    }
}
```



**Problema.** Dovete scrivere un programma che legga un'immagine da un file, ne riduca le dimensioni e vi aggiunga un bordo. Le dimensioni dell'immagine vanno ridotte quanto basta perché ci sia un bordo trasparente all'interno del bordo nero, come in questa seconda figura.



Dovreste *evitare* di guardare i dettagli realizzativi interni della classe `Picture`. Usate, invece, la relativa documentazione API, consultabile nel file `index.html` presente nella sotto-cartella `api`.

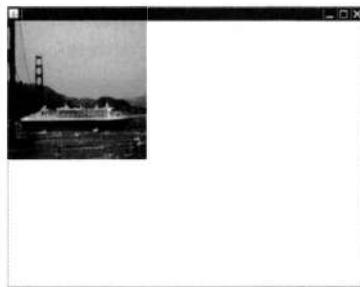
La classe contiene alcuni metodi che non servono al nostro scopo, ma due di essi ci saranno decisamente utili:

```
public void scale(int newWidth, int newHeight)
public void border(int width)
```

Se i commenti di alcuni metodi non vi sono chiari, è sempre utile scrivere un paio di semplici programmi di collaudo che consentano di vederne gli effetti. Ad esempio, questo programma mostra il metodo `scale` in azione:

```
public class PictureScaleDemo
{
    public static void main(String[] args)
    {
        Picture pic = new Picture();
        pic.load("queen-mary.png");
        pic.scale(200, 200);
    }
}
```

Ecco il risultato:



Come potete vedere, l'immagine è stata ridimensionata, portandola alla dimensione di un quadrato 200 per 200.

Ma questo non è esattamente ciò che volevamo: vogliamo, invece, ottenere un'immagine poco più piccola dell'originale. Diciamo che il bordo nero abbia uno spessore di 10 pixel e che, al suo interno, vogliamo avere un'ulteriore bordo trasparente di 10 pixel: di conseguenza, la larghezza e l'altezza a cui vogliamo arrivare sono inferiori di 40 pixel rispetto ai valori originari, lasciando così 20 pixel liberi su ogni lato per i bordi.

Consultando la documentazione API, troviamo i metodi che ci consentono di ispezionare i valori originari di larghezza e altezza dell'immagine, per cui scriveremo:

```
int newWidth = pic.getWidth() - 40;
int newHeight = pic.getHeight() - 40;
pic.scale(newWidth, newHeight);
```

Infine, aggiungiamo il bordo:

```
pic.border(10);
```

ottenendo questo risultato:



Se riusciamo a spostare un po' l'immagine prima di applicarvi il bordo, ce l'abbiamo fatta. Consultando nuovamente la documentazione, scopriamo il metodo

```
public void move(int dx, int dy)
```

che è proprio quello che ci serve: l'immagine va spostata di 20 pixel verso il basso e verso destra. Il nostro programma completo è:

#### File BorderMaker.java

```
1 public class BorderMaker
2 {
3     public static void main(String[] args)
4     {
5         Picture pic = new Picture();
6         pic.load("queen-mary.png");
7         int newWidth = pic.getWidth() - 40;
8         int newHeight = pic.getHeight() - 40;
9         pic.scale(newWidth, newHeight);
10        pic.move(20, 20);
11        pic.border(10);
12    }
13 }
```

Avremmo potuto ottenere il medesimo risultato usando un programma di elaborazione d'immagini, come Photoshop o GIMP? Certamente sì, ma a questo punto è semplice aggiungere funzionalità al nostro programma, in modo che possa automaticamente applicare un bordo a molte immagini diverse.

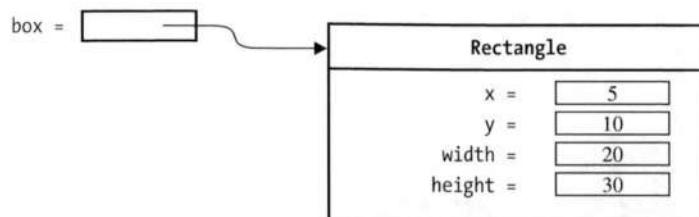
## 2.8 Riferimenti a oggetti

In Java, una “variabile oggetto” (cioè una variabile il cui tipo dichiarato sia una classe) non contiene, in realtà, un oggetto: memorizza al proprio interno soltanto la *posizione* dell’oggetto all’interno della memoria. L’oggetto è memorizzato altrove, come si può vedere nella Figura 15.

C’è ovviamente una motivazione per questo comportamento: gli oggetti possono essere molto grandi, cioè occupare una zona di memoria di grandi dimensioni. Conseguentemente, è più efficiente memorizzare soltanto la posizione dell’oggetto, piuttosto dell’oggetto intero.

**Figura 15**

Una variabile oggetto contenente un riferimento a un oggetto



Un riferimento a un oggetto descrive la posizione dell'oggetto in memoria.

Per indicare la posizione di un oggetto all'interno della memoria del computer usiamo il termine tecnico **riferimento a oggetto** (*object reference*) e quando una variabile contiene la posizione in memoria di un oggetto diciamo che *fa riferimento* o *si riferisce* a quell'oggetto. Ad esempio, dopo l'esecuzione dell'enunciato

```
Rectangle box = new Rectangle(5, 10, 20, 30);
```

la variabile `box` fa riferimento all'oggetto di tipo `Rectangle` costruito dall'operatore `new`. Tecnicamente parlando, l'operatore `new` ha restituito un riferimento al nuovo oggetto: proprio tale riferimento viene memorizzato nella variabile `box`.

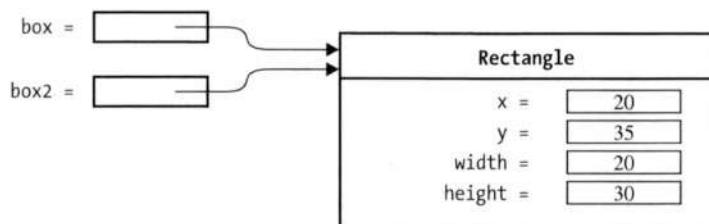
Occorre assolutamente ricordare che la variabile `box` *non contiene* l'oggetto, ma *fa solamente riferimento* a esso. Si possono anche avere due variabili oggetto che fanno riferimento al medesimo oggetto:

```
Rectangle box2 = box;
```

A questo punto è possibile accedere al medesimo oggetto di tipo `Rectangle` usando indifferentemente la variabile `box` oppure la variabile `box2`, come si può vedere nella Figura 16.

**Figura 16**

Due variabili oggetto che fanno riferimento al medesimo oggetto



In Java, i numeri non sono oggetti. Le variabili numeriche contengono veramente numeri, per cui, quando si dichiara

```
int luckyNumber = 13;
```

la variabile `luckyNumber` contiene effettivamente il numero 13, come si può vedere nella Figura 17, e non un riferimento al numero. Il motivo di questo diverso comportamento è, di nuovo, l'efficienza: dal momento che i numeri necessitano di una modesta quantità di memoria, è più efficiente memorizzarli direttamente in una variabile.

**Figura 17**

Una variabile di tipo numerico memorizza un numero

`luckyNumber = 13`

Le variabili numeriche memorizzano numeri, mentre le variabili oggetto memorizzano riferimenti.

Quando si copia una variabile si apprezza bene la differenza tra variabili di tipo numerico e variabili oggetto. Quando si copia un numero, il numero originario e la sua copia sono valori tra loro indipendenti, mentre quando si copia un riferimento a un oggetto l'originale e la copia sono riferimenti al medesimo oggetto.

Considerate il codice seguente, che copia un numero e poi modifica la variabile contenente la copia appena effettuata, come rappresentato nella Figura 18:

```
int luckyNumber = 13;
int luckyNumber2 = luckyNumber;
luckyNumber2 = 12;
```

**Figura 18**  
Copiatura di numeri



A questo punto la variabile `luckyNumber` contiene il valore 13, mentre `luckyNumber2` contiene 12.

Considerate ora il codice, apparentemente simile, che effettua la copia di oggetti di tipo `Rectangle`, come rappresentato nella Figura 19:

```
Rectangle box = new Rectangle(5, 10, 20, 30);
Rectangle box2 = box;
box2.translate(15, 25);
```

Dal momento che `box` e `box2`, in seguito all'esecuzione del secondo enunciato, fanno riferimento al medesimo rettangolo, dopo l'invocazione del metodo `translate` entrambe le variabili si riferiscono al rettangolo spostato.

Non c'è, però, bisogno di preoccuparsi troppo della differenza tra oggetti e riferimenti a oggetti: la maggior parte delle volte sarete in grado di intuire chiaramente che quando si parla dell'"oggetto `box`" si sta in realtà parlando del "riferimento (a un oggetto) memorizzato in `box`", ricordando comunque che quest'ultima sarebbe la definizione più corretta. La differenza tra oggetti e riferimenti a oggetti diviene evidente soltanto quando si hanno più variabili che fanno riferimento al medesimo oggetto.



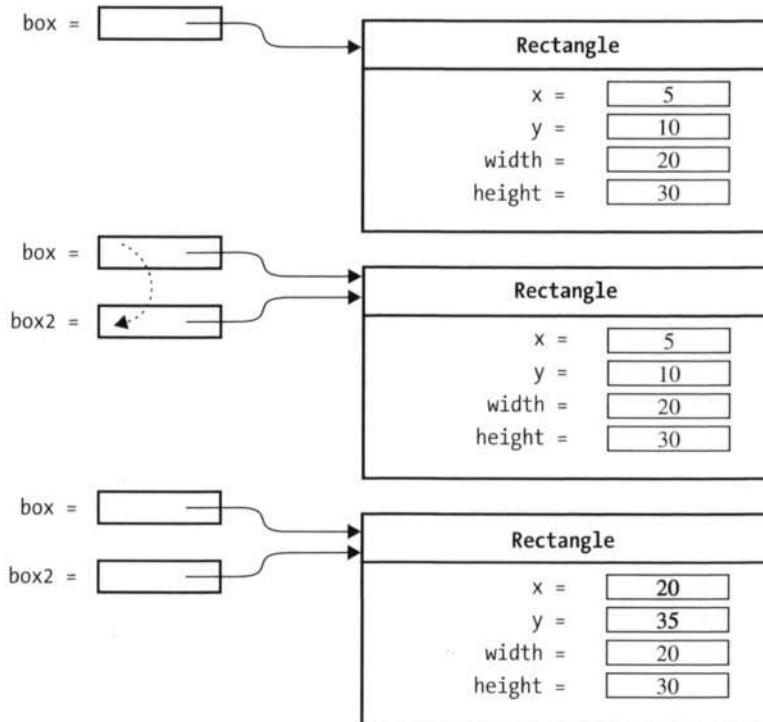
## Auto-valutazione

38. Che effetto ha l'assegnazione `String greeting2 = greeting`?
39. Dopo l'invocazione `greeting2.toUpperCase()`, cosa contengono `greeting` e `greeting2`?

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi R.2.17 e R.2.21, al termine del capitolo.

**Figura 19**  
Copiatura di riferimenti a oggetti



## Computer e società 2.1

### Il monopolio dei calcolatori

Quando, nei primi Anni Cinquanta, International Business Machines Corporation (IBM), un'affermata società che costruiva apparecchiature a schede perforate per l'archiviazione di dati, cominciò a interessarsi alla progettazione di computer, i suoi responsabili della pianificazione stimarono che forse esistesse un mercato per non più di 50 apparecchiature di quel tipo, destinate a organizzazioni governative o militari e ad alcune delle più grandi aziende del Paese. Invece, vendettero circa 1500 esemplari del loro modello System 650 e iniziarono a produrre e vendere computer più potenti.

Quei computer, chiamati *mainframe*, erano enormi: riempivano intere stanze, che bisognava climatizzare per proteggerne le delicate apparecchiature. IBM non fu la prima azienda a costruire computer mainframe: questo primato appartiene a Univac Corporation. Tuttavia, ben presto IBM si guadagnò il ruolo principale, in parte grazie alla qualità tecnologica e all'attenzione posta alle necessità dei clienti, e in parte perché sfruttò il suo predominio, strutturando i suoi prodotti e i suoi servizi in modo da rendere difficile ai clienti il loro utilizzo insieme a quelli di altri fornitori.

Nel momento in cui i concorrenti di IBM stavano per essere

sostanzialmente sconfitti, nel 1969 il governo degli Stati Uniti intentò una causa contro IBM per abuso di posizione dominante nel mercato, secondo la legge *antitrust*. Negli Stati Uniti è consentito avere il monopolio in

Un computer mainframe



un settore, ma non è lecito usare il proprio monopolio in un settore per acquisire un ruolo dominante in un altro settore, per cui IBM venne accusata di costringere i propri clienti ad acquistare in blocco computer, software e dispositivi periferici, rendendo impossibile la competizione sul mercato da parte di altri fornitori di software e di periferiche.

Il dibattimento iniziò nel 1975, trascinandosi fino al 1982, quando venne abbandonato, perché l'avvento di computer di minori dimensioni rese il fenomeno irrilevante.

In effetti, quando IBM presentò il suo primo personal computer, il suo sistema operativo era fornito da un'azienda esterna, Microsoft, che poi divenne talmente dominante nel settore da essere incriminata

dal governo statunitense per abuso della propria posizione monopolista, nel 1998. Microsoft fu accusata di inserire il proprio browser Web all'interno del sistema operativo e, in quel periodo, presumibilmente minacciò i produttori di hardware di non fornire loro la licenza di Windows se avessero distribuito il browser concorrente, Netscape. Nel 2000 l'azienda venne dichiarata colpevole di violazioni della legge antitrust e il giudice ordinò che venisse suddivisa in un'unità dedicata al sistema operativo e un'altra che si occupasse di applicazioni. L'ordine di scioglimento fu revocato in appello e nel 2001 fu stipulato un accordo, che però ebbe ben poco successo nell'individuazione di alternative per applicazioni software.

Oggi il panorama nel mondo del software sta cambiando di nuovo, orientandosi verso i dispositivi mobili e l'elaborazione diffusa (*cloud computing*): osservando questi cambiamenti nel prossimo futuro, è probabile che si assista all'instaurazione di nuovi monopoli. Quando uno sviluppatore di software deve chiedere a un produttore di hardware il permesso di distribuire un proprio prodotto all'interno di un "app store", o quando il produttore di un lettore di libri digitali cerca di costringere gli editori ad adottare una determinata politica dei prezzi, ci si deve chiedere se tali comportamenti siano, a tutti gli effetti, uno sfruttamento illecito di una posizione di monopolio.

## 2.9 Applicazioni grafiche

I paragrafi che seguono sono facoltativi rispetto al flusso didattico del testo e insegnano a progettare *applicazioni grafiche*, cioè applicazioni che visualizzano disegni all'interno di una finestra. I disegni sono costituiti da oggetti che rappresentano forme: rettangoli, ellissi e segmenti. Questi oggetti saranno ulteriori esempi utili per la programmazione e molti studenti ne gradiscono l'effetto visivo.

### 2.9.1 Finestre con cornice

Per visualizzare una finestra con cornice (frame) si costruisce un oggetto di tipo JFrame, se ne impostano le dimensioni e lo si rende visibile.

Un'applicazione grafica visualizza informazioni all'interno di un **frame**, cioè una "finestra con cornice", dotata di una barra del titolo, come mostrato nella Figura 20. In questo paragrafo vedrete come si visualizza un frame, mentre nel Paragrafo 2.9.2 imparerete a creare disegni al suo interno.

Per visualizzare un frame occorre seguire queste fasi:

1. Costruire un oggetto della classe JFrame:

```
JFrame frame = new JFrame();
```

2. Impostare le dimensioni del frame:

```
frame.setSize(300, 400);
```

**Figura 20**

Una finestra con cornice  
(frame)



Questo frame sarà largo 300 pixel e alto 400 pixel (i pixel sono i piccoli punti che costituiscono un'immagine digitale). Se dimenticate questa fase otterrete un frame di dimensioni 0 per 0, che non sarà visibile.

3. Assegnare eventualmente un titolo al frame (altrimenti la barra del titolo rimarrà vuota):

```
frame.setTitle("An Empty Frame");
```

4. Decidere quale sarà la “operazione di chiusura predefinita”:

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Quando l’utente chiude la finestra, il programma termina automaticamente la propria esecuzione. Non dimenticate questo passaggio, altrimenti il programma continuerà a restare in esecuzione indefinitamente, anche dopo la chiusura della finestra.

5. Rendere visibile il frame:

```
frame.setVisible(true);
```

Il semplice programma che segue mostra tutte queste fasi e produce il frame vuoto che potete vedere nella Figura 20.

La classe `JFrame` fa parte del pacchetto `javax.swing`. `Swing` è il nome della libreria Java per le interfacce grafiche destinate all’interazione con l’utente, mentre la lettera “x” alla fine di `javax` sta a indicare che il pacchetto `Swing` era inizialmente una *estensione* di Java, prima di essere aggiunto alla libreria standard.

Nei Capitoli 3 e 10 entreremo in maggiore dettaglio nella programmazione con il pacchetto `Swing`; per il momento tenete soltanto presente che questo programma costituisce l’intelaiatura essenziale che è richiesta per visualizzare un frame.

### File EmptyFrameViewer.java

```

1 import javax.swing.JFrame;
2
3 public class EmptyFrameViewer
4 {
5     public static void main(String[] args)
6     {
7         JFrame frame = new JFrame();
8         frame.setSize(300, 400);
9         frame.setTitle("An Empty Frame");
10        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        frame.setVisible(true);
12    }
13 }
```

#### 2.9.2 Disegnare in un componente

In questo paragrafo vedrete come tracciare figure all'interno di una finestra con cornice. Il primo disegno sarà davvero modesto, una coppia di rettangoli (Figura 21), ma presto imparerete a tracciare figure più interessanti. Lo scopo di questo esempio è quello di illustrare lo schema elementare di un programma che disegna figure.

**Figura 21**

Una coppia di rettangoli



Per visualizzare qualcosa in un frame,  
occorre dichiarare una classe  
che estenda la classe `JComponent`

Non è possibile disegnare direttamente in un frame, serve un oggetto di tipo **componente**. Nell'insieme di strumenti Swing, la classe `JComponent` rappresenta un componente vuoto.

Dato che non vogliamo aggiungere al frame un componente vuoto, dobbiamo modificare la classe `JComponent`, specificando come debba essere disegnato il componente stesso. La soluzione consiste nel dichiarare una nuova classe che estenda la classe `JComponent`, un procedimento che vedrete in dettaglio nel Capitolo 9.

Per il momento, seguite semplicemente lo schema qui proposto:

```
public class RectangleComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        istruzioni per disegnare il componente
    }
}
```

Inserite le istruzioni di disegno all'interno del metodo `paintComponent`, che viene invocato ogni volta che il componente deve essere ridisegnato.

Per ottenere l'oggetto di tipo `Graphics2D` a partire dal parametro di tipo `Graphics` del metodo `paintComponent` bisogna usare un **cast**.

La parola riservata `extends` indica che la nostra classe, `RectangleComponent`, può essere utilizzata come se fosse `JComponent`, anche se differisce per un aspetto importante: il metodo `paintComponent` conterrà istruzioni per disegnare i rettangoli.

Il metodo `paintComponent` viene invocato automaticamente quando il componente viene visualizzato per la prima volta; poi, viene invocato di nuovo ogni volta che la finestra viene ridimensionata oppure torna visibile dopo essere stata nascosta.

Il metodo `paintComponent` riceve come parametro un oggetto di tipo `Graphics`, che memorizza lo stato grafico utilizzato per le operazioni di disegno: il valore attuale del colore utilizzato per tracciare le forme, il font attualmente selezionato per la visualizzazione di testi, e così via.

La classe `Graphics`, però, non è molto utile. Quando i programmatore protestarono perché venisse seguito un approccio grafico maggiormente orientato agli oggetti, i progettisti di Java crearono la classe `Graphics2D`, che estende la classe `Graphics`. Ogni volta che l'insieme di strumenti Swing invoca il metodo `paintComponent`, viene utilizzato come parametro effettivo un oggetto di tipo `Graphics2D`. Dal momento che vogliamo usare i metodi più complessi, che consentono di disegnare oggetti grafici bidimensionali, abbiamo la necessità di usare la classe `Graphics2D`, azione possibile mediante l'utilizzo di un **cast**:

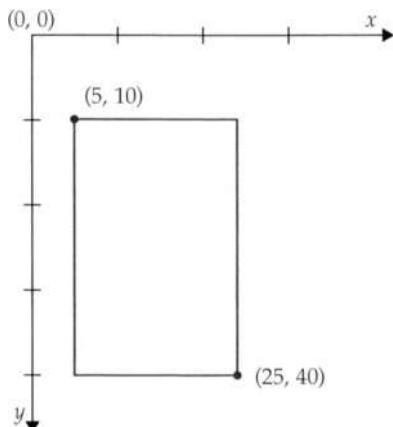
```
public class RectangleComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        // recupera il riferimento a Graphics2D
        Graphics2D g2 = (Graphics2D) g;
        ...
    }
}
```

Il Capitolo 9 spiegherà meglio l'operazione di cast: per ora, inserite semplicemente il cast all'inizio dei vostri metodi `paintComponent`, così come lo abbiamo presentato qui.

A questo punto siete pronti per disegnare. Il metodo `draw` della classe `Graphics2D` è in grado di disegnare forme geometriche, come rettangoli, ellissi, segmenti di retta, poligoni e archi. Ecco come si disegna un rettangolo:

```
public class RectangleComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        ...
        Rectangle box = new Rectangle(5, 10, 20, 30);
        g2.draw(box);
        ...
    }
}
```

Quando si posizionano le figure, occorre fare attenzione al sistema di coordinate, che è diverso da quello usato in matematica. L'origine, (0, 0), si trova nell'angolo superiore sinistro del componente e le coordinate  $y$  crescono verso il basso.



Infine, ecco il codice sorgente completo per la classe `RectangleComponent`. Notate che il suo metodo `paintComponent` disegna due rettangoli.

Come potete notare dagli enunciati `import`, le classi `Graphics` e `Graphics2D` fanno parte del pacchetto `java.awt`.

### File `RectangleComponent.java`

```
1 import java.awt.Graphics;
2 import java.awt.Graphics2D;
3 import java.awt.Rectangle;
4 import javax.swing.JComponent;
5
6 /**
7     Un componente che disegna due rettangoli.
8 */
9 public class RectangleComponent extends JComponent
10 {
11     public void paintComponent(Graphics g)
12     {
13         // recupera il riferimento a Graphics2D
14         Graphics2D g2 = (Graphics2D) g;
15
16         // costruisce un rettangolo e lo disegna
17         Rectangle box = new Rectangle(5, 10, 20, 30);
18         g2.draw(box);
19
20         // sposta il rettangolo di 15 unità a destra e 25 unità in basso
21         box.translate(15, 25);
22
23         // disegna il rettangolo nella nuova posizione
```

```

24         g2.draw(box);
25     }
26 }
```

### 2.9.3 Visualizzare un componente in un frame

In un'applicazione grafica serve un frame per visualizzare l'applicazione stessa e un componente per poter disegnare qualcosa: ora vedremo come combinare le due cose. Fate così:

1. Costruite un frame e configuratevelo.
2. Costruite un oggetto della vostra classe componente:

```
RectangleComponent component = new RectangleComponent();
```

3. Aggiungete il componente al frame:

```
frame.add(component);
```

4. Rendete visibile il frame.

Ecco il codice completo.

#### File RectangleViewer.java

```

1 import javax.swing.JFrame;
2
3 public class RectangleViewer
4 {
5     public static void main(String[] args)
6     {
7         JFrame frame = new JFrame();
8
9         frame.setSize(300, 400);
10        frame.setTitle("Two rectangles");
11        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12
13        RectangleComponent component = new RectangleComponent();
14        frame.add(component);
15
16        frame.setVisible(true);
17    }
18 }
```

Notate che il programma per disegnare rettangoli è composto da due classi:

- La classe `RectangleComponent`, il cui metodo `paintComponent` genera il disegno.
- La classe `RectangleViewer`, il cui metodo `main` costruisce un frame e vi aggiunge un componente di tipo `RectangleComponent`, rendendo poi visibile il frame.



## Auto-valutazione

40. Come si visualizza un frame quadrato con la scritta "Hello,World!" nella barra del titolo?
41. Come può un programma visualizzare contemporaneamente due frame?
42. Come si modifica il programma in modo che disegni due quadrati?
43. Come si modifica il programma in modo che disegni un rettangolo e un quadrato?
44. Cosa succede se si invoca `g.draw(box)` invece di `g2.draw(box)`?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R.2.22, R.2.26 e E.2.18, al termine del capitolo.

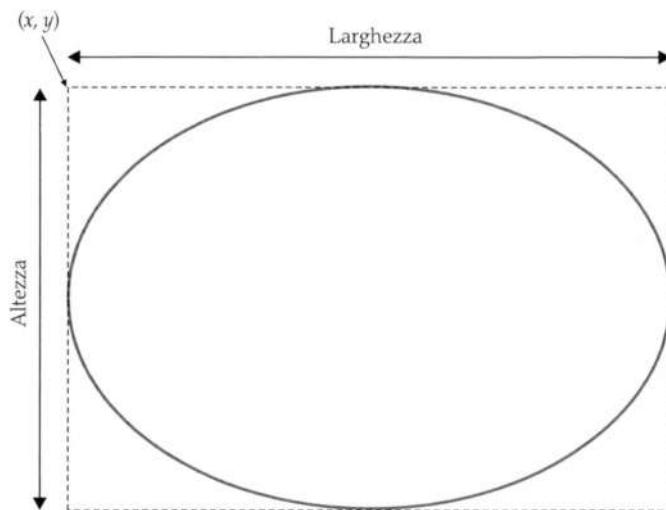
## 2.10 Ellissi, segmenti, testo e colore

Nel Paragrafo 2.9 avete imparato a scrivere un programma che disegna rettangoli: qui imparerete a tracciare altre forme geometriche, come ellissi e segmenti. Grazie a questi elementi grafici potrete comporre immagini piuttosto interessanti.

### 2.10.1 Ellissi e cerchi

Per disegnare un'ellisse occorre specificare il suo rettangolo di delimitazione (*bounding box*, illustrato nella Figura 22), analogamente a come definireste un rettangolo, cioè tramite le coordinate *x* e *y* del vertice superiore sinistro, la larghezza (*width*) e l'altezza (*height*).

**Figura 22**  
Un'ellisse e il suo  
rettangolo di delimitazione



`Ellipse2D.Double` e  
`Line2D.Double` sono classi  
che descrivono forme grafiche.

Tuttavia, non esiste una semplice classe `Ellipse`: bisogna, invece, utilizzare una delle due classi `Ellipse2D.Float` e `Ellipse2D.Double`, a seconda che vogliate esprimere le coordinate dell'ellisse mediante valori in virgola mobile in semplice o, rispettivamente, doppia precisione. Dal momento che in Java la seconda opzione è più diffusa, utilizzeremo sempre la classe `Ellipse2D.Double`.

Ecco come si costruisce un'ellisse:

```
Ellipse2D.Double ellipse = new Ellipse2D.Double(x, y, width, height);
```

Il nome della classe, `Ellipse2D.Double`, appare diverso dai nomi di classe visti finora: è formato dai nomi delle due classi `Ellipse2D` e `Double`, separati da un punto. Ciò significa che `Ellipse2D.Double` è una cosiddetta **classe interna** (*inner class*) di `Ellipse2D`. In realtà, costruendo e manipolando ellissi non occorre preoccuparsi del fatto che `Ellipse2D.Double` sia una classe interna: consideratela solo una classe con un nome lungo. Tuttavia, nell'enunciato `import` all'inizio del programma, dovete stare attenti a importare solamente la classe esterna:

```
import java.awt.geom.Ellipse2D;
```

Disegnare un'ellisse è facile: usate esattamente lo stesso metodo `draw` della classe `Graphics2D` che avete impiegato per disegnare rettangoli.

```
g2.draw(ellipse);
```

Per tracciare un cerchio, assegnate semplicemente lo stesso valore alla larghezza e all'altezza:

```
Ellipse2D.Double circle = new Ellipse2D.Double(x, y, diameter, diameter);
g2.draw(circle);
```

Notate che le coordinate  $(x, y)$  indicano il punto corrispondente al vertice superiore sinistro del rettangolo di delimitazione, non il centro del cerchio.

## 2.10.2 Segmenti

Per tracciare un segmento si usa un oggetto della classe `Line2D.Double`, nel cui costruttore si specificano i due punti terminali del segmento stesso. Si può fare in due modi: potete semplicemente indicare le coordinate  $x$  e  $y$  di entrambi gli estremi

```
Line2D.Double segment = new Line2D.Double(x1, y1, x2, y2);
```

oppure potete specificare ciascun estremo mediante un oggetto della classe `Point2D.Double`:

```
Point2D.Double from = new Point2D.Double(x1, y1);
Point2D.Double to = new Point2D.Double(x2, y2);

Line2D.Double segment = new Line2D.Double(from, to);
```

Il secondo metodo è più orientato agli oggetti e spesso è anche più utile, specialmente se gli oggetti che rappresentano i singoli punti si utilizzano nuovamente in un'altra zona dello stesso disegno.

## 2.10.3 Scrivere testo

Il metodo `drawString` disegna una stringa a partire dal suo punto base.

Spesso in una figura occorre scrivere testo, ad esempio per etichettarne alcune porzioni. Per inserire una stringa in una finestra si usa il metodo `drawString` della classe `Graphics2D`.

**Figura 23**  
Il punto base  
e la linea base



Dovete specificare come parametri la stringa e le coordinate *x* e *y* del punto base (*basepoint*) del primo carattere della stringa stessa (osservate la Figura 23), come in questo esempio:

```
g2.drawString("Message", 50, 100);
```

#### 2.10.4 Colori

Quando iniziate a disegnare, tutti gli oggetti vengono tracciati usando un pennino nero. Per cambiarne il colore, dovete fornire un oggetto di tipo `Color`. Java usa il modello di colore RGB (sigla che sta per *Red-Green-Blue*, cioè rosso-verde-blu): questo significa che per specificare un colore si indica la quantità dei colori primari (rosso, verde e blu, appunto) che generano il colore desiderato. Le quantità sono espresse da numeri interi compresi tra 0 (colore primario non presente) e 255 (presente in quantità massima). Per esempio, l'enunciato seguente costruisce un oggetto di tipo `Color` mediante una quantità massima di rosso e di blu, in assenza di verde, dando luogo a un colore viola brillante chiamato “magenta”.

```
Color magenta = new Color(255, 0, 255);
```

Per comodità, un certo numero di colori è già predefinito nella classe `Color`: la Tabella 4 riporta questi colori predefiniti e i loro valori RGB. Per esempio, `Color.PINK` (rosa) è predefinito in modo da produrre lo stesso colore della definizione `new Color(255, 175, 175)`.

Per disegnare una forma con un colore diverso dal nero, occorre per prima cosa impostare il colore desiderato nell’oggetto di tipo `Graphics2D`, poi invocarne il metodo `draw`:

```
g2.setColor(Color.RED); // imposta il colore rosso
g2.draw(circle); // disegna in rosso
```

Se volete colorare l’interno della figura, usate il metodo `fill` invece del metodo `draw`. Per esempio, questo enunciato riempie l’interno del cerchio con il colore attivo nel contesto grafico:

```
g2.fill(circle);
```

Il programma seguente utilizza tutte le forme viste, creando il semplice disegno visibile nella Figura 24.

Quando impostate un nuovo colore  
nel contesto grafico, verrà usato  
nelle operazioni grafiche successive.

**Tabella 4**  
Colori predefiniti

Colore	Descrizione	Valore RGB
Color.BLACK	Nero	0, 0, 0
Color.BLUE	Blu	0, 0, 255
Color.CYAN	Azzurro ciano	0, 255, 255
Color.GRAY	Grigio	128, 128, 128
Color.DARK_GRAY	Grigio scuro	64, 64, 64
Color.LIGHT_GRAY	Grigio chiaro	192, 192, 192
Color.GREEN	Verde	0, 255, 0
Color.MAGENTA	Magenta	255, 0, 255
Color.ORANGE	Arancione	255, 200, 0
Color.PINK	Rosa	255, 175, 175
Color.RED	Rosso	255, 0, 0
Color.WHITE	Bianco	255, 255, 255
Color.YELLOW	Giallo	255, 255, 0

**Figura 24**  
Il viso di un alieno



**File FaceComponent.java**

```

1 import java.awt.Color;
2 import java.awt.Graphics;
3 import java.awt.Graphics2D;
4 import java.awt.Rectangle;
5 import java.awt.geom.Ellipse2D;
6 import java.awt.geom.Line2D;
7 import javax.swing.JComponent;
8
9 /*
10      Un componente che disegna il viso di un alieno.
11 */
12 public class FaceComponent extends JComponent
13 {
14     public void paintComponent(Graphics g)
15     {
16         // recupera il riferimento a Graphics2D
17         Graphics2D g2 = (Graphics2D) g;
18
19         // disegna la testa
20         Ellipse2D.Double head = new Ellipse2D.Double(5, 10, 100, 150);
21         g2.draw(head);

```

```

22
23     // disegna gli occhi
24     g2.setColor(Color.GREEN);
25     Rectangle eye = new Rectangle(25, 70, 15, 15);
26     g2.fill(eye);
27     eye.translate(50, 0);
28     g2.fill(eye);
29
30     // disegna la bocca
31     Line2D.Double mouth = new Line2D.Double(30, 110, 80, 110);
32     g2.setColor(Color.RED);
33     g2.draw(mouth);
34
35     // scrive il saluto
36     g2.setColor(Color.BLUE);
37     g2.drawString("Hello, World!", 5, 175);
38 }
39 }
```

### File FaceViewer.java

```

1 import javax.swing.JFrame;
2
3 public class FaceViewer
4 {
5     public static void main(String[] args)
6     {
7         JFrame frame = new JFrame();
8         frame.setSize(150, 250);
9         frame.setTitle("An Alien Face");
10        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11
12        FaceComponent component = new FaceComponent();
13        frame.add(component);
14
15        frame.setVisible(true);
16    }
17 }
```



### Auto-valutazione

45. Specificate le istruzioni necessarie per disegnare un cerchio avente raggio 25 e centro nel punto (100, 100).
46. Specificate le istruzioni necessarie per disegnare una lettera "V" mediante due segmenti.
47. Specificate le istruzioni necessarie per disegnare una stringa contenente la sola lettera "V".
48. Quali sono i valori RGB del colore Color.BLUE?
49. Come si disegna un quadrato giallo su uno sfondo rosso?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R.2.27, E2.19 e E2.20, al termine del capitolo.

## Riepilogo degli obiettivi di apprendimento

### Oggetti, classi e metodi

- Gli oggetti sono entità di un programma che si possono manipolare invocando metodi.
- Un metodo è una sequenza di istruzioni che accedono ai dati di un oggetto.
- Una classe descrive un insieme di oggetti aventi lo stesso comportamento.

### Dichiarazione di variabili e assegnazione

- Una variabile è una zona di memoria dotata di un nome.
- Quando si dichiara una variabile di solito se ne specifica anche un valore iniziale.
- Quando si dichiara una variabile si specifica anche il tipo dei suoi valori.
- Il tipo `int` si usa per numeri che non possono avere una parte frazionaria.
- Il tipo `double` si usa per i numeri in virgola mobile.
- I numeri si possono combinare con operatori aritmetici, come `+`, `-` e `*`.
- Per convenzione, i nomi delle variabili devono iniziare con una lettera minuscola.
- I commenti si usano per fornire spiegazioni ai lettori umani in merito al codice. Il compilatore ignora i commenti.
- Per modificare il valore di una variabile si usa l'operatore di assegnazione (`=`).
- Tutte le variabili devono essere inizializzate prima di essere utilizzate.
- L'operatore di assegnazione *non* esprime un'uguaglianza matematica.

### Valori restituiti dai metodi e loro parametri

- L'interfaccia pubblica di una classe specifica cosa si può fare con i suoi oggetti, mentre l'implementazione nascosta descrive come si portano a termine tali azioni.
- Un argomento o parametro è un dato fornito durante l'invocazione di un metodo.
- Il valore restituito da un metodo è un risultato calcolato dal metodo stesso.

### Utilizzo di costruttori per costruire nuovi oggetti

- Per costruire nuovi oggetti si usa l'operatore `new`, seguito dal nome di una classe e da parametri opportuni.

### Metodi d'accesso e metodi modificatori

- Un metodo d'accesso non modifica i dati interni al suo parametro implicito, mentre un metodo modificatore lo fa.

### La documentazione API: descrizione di pacchetti e di metodi

- La documentazione API (Application Programming Interface) elenca le classi e i metodi della libreria di Java.
- Le classi Java sono raggruppate in pacchetti. Per utilizzare classi definite in pacchetti diversi da quello standard si usa l'enunciato `import`.

### Programmi che collaudano il comportamento dei metodi

- Un programma di collaudo verifica che i metodi si comportino come previsto.
- La determinazione a priori dei risultati attesi è un'attività essenziale nel collaudo.

### Oggetti e riferimenti

- Un riferimento a un oggetto descrive la posizione dell'oggetto in memoria.
- Più variabili oggetto possono fare riferimento al medesimo oggetto.
- Le variabili numeriche memorizzano numeri, mentre le variabili oggetto memorizzano riferimenti.

### Programmi che visualizzano finestre

- Per visualizzare una finestra con cornice (*frame*) si costruisce un oggetto di tipo `JFrame`, se ne impostano le dimensioni e lo si rende visibile.
- Per visualizzare qualcosa in un frame, occorre dichiarare una classe che estenda la classe `JComponent`.
- Inserite le istruzioni di disegno all'interno del metodo `paintComponent`, che viene invocato ogni volta che il componente deve essere ridisegnato.
- Per ottenere l'oggetto di tipo `Graphics2D` a partire dal parametro di tipo `Graphics` del metodo `paintComponent` bisogna usare un *cast*.

### Utilizzo dell'API di Java per disegnare semplici figure

- `Ellipse2D.Double` e `Line2D.Double` sono classi che descrivono forme grafiche.
- Il metodo `drawString` disegna una stringa a partire dal suo punto base.
- Quando impostate un nuovo colore nel contesto grafico, verrà usato nelle operazioni grafiche successive.

## Elementi di libreria presentati in questo capitolo

<code>java.awt.Color</code>	<code>java.awt.Rectangle</code>
<code>java.awt.Component</code>	<code>getHeight</code>
<code>getHeight</code>	<code>getWidth</code>
<code>getWidth</code>	<code>getX</code>
<code>setSize</code>	<code>getY</code>
<code>setVisible</code>	<code>setSize</code>
<code>java.awt.Frame</code>	<code>translate</code>
<code>setTitle</code>	<code>java.lang.String</code>
<code>java.awt.geom.Ellipse2D.Double</code>	<code>length</code>
<code>java.awt.geom.Line2D.Double</code>	<code>replace</code>
<code>java.awt.geom.Point2D.Double</code>	<code>toLowerCase</code>
<code>java.awt.Graphics</code>	<code>toUpperCase</code>
<code>setColor</code>	<code>javax.swing.JComponent</code>
<code>java.awt.Graphics2D</code>	<code>paintComponent</code>
<code>draw</code>	<code>javax.swing.JFrame</code>
<code>drawString</code>	<code>setDefaultCloseOperation</code>
<code>fill</code>	

## Esercizi di riepilogo e approfondimento

- \* **R2.1.** Spiegate la differenza fra un oggetto e una classe.
- \* **R2.2.** Fornite tre esempi di oggetti che appartengono alla classe `String`. Fornite un esempio di oggetto che appartiene alla classe `PrintStream`. Fornite il nome di due metodi che appartengono alla classe `String` ma non alla classe `PrintStream`. Fornite il nome di un metodo della classe `PrintStream` che non appartiene alla classe `String`.
- \* **R2.3.** Cos'è l'*interfaccia pubblica* di una classe? In cosa differisce dalla sua *implementazione*?
- \* **R2.4.** Dichiarate e inizializzate variabili che memorizzino il prezzo e la descrizione di un articolo posto in vendita.
- \* **R2.5.** Che valore ha `mystery` dopo questa sequenza di enunciati?

```
int mystery = 1;
```

```
mystery = 1 - 2 * mystery;
mystery = mystery + 1;
```

- \* **R2.6.** Che errore c'è in questa sequenza di enunciati?

```
int mystery = 1;
mystery = mystery + 1;
int mystery = 1 - 2 * mystery;
```

- \*\* **R2.7.** Illustrate il significato del simbolo `=` in Java e in matematica.

- \*\* **R2.8.** Fornite un esempio di metodo che riceve un argomento di tipo `int`. Fornite un esempio di metodo che restituisce un valore di tipo `int`. Fate le stesse cose con il tipo `String`.

- \*\* **R2.9.** Scrivete enunciati Java che inizializzino la variabile stringa `message` con "Hello", modificandone poi il contenuto in "HELLO". Usate il metodo `toUpperCase`.

- \*\* **R2.10.** Scrivete enunciati Java che inizializzino la variabile stringa `message` con "Hello", modificandone poi il contenuto in "hello". Usate il metodo `replace`.

- \*\* **R2.11.** Scrivete enunciati Java che inizializzino la variabile stringa `message` con un messaggio come "Hello, World", eliminandovi poi i segni di punteggiatura mediante ripetute invocazioni del metodo `replace`.

- \* **R2.12.** Spiegate la differenza fra un oggetto e una variabile oggetto.

- \*\* **R2.13.** Scrivete enunciati Java per costruire un *oggetto* della classe `Rectangle` e per dichiarare una *variabile oggetto* della stessa classe.

- \*\* **R2.14.** Scrivete enunciati Java per costruire gli oggetti così descritti:

- Un rettangolo con il centro nel punto di coordinate (100, 100) e con la lunghezza di tutti i lati uguale a 50.
- La stringa "Hello, Dave!".

Create soltanto gli oggetti, non le variabili oggetto.

- \*\* **R2.15.** Ripetete l'esercizio precedente, definendo ora variabili oggetto che vengano inizializzate con gli oggetti appena costruiti.

- \*\* **R2.16.** Scrivete un enunciato Java che inizializzi una variabile `square` con un rettangolo il cui vertice superiore sinistro abbia coordinate (10, 20) e i cui lati abbiano tutti lunghezza 40. Scrivete, poi, un enunciato che sostituisca il contenuto di `square` con un rettangolo avente le stesse dimensioni ma vertice superiore sinistro posizionato nel punto (20, 20).

- \*\* **R2.17.** Scrivete enunciati Java che inizializzino due variabili, `square1` e `square2`, in modo che facciano riferimento al medesimo quadrato, i cui lati abbiano tutti lunghezza 40 e il cui centro sia posizionato nel punto di coordinate (20, 20).

- \*\* **R2.18.** Identificate gli errori presenti negli enunciati seguenti:

- `Rectangle r = (5, 10, 15, 20);`
- `double width = Rectangle(5, 10, 15, 20).getWidth();`
- `Rectangle r;`  
`r.translate(15, 25);`
- `r = new Rectangle();`  
`r.translate("far, far away!");`

- \* **R2.19.** Indicate due metodi d'accesso e due metodi modificatori della classe `Rectangle`.

\*\* R2.20. Consultate la documentazione API per scoprire metodi per

- concatenare due stringhe, cioè costruire una stringa costituita dalla prima stringa, seguita dalla seconda;
- eliminare da una stringa gli eventuali spazi iniziali e finali;
- convertire un rettangolo in una stringa;
- individuare il più piccolo rettangolo che contiene due rettangoli dati;
- restituire un numero casuale in virgola mobile.

Per la soluzione di ciascun problema, citate il nome del metodo identificato e la classe in cui è definito, il suo valore restituito e i tipi dei suoi parametri.

\* R2.21. Spiegate la differenza fra un oggetto e un riferimento a un oggetto.

\* R2.22 (grafica). Spiegate la differenza fra un'applicazione grafica e un'applicazione per console.

\*\* R2.23 (grafica). Chi invoca il metodo `paintComponent` di un componente grafico? E quando viene invocato?

\*\* R2.24 (grafica). Perché il parametro del metodo `paintComponent` è di tipo `Graphics`, anziché `Graphics2D`?

\*\* R2.25 (grafica). A cosa serve un contesto grafico?

\*\* R2.26 (grafica). Perché nei programmi grafici usiamo classi separate per i componenti e per la loro visualizzazione?

\* R2.27 (grafica). In che modo si specifica il colore del testo?

## Esercizi di programmazione

---

\* E2.1 (collaudo). Scrivete il programma `AreaTester` che costruisca un oggetto di tipo `Rectangle`, ne calcoli l'area e la visualizzi. Usate i metodi `getWidth` e `getHeight` e visualizzate anche il valore previsto.

\* E2.2 (collaudo). Scrivete il programma `PerimeterTester` che costruisca un oggetto di tipo `Rectangle`, ne calcoli il perimetro e lo visualizzi. Usate i metodi `getWidth` e `getHeight` e visualizzate anche il valore previsto.

\*\* E2.3. Scrivete un programma che inizializzi una stringa al valore "Mississippi", per poi sostituirvi tutte le lettere "i" con "ii", visualizzando infine la lunghezza della stringa ottenuta. In tale stringa, poi, si sostituiscono tutte le stringhe "ss" con "s", visualizzando di nuovo la lunghezza della stringa così ottenuta.

\* E2.4. Scrivete un programma che costruisca un rettangolo avente area 42 e un rettangolo avente perimetro 42, visualizzando larghezza e altezza di entrambi.

\*\* E2.5 (collaudo). Consultate la documentazione API della classe `Rectangle` e individuate il metodo:

```
void add(int newx, int newy)
```

Leggete la documentazione del metodo, poi determinate il risultato prodotto dagli enunciati seguenti:

```
Rectangle box = new Rectangle(5, 10, 20, 30);
box.add(0, 0);
```

Scrivete il programma `AddTester` che visualizzi i valori previsti ed effettivi della posizione, della larghezza e dell'altezza del rettangolo `box` dopo l'invocazione di `add`.

- ★★ **E2.6 (collaudo).** Scrivete il programma `ReplaceTester` che codifichi una stringa sostituendo, mediante il metodo `replace`, tutte le lettere "i" con "1" e tutte le lettere "s" con "\$". Fate vedere che riuscite a codificare correttamente la stringa "Mississippi", visualizzando il risultato prodotto e quello previsto.
- ★★ **E2.7.** Scrivete il programma `HollePrinter` che scambi tra loro le lettere "e" e "o" in una stringa, usando ripetutamente il metodo `replace`. Fate vedere che la stringa "Hello, World!" si trasforma in "Holle, Werld!".
- \* **E2.8 (collaudo).** La classe `StringBuilder` ha un metodo `reverse` che inverte una stringa. Nella classe `ReverseTester`, costruite un oggetto di tipo `StringBuilder` usando come parametro di costruzione una stringa data (come "desserts"), poi invocate il metodo `reverse` seguito dal metodo `toString` e visualizzate il risultato prodotto, oltre a quello previsto.
- ★★ **E2.9.** Nella libreria Java, un colore viene specificato mediante le sue tre componenti (rosso, verde e blu), con valori numerici compresi tra 0 e 255, come visto nella Tabella 4. Scrivete il programma `BrighterDemo` che costruisca un oggetto di tipo `Color` con i valori di rosso, verde e blu rispettivamente uguali a 50, 100 e 150. Successivamente, applicate il metodo `brighter` della classe `Color` e visualizzate i valori delle tre componenti del colore risultante (non vedrete veramente il colore: per visualizzare i colori, svolgete il prossimo esercizio).
- ★★ **E2.10 (grafica).** Risolvete nuovamente l'esercizio precedente, ma inserite il vostro codice nella classe qui riportata: in questo modo il colore verrà visualizzato veramente.

```
import java.awt.Color;
import javax.swing.JFrame;

public class BrighterDemo
{
    public static void main(String[] args)
    {
        JFrame frame = new JFrame();
        frame.setSize(200, 200);
        Color myColor = ...;
        frame.getContentPane().setBackground(myColor);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

- ★★ **E2.11.** Risolvete nuovamente l'esercizio E2.9, ma applicate due volte il metodo `darker` della classe `Color` al colore predefinito `Color.RED`. Chiamate il vostro programma `DarkerDemo`.
- ★★ **E2.12.** La classe `Random` realizza un *generatore di numeri casuali*, cioè genera sequenze di numeri che appaiono essere casuali. Per generare numeri interi casuali bisogna costruire un oggetto della classe `Random`, a cui applicare poi il metodo `nextInt`. Ad esempio, l'invocazione `generator.nextInt(6)` fornisce un numero casuale compreso tra 0 e 5. Scrivete il programma `DieSimulator` che usi la classe `Random` per simulare il lancio di un dado, visualizzando un numero casuale compreso tra 1 e 6 ogni volta che viene eseguito.
- ★★ **E2.13.** Scrivete il programma `RandomPrice` che visualizzi un prezzo casuale compreso tra \$10.00 e \$19.95 ogni volta che viene eseguito.

- \*\* **E2.14 (collaudo).** Consultate la documentazione API della classe `Point` e scoprite come si costruisce un oggetto di tipo `Point`. Nel programma `PointTester`, costruite due punti aventi coordinate  $(3, 4)$  e  $(-3, -4)$ , poi calcolate la loro distanza usando il metodo `distance`. Visualizzate la distanza così calcolata, oltre a quella prevista (eventualmente facendo una figura su un foglio di carta per calcolarla).
- \* **E2.15.** Usando la classe `Day` vista nella sezione Esempi completi 2.1, scrivete il programma `DayTester` che costruisca un oggetto di tipo `Day` rappresentante la data odierna, aggiungetevi dieci giorni e calcolate la differenza tra le due date, visualizzando il risultato prodotto insieme a quello previsto.
- \*\* **E2.16.** Usando la classe `Picture` vista nella sezione Esempi completi 2.2, scrivete il programma `HalfSizePicture` che carichi una figura e la visualizzi con una dimensione pari alla metà di quella originale, centrata all'interno della finestra.
- \*\* **E2.17.** Usando la classe `Picture` vista nella sezione Esempi completi 2.2, scrivete il programma `DoubleSizePicture` che carichi una figura e la visualizzi con una dimensione pari al doppio di quella originale, centrata all'interno della finestra.
- \*\* **E2.18 (grafica).** Scrivete un programma grafico che disegni due quadrati aventi il medesimo centro. Progettate la classe `TwoSquareViewer` e la classe `TwoSquareComponent`.
- \*\* **E2.19 (grafica).** Scrivete un programma grafico che disegni due quadrati con l'intera superficie colorata, uno rosa e uno viola. Per uno dei quadrati usate uno dei colori predefiniti, mentre per l'altro usate un colore personalizzato. Progettate la classe `TwoSquareViewer` e la classe `TwoSquareComponent`.
- \*\* **E2.20 (grafica).** Scrivete un programma grafico che disegni il vostro nome in rosso, all'interno di un rettangolo blu. Progettate la classe `NameViewer` e la classe `NameComponent`.

Sul sito web dedicato al libro si trova una raccolta di progetti di programmazione più complessi.



# 3

## Realizzare classi



### Obiettivi del capitolo

- Acquisire familiarità con il procedimento di implementazione di classi
- Essere in grado di realizzare e collaudare semplici metodi
- Capire a cosa servono i costruttori e come si usano
- Capire come si accede a variabili di esemplare e variabili locali
- Saper scrivere commenti per la documentazione
- Realizzare classi per disegnare forme grafiche

In questo capitolo imparerete a progettare e a realizzare vostre classi. Quando si progetta una classe, bisogna decidere quale sia la sua interfaccia pubblica, cioè quali siano i metodi che i programmati potranno utilizzare per manipolare oggetti di quella classe, dopodiché occorre realizzare tali metodi. Questa seconda fase richiede l'identificazione di una rappresentazione dei dati necessari al funzionamento degli oggetti, per poi scrivere le istruzioni di ciascun metodo. Infine, è necessario documentare gli sforzi compiuti, in modo che altri programmati possano comprendere e utilizzare ciò che avete prodotto, e occorre fornire strumenti di collaudo, per dimostrare che la vostra classe funziona correttamente.

## 3.1 Variabili di esemplare e encapsulamento

Nel Capitolo 1 avete visto come si usano oggetti di classi esistenti, ora inizierete a realizzare vostre classi. Partiamo da un esempio molto semplice, che vi fa vedere come gli oggetti memorizzano i propri dati e come i metodi accedono ai dati degli oggetti stessi.

Il nostro primo esempio riguarda una classe che rappresenta un *conta-persone* (“tally counter”), un dispositivo meccanico, visibile in Figura 1, usato per il conteggio di persone (ad esempio, per vedere quante persone sono presenti a un concerto o si trovano a bordo di un autobus).

**Figura 1**  
Un conta-persone



### 3.1.1 Variabili di esemplare

Ogni volta che l'operatore preme il pulsante del conta-persone, il valore del conteggio viene incrementato di un'unità: modelliamo questa operazione mediante il metodo `click` di una classe, che chiamiamo `Counter`. Un conta-persone reale ha un visualizzatore che mostra sempre il valore del conteggio: nella nostra simulazione useremo, invece, il metodo `getValue` per ispezionare il valore di conteggio, come in questo esempio:

```
Counter tally = new Counter();
tally.click();
tally.click();
int result = tally.getValue(); // assegna a result il valore 2
```

Per realizzare la classe `Counter`, dobbiamo decidere quali dati vengano memorizzati all'interno di ciascun oggetto contatore. In questo esempio, così semplice, tale indagine è davvero elementare: ogni contatore ha la necessità di usare una variabile per tenere traccia del numero di avanzamenti del conteggio che sono stati richiesti mediante il pulsante.

Un oggetto memorizza i propri dati all'interno di **variabili di esemplare** (o di istanza, *instance variable*). Un *esemplare* di una classe è quello che finora abbiamo chiamato “un oggetto della classe”. Una variabile di esemplare è, quindi, una zona di memorizzazione presente in ciascun oggetto della classe.

La dichiarazione della classe specifica le sue variabili di esemplare:

```
public class Counter
{
    private int value;
    ...
}
```

Un oggetto usa variabili di esemplare per memorizzare i dati necessari per l'esecuzione dei propri metodi.

## Sintassi di Java

### 3.1 Dichiarazione di variabile di esemplare

#### Sintassi

```
public class NomeClasse
{
    private nomeDiTipo nomeVariabile;
    ...
}
```

#### Esempio

Le variabili di esemplare dovrebbero sempre essere private.

```
public class Counter
{
    private int value;
    ...
}
```

Ogni oggetto di questa classe ha una propria copia di questa variabile di esemplare, distinta dalle altre.

Tipo della variabile.

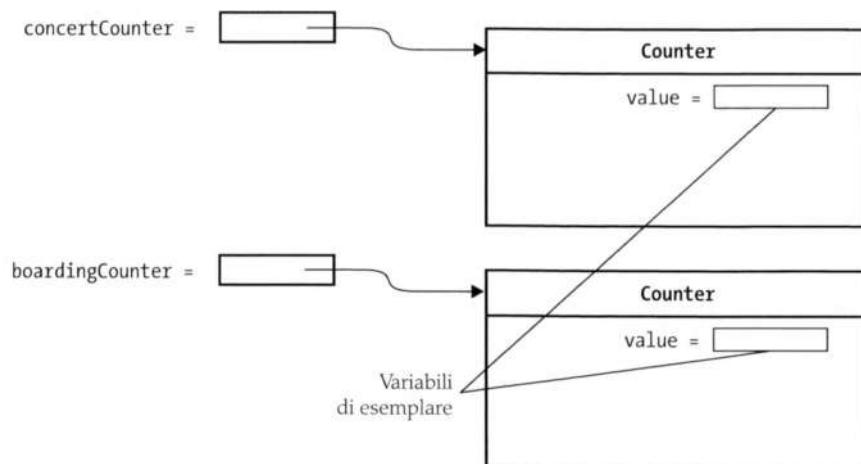
La dichiarazione di una variabile di esemplare è così composta:

- Una **modalità d'accesso** (`private`)
- Il **tipo** della variabile di esemplare (come `int`)
- Il nome della variabile di esemplare (come `value`)

Ciascun oggetto di una classe ha il proprio insieme di variabili di esemplare.

Ciascun oggetto di una classe ha il proprio insieme di variabili di esemplare. Ad esempio, se `concertCounter` e `boardingCounter` sono due oggetti della classe `Counter`, ognuno di essi ha la propria variabile `value`, come rappresentato in Figura 2. Come vedrete nel Paragrafo 3.3, nel momento in cui viene costruito un oggetto di tipo `Counter` alla sua variabile di esemplare `value` viene assegnato il valore 0.

**Figura 2**  
Variabili di esemplare



### 3.1.2 I metodi della classe Counter

In questo paragrafo ci occuperemo della realizzazione dei metodi della classe Counter.

Il metodo `click` incrementa di un'unità il valore del contatore. Nel Capitolo 2 avete visto la sintassi relativa all'intestazione di un metodo, mentre qui ci concentriamo sul suo corpo, all'interno delle parentesi graffe:

```
public void click()
{
    value = value + 1;
}
```

Osservate che il metodo `click` accede alla variabile di esemplare `value`: a *quale* variabile di esemplare accede? A quella che appartiene all'oggetto con cui si invoca il metodo. Considerate, ad esempio, questa invocazione:

```
concertCounter.click();
```

In questo caso viene incrementata la variabile `value` dell'oggetto `concertCounter`.

Il metodo `getValue` restituisce il valore attuale del conteggio:

```
public int getValue()
{
    return value;
}
```

L'enunciato `return` pone termine all'esecuzione del metodo e restituisce un risultato (il **valore restituito**, *return value*) a chi ha invocato il metodo stesso.

Alle variabili di esemplare private possono accedere soltanto i metodi appartenenti alla medesima classe.

Le variabili di esemplare sono generalmente dichiarate con modalità di accesso `private`: questo significa che vi si può accedere soltanto da metodi della *medesima classe* e da nessun altro metodo. Ad esempio, alla variabile `value` si può accedere dai metodi `click` e `getValue` della classe Counter, ma non da un metodo di una classe diversa: se quest'ultimo ha bisogno di manipolare il valore di conteggio di un contatore, deve usare i metodi della classe Counter.

### 3.1.3 Incapsulamento

Nel paragrafo precedente avete imparato a nascondere le variabili di esemplare, rendendole private. Ma perché un programmatore dovrebbe voler nascondere qualcosa?

L'occultamento delle informazioni (*information hiding*) non è una strategia adottata soltanto nella programmazione di calcolatori: viene usata in molte discipline dell'ingegneria. Considerate il termostato di casa: si tratta di un dispositivo che consente all'utente di selezionare alcune preferenze relative alla temperatura e, di conseguenza, controlla il funzionamento della caldaia e del condizionatore. Se chiedete al vostro idraulico di fiducia che cosa ci sia all'interno del termostato, probabilmente dirà che non ne sa nulla e che non gli interessa.

Il termostato è una "scatola nera" (*black box*), qualcosa che svolge quasi magicamente i propri compiti. Un idraulico non apre mai la scatola, perché contiene componenti che possono essere riparati soltanto dal costruttore. Più in generale, gli ingegneri usano il termine "scatola nera" per descrivere qualsiasi dispositivo i cui meccanismi interni

di funzionamento vengono tenuti nascosti. Notate, però, che una scatola nera non è completamente misteriosa: la sua interfaccia con il mondo esterno è ben definita. Ad esempio, l'idraulico sa come collegare il termostato alla caldaia e al condizionatore.

Il processo che nasconde i dettagli realizzativi, rendendo invece pubblica un'interfaccia, si chiama **incapsulamento** (*encapsulation*). In Java, è il costrutto sintattico `class` che realizza l'incapsulamento, mentre i metodi pubblici di una classe sono l'interfaccia attraverso cui viene manipolata l'implementazione privata.

Per quale motivo gli idraulici usano componenti prefabbricati, come i termostati e le caldaie? Queste "scatole nere" semplificano molto il lavoro degli idraulici: anticamente i muratori e gli idraulici dovevano sapere come costruire una caldaia usando malta e mattoni, ma oggi si recano semplicemente in un negozio di ferramenta, senza aver bisogno di sapere cosa si trova all'interno di ciò che acquistano.

Analogamente, un programmatore che usa una classe non è oberato da dettagli non strettamente necessari, come già sapete per esperienza diretta: nel Capitolo 2 avete usato classi che descrivono stringhe, flussi e finestre, senza preoccuparvi di come tali classi siano state realizzate.

L'incapsulamento aiuta anche a diagnosticare gli errori. Un programma di grandi dimensioni può essere costituito da centinaia di classi e migliaia di metodi, ma se viene identificato un errore relativo ai dati interni di un oggetto dovete analizzare i metodi di una sola classe. Infine, l'incapsulamento consente di modificare i dettagli realizzativi di una classe senza dover comunicare questa decisione ai programmatore che ne fanno uso.

Nel Capitolo 2 avete imparato a essere utenti di oggetti, creandoli, manipolandoli e usandoli per comporre programmi: trattandoli, cioè, come scatole nere. Il vostro ruolo, quindi, è stato sostanzialmente analogo a quello di un idraulico che installa un termostato.

In questo capitolo inizierete a progettare classi. In questi paragrafi, il vostro ruolo è analogo a quello del progettista di componenti per abitazioni, che progetta un termostato a partire da pulsanti, sensori e altri componenti elettronici. Apprenderete le tecniche di programmazione Java necessarie per consentire ai vostri oggetti di esibire il comportamento desiderato.

## File Counter.java

```

1  /**
2   * Questa classe definisce un conta-persone.
3  */
4  public class Counter
5  {
6      private int value;
7
8      /**
9       * Ispeziona il valore del conteggio di questo contatore.
10      @return il valore del conteggio
11     */
12     public int getValue()
13     {
14         return value;
15     }
16
17     /**
18      Incrementa di un'unità il conteggio di questo contatore.
19     */

```

L'incapsulamento prevede di nascondere i dettagli realizzativi, fornendo metodi per l'accesso ai dati.

L'incapsulamento consente ai programmatori di usare una classe senza doverne conoscere i dettagli realizzativi.

L'incapsulamento agevola l'individuazione di errori e la modifica dei dettagli realizzativi di una classe.

```

20    public void click()
21    {
22        value = value + 1;
23    }
24
25    /**
26     * Azzera il conteggio di questo contatore.
27     */
28    public void reset()
29    {
30        value = 0;
31    }
32 }
```



### Auto-valutazione

- Scrivete il corpo di un metodo, `public void unclick()`, che annulli l'effetto della errata pressione del pulsante di un conta-persone.
- Immaginate che una classe `Clock` abbia le variabili di esemplare `private hours` e `minutes`. Da un vostro programma, come si può accedere a tali variabili?
- Considerate la classe `Counter`. Il valore di conteggio parte da zero e viene incrementato dal metodo `click`, per cui non dovrebbe mai essere negativo. Immaginate di scoprire, durante il collaudo, un valore negativo contenuto nella variabile `value`. Dove guardereste per trovare l'errore?
- Nei Capitoli 1 e 2 avete usato `System.out` come una scatola nera per visualizzare informazioni sullo schermo del computer. Chi ha progettato e realizzato `System.out`?
- Supponete di lavorare in un'azienda che produce software per elaborazioni finanziarie personali e che vi venga chiesto di progettare e realizzare una classe che rappresenti conti bancari. Quali saranno gli utenti della vostra classe?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R3.1, R3.3 e E3.1, al termine del capitolo.

## 3.2 Progettare l'interfaccia pubblica di una classe

In questo paragrafo analizzeremo il procedimento da seguire per specificare l'interfaccia pubblica di una classe. Immaginate di far parte di un gruppo di progettisti che sta lavorando a un software bancario, nel quale uno dei concetti chiave è il *conto bancario*: il vostro compito consiste nella progettazione e realizzazione di una classe, `BankAccount`, che possa essere utilizzata dagli altri programmati del gruppo di lavoro che debbano elaborare conti bancari. Di quali metodi bisogna dotarla? Quali informazioni bisogna fornire ai programmati che la useranno? Queste sono domande a cui dovete dare una risposta prima di iniziare a realizzare la classe.

### 3.2.1 Definire i metodi

Per realizzare una classe, occorre prima individuare quali metodi siano necessari.

Dovete capire bene quali caratteristiche di un conto bancario debbano essere realizzate: alcune sono essenziali (come la possibilità di fare versamenti), mentre altre sono meno importanti (come l'omaggio che un cliente a volte riceve in seguito all'apertura di un

nuovo conto). Decidere quali caratteristiche siano essenziali non è sempre cosa facile e ne ripareremo nel Capitolo 8. Per il momento, immaginiamo che un progettista esperto abbia deciso quali operazioni siano considerate irrinunciabili per un conto bancario:

- Versare denaro
- Prelevare denaro
- Conoscere il saldo attuale

In Java, le operazioni sugli oggetti vengono espresse mediante invocazioni di metodi: per identificare le specifiche corrette per le invocazioni di questi metodi, immaginate come un programmatore effettuerà le operazioni sui conti bancari. Ipotizziamo che la variabile `harrysChecking` contenga un riferimento a un oggetto di tipo `BankAccount`. Vogliamo che i metodi possano funzionare nel modo seguente:

```
harrysChecking.deposit(2240.59);
harrysChecking.withdraw(500);
double currentBalance = harrysChecking.getBalance();
```

I primi due sono metodi modificatori: modificano il saldo del conto bancario e non restituiscono alcun valore. Il terzo, invece, è un metodo d'accesso: restituisce un valore che può essere memorizzato in una variabile o passato come parametro a un altro metodo.

Come si può evincere da queste invocazioni usate come esempio, la classe `BankAccount` deve dichiarare tre metodi:

- `public void deposit(double amount)`
- `public void withdraw(double amount)`
- `public double getBalance()`

Ricordate, dal Capitolo 2, che `double` indica un tipo numerico in virgola mobile a doppia precisione, mentre `void` sta a significare che il metodo non restituisce alcun valore.

Qui abbiamo specificato solamente le *intestazioni* dei metodi. Quando dichiarate un metodo, ne dovete anche fornire il **corpo** (*body*), composto dagli enunciati che vengono eseguiti quando il metodo viene invocato:

```
public void deposit(double amount)
{
    corpo del metodo, che verrà riempito in seguito
}
```

Nel Paragrafo 3.3 scriveremo il corpo dei metodi.

Osservate che i metodi sono stati elencati come “pubblici” (usando la parola `public`), specificando così che possono essere invocati da qualsiasi altro metodo. A volte può essere utile avere metodi “privati” (`private`), che possono essere invocati soltanto da altri metodi della stessa classe.

Alcuni programmatori preferiscono inserire enunciati nel corpo fin da subito, in modo che il metodo possa essere compilato, anche se non nella sua forma completa e definitiva, come in questo esempio:

```
public double getBalance()
{
    // DA FARE: manca la vera implementazione
    return 0;
}
```

È una buona idea, soprattutto se scrivete i vostri programmi usando un ambiente di sviluppo, perché in questo modo non verranno generati avvertimenti relativi a “codice non corretto”.

### 3.2.2 Definire i costruttori

I costruttori impostano i valori iniziali per i dati degli oggetti.

Come avete visto nel Capitolo 2, i costruttori vengono usati per inizializzare gli oggetti. In Java, un **costruttore** è molto simile a un metodo, con due differenze rilevanti:

- Il nome di un costruttore è sempre uguale al nome della classe (ad esempio, `BankAccount`).
- I costruttori non definiscono un tipo per il “valore restituito” (nemmeno `void`).

Vogliamo poter costruire conti bancari con saldo iniziale pari a zero e conti bancari con un saldo iniziale assegnato.

Per questo motivo, specifichiamo due costruttori:

- `public BankAccount()`
- `public BankAccount(double initialBalance)`

che vengono usati in questo modo:

```
BankAccount harrysChecking = new BankAccount();
BankAccount momssSavings = new BankAccount(5000);
```

Il nome di un costruttore  
è sempre uguale al nome  
della classe.

Non preoccupatevi per il fatto che ci siano due costruttori con lo stesso nome: *tutti* i costruttori di una classe hanno lo stesso nome, che peraltro coincide con il nome della classe stessa. Il compilatore è in grado di distinguerli l’uno dall’altro perché richiedono parametri diversi: il primo costruttore non richiede alcun parametro, e viene chiamato **costruttore privo di parametri**, mentre il secondo richiede un parametro di tipo `double`.

Esattamente come un metodo, anche un costruttore ha un corpo, cioè una sequenza di enunciati che vengono eseguiti quando si costruisce un nuovo oggetto.

```
public BankAccount()
{
    corpo del costruttore, che verrà riempito in seguito
}
```

Gli enunciati presenti nel corpo del costruttore imposteranno i valori iniziali delle variabili di esemplare dell’oggetto che è in fase di costruzione, come vedrete nel Paragrafo 3.3.

Quando dichiarate una classe, inserite al suo interno le dichiarazioni di tutti i costruttori e di tutti i metodi, in questo modo:

```
public class BankAccount
{
    variabili di esemplare private, definite in seguito
```

```

// Costruttori
public BankAccount()
{
    corpo, che verrà riempito in seguito
}

public BankAccount(double initialBalance)
{
    corpo, che verrà riempito in seguito
}

// Metodi
public void deposit(double amount)
{
    corpo, che verrà riempito in seguito
}

public void withdraw(double amount)
{
    corpo, che verrà riempito in seguito
}

public double getBalance()
{
    corpo, che verrà riempito in seguito
}
}

```

I costruttori e i metodi pubblici di una classe costituiscono la sua **interfaccia pubblica** (*public interface*): sono le operazioni che qualsiasi programmatore può utilizzare per creare e manipolare oggetti di tipo BankAccount.

## Sintassi di Java

### 3.2 Dichiarazione di classe

#### Sintassi

```

modalitàDiAccesso class NomeClasse
{
    variabili di esemplare
    costruttori
    metodi
}

```

#### Esempio

```

public class Counter
{
    private int value; Realizzazione privata

    public Counter(int initialValue) { value = initialValue; }

    public void click() { value = value + 1; }

    public int getValue() { return value; }
}

```

**Interfaccia pubblica**

### 3.2.3 Usare l'interfaccia pubblica

La nostra classe `BankAccount` è semplice, ma consente ai programmati di eseguire tutte le principali operazioni che normalmente si effettuano con i conti bancari. Considerate, ad esempio, questa porzione di programma, scritta da un programmatore che usa la classe `BankAccount`: si tratta di enunciati che trasferiscono una somma di denaro da un conto bancario a un altro.

```
// trasferisce denaro da un conto a un altro
double transferAmount = 500;
momSavings.withdraw(transferAmount);
harrysChecking.deposit(transferAmount);
```

Ed ecco, invece, una porzione di programma che accredita gli interessi a un conto di risparmio:

```
double interestRate = 5; // 5% di interesse
double interestAmount = momSavings.getBalance() * interestRate / 100;
momSavings.deposit(interestAmount);
```

Come potete vedere, i programmati possono utilizzare oggetti della classe `BankAccount` per portare a termine compiti di un certo rilievo, senza sapere come gli oggetti di tipo `BankAccount` memorizzino i propri dati o come i metodi di `BankAccount` svolgano il proprio lavoro.

D'altra parte, è ovvio che, in qualità di progettisti della classe `BankAccount`, saremo chiamati a descriverne i dettagli interni. Lo faremo nel Paragrafo 3.3, ma, prima di quello, dobbiamo esaminare ancora un elemento: la *documentazione* dell'interfaccia pubblica, che sarà argomento del prossimo paragrafo.

### 3.2.4 Commentare l'interfaccia pubblica

Durante la realizzazione di classi e metodi, dovreste prendere l'abitudine di *commentare* esaurientemente il loro comportamento. In Java esiste un formato standard, molto utile per i **commenti di documentazione**. Se nelle vostre classi userete tale formato, potrete usare il programma `javadoc` per generare automaticamente un insieme elegante e chiaro di pagine HTML che le descrivono (si vedano i Suggerimenti per la programmazione 3.1 per una descrizione di questo programma di utilità).

Un commento di documentazione va inserito prima della dichiarazione della classe o del metodo che deve documentare e inizia con i caratteri `/**`: un delimitatore speciale per i commenti che devono essere interpretati dal programma di utilità `javadoc`. Di seguito si descrive per prima cosa lo *scopo* del metodo. Poi, per ciascun parametro del metodo, si inserisce una riga che inizia con il marcitore `@param`, seguito dal nome della variabile che riceve il parametro (e, per questo, chiamata **variabile parametro**) e da una sua breve spiegazione. Infine, deve essere presente una riga che inizia con `@return`, per descrivere il valore restituito. Non si indica il marcitore `@param` nei metodi che non hanno parametri, né il marcitore `@return` nei metodi il cui tipo restituito è `void`.

Il programma `javadoc` copia la *prima* frase di ciascun commento in una tabella riassuntiva all'interno della documentazione HTML, per cui è bene scrivere tale frase

Per descrivere le classi e i metodi pubblici dei programmi si usano i commenti di documentazione.

con cura, possibilmente iniziando con una lettera maiuscola e terminando con un punto. Non è necessario che sia una frase completa dal punto di vista grammaticale, ma dovrebbe mantenere un significato compiuto anche quando viene estratta dal commento e visualizzata in un riassunto.

Ecco due tipici esempi:

```
/**
 * Preleva denaro dal conto bancario.
 * @param amount l'importo da prelevare
 */
public void withdraw(double amount)
{
    implementazione (completata in seguito)
}

/**
 * Ispeziona il valore attuale del saldo del conto bancario.
 * @return il saldo attuale
 */
public double getBalance()
{
    implementazione (completata in seguito)
}
```

I commenti che avete appena visto descrivono singoli *metodi*, ma è bene fornire un breve commento anche per ciascuna *classe*, per illustrarne lo scopo, subito prima della definizione della classe stessa. La sintassi per il commento di documentazione di una classe è molto semplice:

```
/**
 * Un conto bancario ha un saldo che può essere
 * modificato mediante versamenti e prelievi.
 */
public class BankAccount
{
    ...
}
```

La vostra prima reazione potrebbe essere: “Ma devo proprio scrivere tutte queste cose?” Questi commenti sembrano molto ripetitivi, ma dovreste comunque trovare il tempo per scriverli, anche se a volte sembra una cosa noiosa.

È sempre una buona idea scrivere il commento di un metodo *prima* di scriverne il codice, perché ciò costituisce una eccellente verifica della reale comprensione di quello che si sta per programmare: se non siete in grado di spiegare cosa faccia un metodo o una classe, non siete pronti per scriverne il codice.

Cosa fare con i metodi molto semplici? Capita spesso di perdere più tempo a pensare se un commento sia troppo banale perché valga la pena scriverlo, piuttosto che scriverlo e basta. Nella realtà, i metodi molto semplici sono rari: avere un metodo banale con un commento inutile non è pericoloso, mentre un metodo complicato senza commento può veramente creare problemi nella futura manutenzione del programma. Secondo lo

Scrivete commenti di documentazione  
per ogni classe, ogni metodo,  
ogni parametro e ogni valore  
restituito.

stile standard per la documentazione Java, *ogni* classe, *ogni* metodo, *ogni* parametro e *ogni* valore restituito dovrebbero essere commentati.

Il programma javadoc organizzerà i vostri commenti in un insieme chiaro ed elegante di documenti che si possono visualizzare con un browser Web, facendo buon uso di quelle frasi apparentemente ripetitive. La prima frase di ogni commento viene inserita in una *tabella riassuntiva* di tutti i metodi della classe (come in Figura 3), mentre i commenti di tipo @param e @return vanno a comporre la descrizione dettagliata di ciascun metodo (come in Figura 4). Se qualcuno di tali commenti viene omesso, il programma javadoc genera documenti con strani spazi vuoti.

Questo formato della documentazione vi dovrebbe risultare familiare: i programmatori che realizzano la libreria di Java usano regolarmente javadoc e anch'essi documentano *ogni* classe, *ogni* metodo, *ogni* parametro e *ogni* valore restituito; poi, usano javadoc per generare la documentazione in formato HTML.

**Figura 3**

Un riassunto dei metodi generato da javadoc

Modifier and Type	Method and Description
void	deposit(double amount) Deposits money into the bank account.
double	getBalance() Gets the current balance of the bank account.
void	withdraw(double amount) Withdraws money from the bank account.

**Figura 4**

La documentazione dettagliata di un metodo generata da javadoc

**Method Detail**

**deposit**

public void deposit(double amount)

Deposits money into the bank account.

**Parameters:**

amount - the amount to deposit



## Auto-valutazione

6. Come è possibile *svuotare* il conto bancario `harrysChecking` usando i metodi dell'interfaccia pubblica della classe?
7. Cosa c'è di sbagliato in questa sequenza di enunciati?  
`BankAccount harrysChecking = new BankAccount(10000);`

- ```
System.out.println(harrysChecking.withdraw(500));
```
8. Supponete di voler realizzare una più potente astrazione di conto bancario che tenga traccia di un *numero di conto*, oltre al saldo. Come modifichereste l'interfaccia pubblica per gestire questo miglioramento?
9. Supponete di aver migliorato la classe `BankAccount` in modo che a ciascun conto sia associato un numero di conto (*account number*). Scrivete un commento per la documentazione del seguente costruttore:
- ```
public BankAccount(int accountNumber, double initialBalance)
```
10. Perché il seguente commento per la documentazione non è adeguato?
- ```
/**  
 * Ogni conto ha un numero di conto.  
 * @return il numero di conto di questo conto  
 */  
public int getAccountNumber()
```

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R3.7, R3.8 e R3.9, al termine del capitolo.



## Errori comuni 3.1

---

### Dichiarare un costruttore void

Quando dichiarate un costruttore, non usate la parola riservata `void`:

```
public void BankAccount() // Errore: non usate void!
```

Questo non dichiarerebbe un costruttore, bensì un metodo con “tipo restituito” `void`, cioè senza alcun valore restituito. Sfortunatamente, per il compilatore Java questo non è un errore di sintassi.



## Suggerimenti per la programmazione 3.1

---

### Il programma di utilità javadoc

Inserite sempre i commenti di documentazione nel vostro codice, indipendentemente dal fatto che usiate `javadoc` per produrre la documentazione HTML. Dato, poi, che molte persone ritengono efficace la documentazione HTML, è utile imparare come si esegue `javadoc`. Alcuni ambienti di programmazione (come BlueJ) sono in grado di eseguire `javadoc` autonomamente; in alternativa, potete invocarlo in questo modo:

```
javadoc MyClass.java
```

oppure, se volete creare la documentazione per più file Java:

```
javadoc *.java
```

Il programma `javadoc` produce file in formato HTML (come, ad esempio, `MyClass.html`) che potete esaminare mediante un browser. Se conoscete il linguaggio HTML, potete

anche incorporare marcatori HTML nei commenti all'interno del file sorgente Java, per specificare font o per aggiungere immagini. Cosa forse ancora più importante, javadoc aggiunge automaticamente *collegamenti ipertestuali* ad altre classi e metodi citati nella documentazione.

Potete addirittura eseguire javadoc prima di realizzare i metodi, lasciando semplicemente vuoti tutti i corpi dei metodi stessi. Non eseguite il compilatore, che segnalerebbe la mancanza degli eventuali valori da restituire: eseguite soltanto javadoc sul vostro file per generare la documentazione dell'interfaccia pubblica che state per realizzare.

Lo strumento javadoc è magnifico, perché fa una cosa veramente corretta: vi permette di scrivere la documentazione insieme con il codice. In questo modo, quando aggiornate il programma, potete vedere immediatamente quale documentazione bisogna conseguentemente aggiornare: quindi, si spera che la aggiorniate senza esitare. Al termine, eseguite nuovamente javadoc per ottenere una nuova pagina HTML, perfettamente impaginata e aggiornata.

## 3.3 Realizzare la classe

Ora che abbiamo ben compreso le specifiche dell'interfaccia pubblica della classe BankAccount, passiamo alla sua realizzazione.

### 3.3.1 Definire le variabili di esemplare

L'implementazione privata di una classe comprende le variabili di esemplare e il corpo di costruttori e metodi.

Dobbiamo, per prima cosa, determinare quali siano i dati contenuti in ciascun oggetto che rappresenti un conto bancario. Nel caso semplice che abbiamo analizzato, ogni conto bancario deve memorizzare al proprio interno un solo valore: il suo saldo (un conto bancario più articolato potrebbe aver bisogno di memorizzare più dati, tra cui, ad esempio, il numero di conto, il tasso di interesse, la data in cui verrà emesso il prossimo estratto conto, ecc.).

```
public class BankAccount
{
    private double balance;
    // seguiranno metodi e costruttori
    ...
}
```

In generale, può non essere facile individuare un valido insieme di variabili di esemplare. Chiedetevi cosa debba necessariamente ricordare un oggetto per poter svolgere i compiti specifici di ciascuno dei suoi metodi.

### 3.3.2 Definire i costruttori

Un **costruttore** ha un compito veramente semplice: assegnare un valore iniziale alle variabili di esemplare di un oggetto.

Ricordate che abbiamo progettato la classe BankAccount in modo che abbia due costruttori, il primo dei quali assegna semplicemente il valore zero alla variabile che rappresenta il saldo:

```
public BankAccount()
{
    balance = 0;
}
```

Il secondo costruttore assegna al saldo il valore ricevuto come parametro di costruzione:

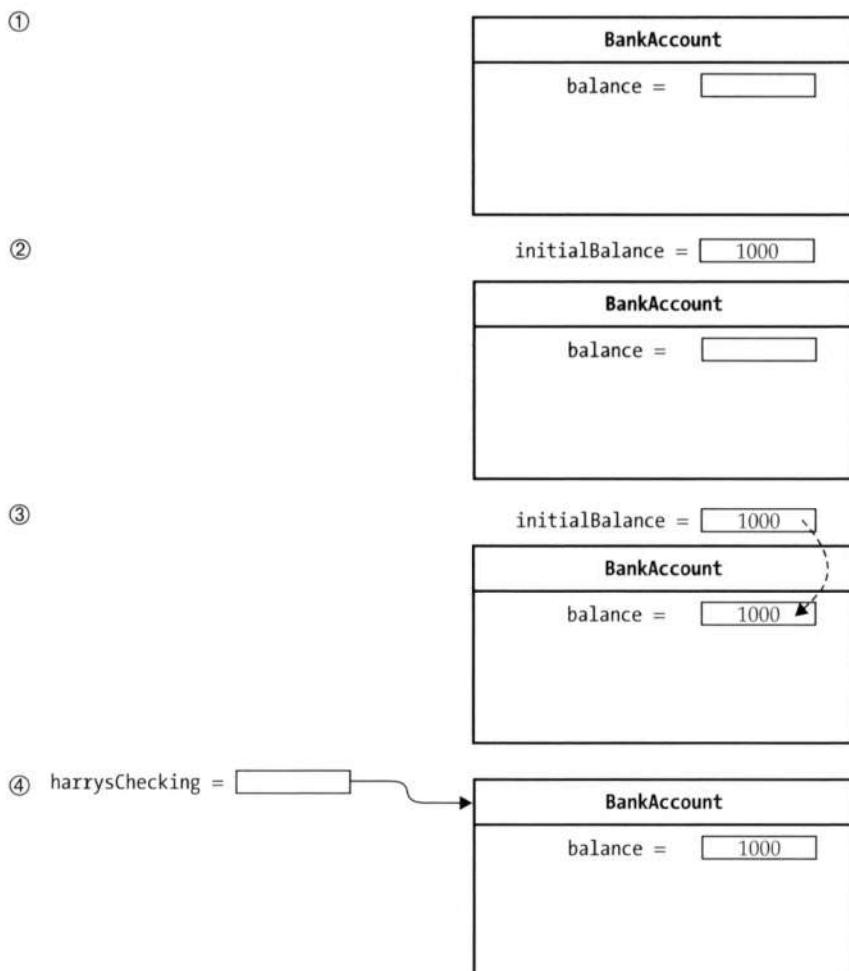
```
public BankAccount(double initialBalance)
{
    balance = initialBalance;
}
```

Per vedere come funzionano questi costruttori, seguiamo passo dopo passo cosa avviene durante una loro invocazione, come questa:

```
BankAccount harrysChecking = new BankAccount(1000);
```

**Figura 5** ①

Funzionamento  
di un costruttore



Le singole azioni elementari che avvengono durante l'esecuzione dell'enunciato sono queste:

- Creazione di un nuovo oggetto di tipo `BankAccount`. ①
- Invocazione del secondo costruttore (perché è stato fornito un parametro di costruzione).
- Assegnazione del valore 1000 alla variabile parametro `initialBalance`. ②
- Assegnazione del valore di `initialBalance` alla variabile di esemplare `balance` dell'oggetto appena creato. ③
- Restituzione, come valore dell'espressione `new`, di un riferimento: la posizione in memoria dell'oggetto appena creato.
- Memorizzazione nella variabile `harrysChecking` del riferimento all'oggetto. ④

In generale, quando si realizzano i costruttori, bisogna accertarsi che ciascuno di essi inizializzi tutte le variabili di esemplare, facendo uso delle variabili parametro, quando sono presenti (alcuni suggerimenti sono forniti nella sezione Errori comuni 3.2).

### 3.3.3 Definire i metodi

In questo paragrafo completeremo l'implementazione dei metodi della classe `BankAccount`.

Quando realizzate un metodo, chiedetevi innanzitutto se si tratti di un metodo d'accesso o di un metodo modificatore. Un metodo modificatore dovrà aggiornare in qualche modo qualche variabile d'esemplare, mentre un metodo d'accesso dovrà ispezionare o calcolare un valore, per poi restituirlo.

Ecco il metodo `deposit`. È un metodo modificatore, che aggiorna il saldo:

```
public void deposit(double amount)
{
    balance = balance + amount;
}
```

Il metodo `withdraw` è molto simile al metodo `deposit`:

```
public void withdraw(double amount)
{
    balance = balance - amount;
}
```

Manca un solo metodo: `getBalance`. Diversamente dai metodi `deposit` e `withdraw`, che modificano le variabili di esemplare dell'oggetto con cui vengono invocati, il metodo `getBalance` restituisce un valore:

```
public double getBalance()
{
    return balance;
}
```

Abbiamo così completato l'implementazione della classe `BankAccount`, di cui presentiamo ora il codice completo. Ci rimane soltanto un piccolo passo da compiere: verificare che la classe funzioni correttamente, cosa che faremo nel prossimo paragrafo.

**Tabella 1**  
Implementazione di classi

| Esempio                                      | Commento                                                                                                                                  |
|----------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| public class BankAccount { . . . }           | L'inizio della dichiarazione di una classe: tra le parentesi graffe troveranno posto le variabili di esemplare, i metodi e i costruttori. |
| private double balance;                      | Una variabile di esemplare di tipo <code>double</code> . Tutte le variabili di esemplare dovrebbero essere dichiarate private.            |
| public double getBalance() { . . . }         | La dichiarazione di un metodo: il corpo del metodo si troverà all'interno delle parentesi graffe.                                         |
| . . . { return balance; }                    | Il corpo del metodo <code>getBalance</code> : l'enunciato <code>return</code> restituisce un valore al metodo invocante.                  |
| public void deposit(double amount) { . . . } | Un metodo con una variabile parametro ( <code>amount</code> ): essendo dichiarato <code>void</code> , non restituisce alcunché.           |
| . . . { balance = balance + amount; }        | Il corpo del metodo <code>deposit</code> : non ha un enunciato <code>return</code> .                                                      |
| public BankAccount() { . . . }               | La dichiarazione di un costruttore: ha lo stesso nome della classe e non ha il tipo del valore restituito.                                |
| . . . { balance = 0; }                       | Il corpo del costruttore: dovrebbe inizializzare le variabili di esemplare.                                                               |

### File BankAccount.java

```

1  /**
2   * Un conto bancario ha un saldo che può essere
3   * modificato mediante versamenti e prelievi.
4  */
5  public class BankAccount
6  {
7      private double balance;
8
9      /**
10      * Costruisce un conto bancario con saldo uguale a zero.
11     */
12     public BankAccount()
13     {
14         balance = 0;
15     }
16
17     /**
18      * Costruisce un conto bancario con saldo assegnato.
19      * @param initialBalance il saldo iniziale
20     */
21     public BankAccount(double initialBalance)
22     {
23         balance = initialBalance;
24     }
25
26     /**
27      * Versa denaro nel conto bancario.
28      * @param amount l'importo da versare
29     */
30     public void deposit(double amount)

```

```

31  {
32      balance = balance + amount;
33  }
34
35 /**
36     Preleva denaro dal conto bancario.
37     @param amount l'importo da prelevare
38 */
39 public void withdraw(double amount)
40 {
41     balance = balance - amount;
42 }
43
44 /**
45     Ispeziona il valore attuale del saldo del conto bancario.
46     @return il saldo attuale
47 */
48 public double getBalance()
49 {
50     return balance;
51 }
52 }
```



### Auto-valutazione

11. Immaginate di aver modificato la classe `BankAccount` in modo che ciascun conto bancario sia dotato di numero di conto. Che ricaduta ha questa modifica sulle variabili di esemplare?
12. Perché con l'esecuzione di questa classe non riuscite a svuotare il conto bancario di vostra madre?

```

public class BankRobber
{
    public static void main(String[] args)
    {
        BankAccount momSavings = new BankAccount(1000);
        momSavings.balance = 0;
    }
}
```

13. La classe `Rectangle` ha quattro variabili di esemplare: `x`, `y`, `width` e `height`. Scrivete una possibile implementazione del metodo `getWidth`.
14. Scrivete una possibile implementazione del metodo `translate` della classe `Rectangle`.

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R3.4, R3.10 e E3.7, al termine del capitolo.



### Errori comuni 3.2

#### Ignorare le variabili parametro

Un errore commesso veramente molto spesso dai principianti è quello di ignorare le variabili parametro di metodi o costruttori, una cosa che solitamente avviene quando si

usa un'assegnazione che costituisce soltanto un esempio, con valori specifici. Supponiamo, ad esempio, di dover progettare una classe `Letter`, che rappresenti una lettera con mittente e destinatario, come in questo esempio:

Dear John:

I am sorry we must part.  
I wish you all the best.

Sincerely,

Mary

Vediamo ora questo tentativo di implementazione sbagliato:

```
public class Letter
{
    private String recipient; // destinatario
    private String sender; // mittente

    public Letter(String aRecipient, String aSender)
    {
        recipient = "John"; // ERRORE, dovrebbe essere la variabile parametro
        sender = "Mary"; // Stesso errore
    }
    . . .
}
```

Il costruttore ignora i nomi del mittente e del destinatario, che vengono invece forniti come parametri. Se l'utente della classe costruisce una lettera in questo modo:

```
new Letter("John", "Yoko")
```

il mittente sarà di nuovo "Mary".

Il costruttore deve usare le variabili parametro, in questo modo:

```
public Letter(String aRecipient, String aSender)
{
    recipient = aRecipient;
    sender = aSender;
}
```



## Consigli pratici 3.1

### Realizzare una classe

Questa sezione speciale vi spiegherà come realizzare una classe a partire da specifiche assegnate.

**Problema.** Progettare una classe che costituisca un modello di registratore di cassa self-service. Il cliente scansiona i prezzi degli articoli che vuole acquistare e versa denaro nella macchina, che restituisce correttamente il resto.

**Fase 1.** Identificare i metodi che bisogna mettere a disposizione

In una simulazione non è necessario realizzare tutte le caratteristiche presenti nel mondo reale, sono troppe: in questo esempio del registratore di cassa, non ci occupiamo di tasse o di pagamenti con carta di credito. Il compito assegnato specifica *quali aspetti* di un registratore di cassa self-service devono essere simulati dalla classe. Elencateli:

- Registra il prezzo di vendita per un articolo acquistato.
- Registra una somma di denaro come pagamento.
- Calcola il resto dovuto al cliente.

**Fase 2.** Specificare l'interfaccia pubblica

Trasformate l'elenco scritto nella Fase 1 in un insieme di metodi, specificando i tipi di dati per i parametri e per i valori restituiti. Molti programmati ritengono che questo passo sia più semplice se si scrivono invocazioni di metodi applicate a un oggetto usato come esempio, in questo modo:

```
CashRegister register = new CashRegister();
register.recordPurchase(29.50);
register.recordPurchase(9.25);
register.receivePayment(50);
double change = register.giveChange();
```

A questo punto abbiamo l'elenco dei metodi:

- public void recordPurchase(double amount)
- public void receivePayment(double amount)
- public double giveChange()

Per completare l'interfaccia pubblica bisogna ancora specificare i costruttori. Chiedetevi quali informazioni siano necessarie per costruire un oggetto della classe. A volte serviranno due costruttori: uno che imposti tutte le variabili di esemplare a valori predefiniti e un altro che usi a tale scopo valori forniti dall'utente.

Nel caso del registratore di cassa, possiamo anche sopravvivere con un solo costruttore che crei un registratore di cassa vuoto, ma un registratore di cassa più realistico inizierà a funzionare con una certa dotazione di monete e di banconote, in modo da poter dare il resto corretto anche ai primi clienti: ciò, però, esula dagli obiettivi del compito assegnato.

Ecco, quindi, l'unico costruttore che inseriamo nella classe:

- public CashRegister()

**Fase 3.** Scrivere la documentazione dell'interfaccia pubblica

Ecco la documentazione, con i commenti che descrivono la classe e i suoi metodi:

```
/**
 * Un registratore di cassa somma i prezzi degli articoli venduti
 * e calcola il resto dovuto al cliente.
```

```
/*
public class CashRegister
{
    /**
     * Costruisce un registratore di cassa senza soldi nel cassetto.
     */
    public CashRegister()
    {
    }

    /**
     * Registra la vendita di un articolo.
     * @param amount il prezzo dell'articolo
     */
    public void recordPurchase(double amount)
    {
    }

    /**
     * Riceve un pagamento dal cliente e lo registra.
     * @param amount l'ammontare del pagamento
     */
    public void receivePayment(double amount)
    {
    }

    /**
     * Calcola il resto dovuto al cliente e azzerà il conto per il successivo.
     * @return il resto dovuto al cliente
     */
    public double giveChange()
    {
    }
}
```

#### Fase 4. Identificare le variabili di esemplare

Chiedetevi quali informazioni debba memorizzare un oggetto al proprio interno per svolgere il suo compito, ricordando che i metodi possono essere invocati in qualunque ordine. L'oggetto deve avere memoria interna a sufficienza per gestire tutti i metodi, usando soltanto le proprie variabili di esemplare e i parametri dei metodi. Analizzate ciascun metodo, preferibilmente iniziando dal più semplice o da un metodo particolarmente significativo, e chiedetevi di cosa abbia bisogno per portare a termine il proprio compito. Create variabili di esemplare per conservare le informazioni necessarie ai metodi.

È anche importante non definire variabili di esemplare non necessarie (si veda la sezione Errori Comuni 3.3): se un valore può essere calcolato a partire da altre variabili di esemplare, solitamente è preferibile calcolarlo quando è richiesto piuttosto che memorizzarlo.

Nell'esempio del registratore di cassa dobbiamo tenere traccia della spesa totale e del pagamento effettuato: da questi due valori si può poi calcolare il resto dovuto al cliente.

```
public class CashRegister
{
```

```

    private double purchase;
    private double payment;
    ...
}

```

#### Fase 5. Realizzare costruttori e metodi

Realizzate i costruttori e i metodi della classe, uno alla volta, iniziando dai più semplici. Ecco, ad esempio, la realizzazione del metodo `recordPurchase`:

```

public void recordPurchase(double amount)
{
    purchase = purchase + amount;
}

```

Il metodo `receivePayment` è molto simile:

```

public void receivePayment(double amount)
{
    payment = payment + amount;
}

```

Per quale motivo il metodo somma la quantità di denaro ricevuta, invece di eseguire la semplice assegnazione `payment = amount`? Un cliente potrebbe fare due pagamenti distinti, ad esempio due banconote da 10 dollari, e la macchina deve essere in grado di elaborarli entrambi, ricordandoli. Come regola generale, salvo eccezioni, ciascun metodo può essere invocato più volte e le invocazioni di metodi diversi possono avvenire in qualsiasi ordine.

Ecco, infine, il metodo `giveChange`, che è un po' più complicato: calcola il resto dovuto e riporta il registratore di cassa al suo stato iniziale, in modo che sia pronto per la vendita successiva.

```

public double giveChange()
{
    double change = payment - purchase;
    purchase = 0;
    payment = 0;
    return change;
}

```

Se vi accorgete di avere problemi con la realizzazione di un metodo o di un costruttore, può darsi che sia necessario ripensare alle scelte fatte nella determinazione delle variabili di esemplare. Per un principiante è molto frequente iniziare con un insieme di variabili di esemplare che non costituisce un modello accurato dello stato di un oggetto. Non abbiate timore di tornare sui vostri passi: fatelo senza esitazioni e modificate l'insieme delle variabili di esemplare.

Nel pacchetto dei file scaricabili per questo libro, la cartella `how_to_1` del Capitolo 3 contiene il codice sorgente completo della classe.

#### Fase 6. Collaudare la classe

Scrivete un breve programma di collaudo ed eseguitelo. Il programma di collaudo può, ad esempio, eseguire le invocazioni di metodi che avete usato come esempio nella Fase 2.

```
public class CashRegisterTester
{
    public static void main(String[] args)
    {
        CashRegister register = new CashRegister();

        register.recordPurchase(29.50);
        register.recordPurchase(9.25);
        register.receivePayment(50);

        double change = register.giveChange();

        System.out.println(change);
        System.out.println("Expected: 11.25");
    }
}
```

Il programma di collaudo visualizza:

```
11.25
Expected: 11.25
```



## Esempi completi 3.1

### Realizzare un semplice menu

L'obiettivo è quello di progettare la classe `Menu`, i cui oggetti sono in grado di visualizzare un menu come questo:

- 1) Open new account
- 2) Log into existing account
- 3) Help
- 4) Quit

I numeri vengono associati alle opzioni in modo automatico, quando queste vengono aggiunte al menu.

#### Fase 1. Identificare i metodi che bisogna mettere a disposizione

La descrizione del problema elenca due diversi compiti.

- Visualizza il menu.
- Aggiungi un'opzione al menu.

#### Fase 2. Specificare l'interfaccia pubblica

Ora trasformiamo l'elenco della Fase 1 in un insieme di metodi, specificando i tipi di dati per i parametri e per i valori restituiti. Come suggerito nella sezione Consigli pratici 3.1, iniziamo scrivendo alcuni esempi di utilizzo:

```
mainMenu.addOption("Open new account");
mainMenu.addOption("Log into existing account");
mainMenu.display();
```

A questo punto abbiamo l'elenco dei metodi.

- `public void addOption(String option)`
- `public void display()`

Per completare l'interfaccia pubblica dobbiamo specificare i costruttori. Possiamo scegliere tra due soluzioni:

- Fornire un costruttore che crei un menu dotato di una prima opzione: `Menu(String firstOption)`
- Fornire un costruttore che crei un menu privo di opzioni: `Menu()`

Entrambe le scelte funzionerebbero bene. Se preferiamo la seconda, l'utente della classe dovrà invocare `addOption` per aggiungere al menu anche la prima opzione, dato che, in fin dei conti, non ha alcun senso costruire un menu privo di opzioni. A prima vista questa procedura sembra un inutile onere per il programmatore che utilizzerà la classe, ma, d'altro canto, di solito è concettualmente più semplice che un'interfaccia pubblica non preveda casi speciali (e dover fornire una prima opzione nel costruttore certamente lo è). Di conseguenza, decidiamo che “la soluzione più semplice è migliore” (*simplest is best*) e dichiariamo questo unico costruttore:

- `public Menu()`

### Fase 3. Scrivere la documentazione dell'interfaccia pubblica

Ecco la documentazione, con i commenti che descrivono la classe e i suoi metodi:

```
/**
 * Un menu che viene visualizzato in una finestra di console.
 */
public class Menu
{
    /**
     * Costruisce un menu privo di opzioni.
     */
    public Menu()
    {
    }

    /**
     * Aggiunge un'opzione alla fine di questo menu.
     * @param option l'opzione da aggiungere
     */
    public void addOption(String option)
    {
    }

    /**
     * Visualizza il menu nella finestra di console.
     */
    public void display()
    {
    }
}
```

**Fase 4.** Identificare le variabili di esemplare

Cosa ha bisogno di memorizzare un oggetto di tipo `Menu` per adempiere alle proprie responsabilità? Ovviamente, per poter visualizzare il menu, deve necessariamente memorizzarne il testo. Considerate ora il metodo `addOption`, che aggiunge al menu un numero progressivo e la relativa opzione. Da dove viene il numero? L'oggetto che rappresenta il menu ha bisogno di memorizzare anche quello, in modo da poterlo incrementare ogni volta che viene invocato il metodo `addOption`.

Quindi, le nostre variabili di esemplare sono:

```
public class Menu
{
    private String menuText;
    private int optionCount;
    ...
}
```

**Fase 5.** Realizzare costruttori e metodi

Realizziamo ora i costruttori e i metodi della classe, uno alla volta, nell'ordine che ci sembra più comodo. Il costruttore è particolarmente semplice:

```
public Menu()
{
    menuText = "";
    optionCount = 0;
}
```

E il metodo `display` è forse altrettanto semplice:

```
public void display()
{
    System.out.println(menuText);
}
```

Per realizzare il metodo `addOption` dobbiamo, invece, ragionare un po' di più. Ecco una traccia del metodo descritta mediante pseudocodice:

Incrementare il contatore delle opzioni.

Aggiungere quanto segue al testo del menu:

Il valore del contatore delle opzioni

Una parentesi tonda chiusa

L'opzione da aggiungere

Un carattere speciale per andare a capo, in modo che l'opzione

successiva inizi su una riga nuova

Come si può aggiungere testo a una stringa? Guardando la documentazione API della classe `String`, scoprirete il metodo `concat`. Ad esempio, l'invocazione

```
menuText.concat(option)
```

crea una stringa contenente, in successione ordinata, i caratteri presenti nelle stringhe `menuText` e `option`. Potete, poi, memorizzarla nuovamente nella variabile `menuText`, in questo modo:

```
menuText = menuText.concat(option);
```

Come vedrete nel Capitolo 4, si può ottenere lo stesso risultato usando l'operatore `+`:

```
menuText = menuText + option;
```

Nella soluzione che proponiamo, usiamo proprio questo operatore, perché è particolarmente comodo. Quindi, il nostro metodo diventa:

```
public void addOption(String option)
{
    optionCount = optionCount + 1;
    menuText = menuText + optionCount + ") " + option + "\n";
}
```

#### Fase 6. Collaudare la classe

Ecco un breve programma di collaudo che mette all'opera tutti i metodi dell'interfaccia pubblica della classe `Menu`.

```
public class MenuDemo
{
    public static void main(String[] args)
    {
        Menu mainMenu = new Menu();
        mainMenu.addOption("Open new account");
        mainMenu.addOption("Log into existing account");
        mainMenu.addOption("Help");
        mainMenu.addOption("Quit");
        mainMenu.display();
    }
}
```

#### Esecuzione del programma:

- 1) Open new account
- 2) Log into existing account
- 3) Help
- 4) Quit

---

## 3.4 Collaudo di unità

Nel paragrafo precedente abbiamo portato a termine l'implementazione della classe `BankAccount`. Come la si può usare? Potete sicuramente compilare il file `BankAccount.java`, ma non potete eseguire il file `BankAccount.class` che viene prodotto, perché non contiene un metodo `main`. Questa è una situazione molto comune: la maggior parte delle classi non contiene un metodo `main`.

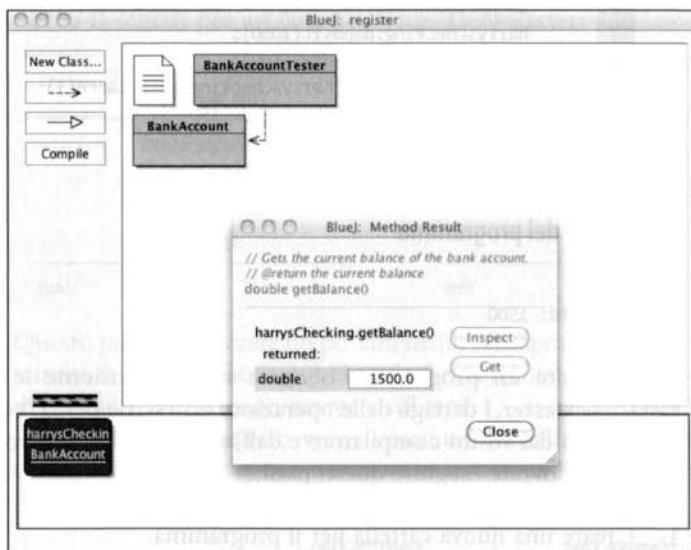
Un collaudo di unità verifica il corretto funzionamento di una classe a sé stante, senza che sia inserita in un programma completo.

Prima o poi la vostra classe diventerà parte di un programma più complesso che interagisce con utenti, memorizza dati in file e così via. Prima, però, di procedere a una tale integrazione è sempre meglio collaudare la classe a sé stante. Tale collaudo, al di fuori di un programma completo, viene chiamato **collaudo di unità** (*unit testing*).

Per collaudare una classe ci sono fondamentalmente due possibilità. Alcuni ambienti di sviluppo interattivi hanno comandi che consentono di creare oggetti e di invocarne metodi (come visto nella sezione Argomenti avanzati 2.1), per cui potete collaudare una classe in modo molto semplice, costruendo un suo oggetto, invocandone metodi e verificando che vengano prodotti i risultati attesi. La Figura 6 mostra il risultato dell'invocazione, all'interno di BlueJ, del metodo `getBalance` applicato a un oggetto della classe `BankAccount`.

**Figura 6**

Il valore restituito dal metodo `getBalance`, visualizzato con BlueJ



Per collaudare una classe si usa un ambiente interattivo oppure si scrive una classe di test che esegue istruzioni di collaudo.

In alternativa, si può scrivere una classe per il collaudo (o classe di test, *tester class*): una classe il cui metodo `main` contiene enunciati che servono al collaudo di un'altra classe. Come visto nel Paragrafo 2.7, una classe di test esegue solitamente questi passi:

1. Costruisce uno o più oggetti della classe che si vuole collaudare.
2. Ne invoca uno o più metodi.
3. Visualizza uno o più risultati.
4. Visualizza i corrispondenti risultati previsti.

La classe `MoveTester`, vista nel Paragrafo 2.7, è un valido esempio di classe di test: esegue metodi della classe `Rectangle`, una classe della libreria Java.

Ecco, invece, una classe che esegue metodi della classe `BankAccount`. Il suo metodo `main` costruisce un oggetto di tipo `BankAccount`, ne invoca i metodi `deposit` e `withdraw`, quindi visualizza il saldo finale.

Visualizziamo anche il valore previsto per tale saldo. Nel nostro programma d'esempio, versiamo 2000 dollari e ne preleviamo 500, quindi ci aspettiamo che il saldo finale sia 1500 dollari.

### File BankAccountTester.java

```

1  /**
2   * Una classe di collaudo per la classe BankAccount.
3  */
4  public class BankAccountTester
5  {
6      /**
7       * Collauda i metodi della classe BankAccount.
8       * @param args non utilizzato
9      */
10     public static void main(String[] args)
11     {
12         BankAccount harrysChecking = new BankAccount();
13         harrysChecking.deposit(2000);
14         harrysChecking.withdraw(500);
15         System.out.println(harrysChecking.getBalance());
16         System.out.println("Expected: 1500");
17     }
18 }
```

### Esecuzione del programma

```

1500
Expected: 1500
```

Per ottenere un programma occorre mettere insieme le due classi, `BankAccount` e `BankAccountTester`. I dettagli delle operazioni necessarie per la “costruzione” del programma dipendono dal vostro compilatore e dall’ambiente di sviluppo; nella maggior parte degli ambienti, dovrete eseguire questi passi:

1. Creare una nuova cartella per il programma.
2. Creare due file, uno per ciascuna classe.
3. Compilare entrambi i file.
4. Eseguire il programma di collaudo.

Molti studenti si sorprendono del fatto che un programma così semplice contenga due classi, ma questo è normale, perché le due classi hanno obiettivi radicalmente distinti: la classe `BankAccount` descrive oggetti che calcolano saldi bancari, mentre la classe `BankAccountTester` esegue un collaudo che mette alla prova l’operatività di un oggetto di tipo `BankAccount`.



### Auto-valutazione

15. Quando si esegue il programma `BankAccountTester`, quanti oggetti di tipo `BankAccount` vengono costruiti? E quanti oggetti di tipo `BankAccountTester`?
16. Perché la classe `BankAccountTester` non è necessaria negli ambienti di sviluppo che, come BlueJ, consentono il collaudo interattivo?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi E3.6 e E3.13, al termine del capitolo.

## 3.5 Problem Solving: tenere traccia dell'esecuzione

Alcuni ricercatori hanno cercato di capire perché alcuni studenti imparano a programmare più facilmente di altri. Una competenza importante dei buoni programmati è la capacità di simulare le azioni compiute da un programma usando carta e penna: in questo paragrafo vedremo proprio come imparare a fare questo, tenendo traccia delle invocazioni di metodi e di come queste agiscono sugli oggetti usati in un programma.

Scrivete i metodi su una faccia di una scheda e le variabili di esemplare sull'altra faccia.

Usiamo una scheda o un foglietto adesivo per ciascun oggetto: su una faccia della scheda scriviamo i metodi che possono essere eseguiti con l'oggetto rappresentato dalla scheda stessa, mentre sull'altra faccia (il retro della scheda) disegniamo una tabella con i nomi e i valori delle sue variabili di esemplare.

Ecco la scheda per un oggetto di tipo `CashRegister`:

| <code>CashRegister reg1</code> | <code>reg1.purchase</code> | <code>reg1.payment</code> |
|--------------------------------|----------------------------|---------------------------|
| <code>recordPurchase</code>    |                            |                           |
| <code>receivePayment</code>    |                            |                           |
| <code>giveChange</code>        |                            |                           |
| <i>fronte</i>                  | <i>retro</i>               |                           |

Questo può farvi toccare un po' con mano l'incapsulamento: un oggetto viene manipolato tramite la sua interfaccia pubblica (la faccia frontale della scheda), mentre le sue variabili di esemplare rimangono nascoste sul retro.

Quando un oggetto viene costruito, inizializziamo i valori delle sue variabili di esemplare:

| <code>reg1.purchase</code> | <code>reg1.payment</code> |
|----------------------------|---------------------------|
| 0                          | 0                         |

Quando viene invocato un metodo modificatore, aggiornate le variabili di esemplare.

Ogni volta che viene invocato un metodo modificatore, tracciamo una riga sui vecchi valori delle variabili di esemplare e scriviamo quelli nuovi proprio al di sotto. Ecco, ad esempio, cosa accade all'oggetto precedente dopo un'invocazione del metodo `recordPurchase`:

| <code>reg1.purchase</code> | <code>reg1.payment</code> |
|----------------------------|---------------------------|
| <del>0</del>               | 0                         |
| 19.95                      |                           |

Se il programma usa più di un oggetto, si useranno più schede, una per ogni oggetto:



## Computer e società 3.1

### Apparati per il voto elettronico

Nelle elezioni presidenziali statunitensi tenutesi nel 2000 i voti sono stati raccolti con apparati di diverso tipo. Alcune macchine elaboravano schede di votazione nelle quali i votanti praticavano fori per indicare la propria scelta. Per imperizia dei votanti, a volte residui di carta (ormai tristemente famosi con il nome di "chads") rimasero al loro posto nelle schede perforate, provocando errori nel conteggio dei voti. Si rese necessario un ulteriore conteggio manuale dei voti, che, però, non venne portato a termine per mancanza di tempo e per controversie procedurali. L'elezione fu vinta con scarto molto ridotto e in molte persone rimase il dubbio che il risultato delle elezioni avrebbe potuto essere diverso se le macchine adibite al conteggio avessero considerato con maggior precisione le intenzioni dei votanti.

In seguito, i fabbricanti di apparecchiature elettroniche per votazioni sostennero che con i loro apparati si sarebbero evitati i problemi delle schede perforate o dei moduli a lettura ottica. In una macchina elettronica per votazioni, i votanti esprimono la propria preferenza premendo un pulsante oppure sfiorando un'icona sullo schermo di un computer e, solitamente, a ogni votante viene poi visualizzata una schermata riassuntiva del proprio voto prima di dargli la possibilità di renderlo definitivo. Il procedimento è molto simile a quello messo in atto con uno sportello bancario automatico.

Sembra ragionevole supporre che queste macchine possano au-

mentare la probabilità che ciascun voto venga contato seguendo esattamente le intenzioni di voto dell'eletto che l'ha espresso, ma ci sono stati dibattiti molto accesi su questa tipologia di macchine elettroniche per le votazioni. Se una macchina si limita a memorizzare i voti espressi, visualizzando solamente il conteggio totale al termine delle votazioni, come si può verificare che il suo funzionamento sia corretto? All'interno della macchina troviamo un computer che esegue un programma e, come sapete bene per esperienza diretta, i programmi possono contenere errori.

Ed è proprio così: alcune macchine elettroniche utilizzate per votazioni hanno veramente dei problemi. Ci sono stati casi in cui i risultati riportati dalla macchina erano impossibili da ottenere: quando una macchina indica un numero di voti inferiore o superiore al numero dei votanti, allora è chiaro che ha compiuto un errore. Sfortunatamente, a quel punto è impossibile ricostruire il conteggio esatto dei voti, anche se è plausibile che, con il passare del tempo, questi problemi software vengano risolti. Un problema ancora più subdolo consiste nel fatto che, se i risultati si rivelano plausibili, nessuno farà mai indagini al riguardo.

Molti teorici dell'informazione hanno affrontato questo problema e hanno confermato che è impossibile, con le tecnologie attuali, affermare con certezza assoluta che un software sia privo di errori e che non sia stato contraffatto. Molti di loro hanno raccomandato l'utilizzo combinato

di una macchina elettronica per votazioni e di un *procedimento di votazione verificabile* (per maggiori informazioni, potete consultare il sito <http://verifiedvoting.org>): tipicamente, una macchina per votazioni che consente la verifica procede alla stampa cartacea di tutti i voti che vengono espressi e ciascun elettore verifica la stampa del proprio voto e la inserisce in un'urna tradizionale, dalla quale i voti possono essere recuperati e contati in caso di problemi all'apparecchiatura elettronica.

In questo periodo, a queste idee si oppongono fermamente sia i produttori di apparecchiature elettroniche per votazioni sia i loro potenziali acquirenti, cioè chi sovrintende alle elezioni politiche e amministrative. I produttori sono contrari perché gli inevitabili aumenti di costo delle apparecchiature sarebbero difficili da far accettare ai compratori, che hanno sempre bilanci molto limitati. I responsabili delle elezioni temono, invece, problemi di malfunzionamento delle stampanti e alcuni di loro hanno dichiarato pubblicamente di preferire apparecchi che eliminino del tutto la necessità di noiosi conteggi manuali.

Cosa ne pensate voi? Probabilmente siete abituati a utilizzare uno sportello automatico per prelevare contante dal vostro conto bancario. Controllate sempre la ricevuta che vi viene consegnata? Effettuate un controllo incrociato con il vostro estratto conto? Anche se non lo fate, confidate nel fatto che altre persone controllino con cura i propri conti, in modo che la banca non cerchi di

defraudare i propri clienti in modo diffuso?

E, alla fine dei conti, la correttezza del funzionamento degli sportelli bancari automatici è più o meno importante di quella delle macchine per elezioni? Non è forse vero che qualsiasi procedura di votazione è

inevitabilmente soggetta a errori e consente frodi? Ha senso aumentare il costo delle apparecchiature, aggiungendo una stampante, per rimediare a un rischio di errore sostanzialmente basso? Gli informatici non possono rispondere a queste domande: a esse deve rispondere

la società intera, dopo essere stata adeguatamente informata. Come ogni professionista, però, un informatico ha l'obbligo di rendere noti i vantaggi e gli svantaggi, dal punto di vista tecnico, di questi apparati elettronici.

| reg1.purchase | reg1.payment | reg2.purchase | reg2.payment |
|---------------|--------------|---------------|--------------|
| <del>0</del>  | <del>0</del> | <del>0</del>  | <del>0</del> |
| 19.95         | 19.95        | 29.50         | 50.00        |

Questi schemi sono utili anche durante la fase di progettazione di una classe. Immaginiamo, infatti, di dover migliorare il progetto della classe `CashRegister`, in modo che sia in grado di calcolare le tasse che gravano sulla vendita (*sales tax*). Aggiungiamo, sulla faccia frontale della scheda, i metodi `recordTaxablePurchase` e `getSalesTax`, poi giriamo la scheda e analizziamo le variabili di esemplare, chiedendoci se l'oggetto memorizza informazioni sufficienti per calcolare tutto quanto è necessario, tenendo presente che ciascun oggetto è un'entità autonoma, quindi qualsiasi valore usato in una elaborazione deve essere:

- una variabile di esemplare, oppure
- il parametro di un metodo, oppure
- una variabile statica (caso poco frequente, che vedremo nel Paragrafo 8.4).

Per calcolare le tasse su una vendita dobbiamo conoscere la percentuale di tassazione e l'importo imponibile, cioè la spesa totale per gli articoli sottoposti a tassazione (ad esempio, negli Stati Uniti il cibo solitamente non è tassato). Con le attuali variabili di esemplare, queste informazioni non sono disponibili: aggiungiamo due variabili, una per la percentuale di tassazione e una per l'importo imponibile. La percentuale di tassazione può essere un parametro di costruzione (nella ragionevole ipotesi che rimanga costante durante la vita dell'oggetto), mentre, nel momento in cui registriamo la vendita di un articolo, dobbiamo essere informati sul fatto che questo sia imponibile o meno: nel primo caso, il suo costo va aggiunto all'imponibile totale.

Consideriamo, ad esempio, gli enunciati seguenti:

```
CashRegister reg3 = new CashRegister(7.5); // tassazione al 7.5%
reg3.recordPurchase(3.95); // articolo non sottoposto a tassazione
reg3.recordTaxablePurchase(19.95); // articolo tassato
```

Registrando sulla scheda gli effetti di questi tre enunciati, si ottiene:

| reg3.purchase | reg3.taxablePurchase | reg3.payment | reg3.taxRate |
|---------------|----------------------|--------------|--------------|
| -0            | -0                   | 0            | 7.5          |
| 3.95          | 19.95                |              |              |

Con queste informazioni siamo in grado di calcolare le tasse sulla vendita: `taxablePurchase × taxRate/100`. Il fatto di tenere traccia, passo dopo passo, di ciò che succede durante l'esecuzione ci ha aiutato a capire che erano necessarie due nuove variabili di esemplare.



## Auto-valutazione

17. Considerate una classe `Car` che simuli il consumo di carburante in un'automobile, che immaginiamo prefissato a un valore fornito dal costruttore (in miglia per gallone). La classe ha metodi per aggiungere carburante (`addGas`), guidare per una determinata distanza (`drive`) e verificare la quantità di carburante rimasta nel serbatoio (`getLeftGas`). Scrivete la scheda per un oggetto di tipo `Car`, individuando le variabili di esemplare opportune e mostrando i loro valori dopo la costruzione dell'oggetto.
  18. Con un'opportuna scheda, tenete traccia dell'esecuzione del codice seguente:
- ```
Car myCar = new Car(25);
myCar.addGas(20);
myCar.drive(100);
myCar.drive(200);
myCar.addGas(5);
```
19. Immaginate di dover simulare il contachilometri dell'automobile, aggiungendo il metodo `getMilesDriven` alla classe `Car` descritta nella domanda 17. Aggiungete alla scheda dell'oggetto una variabile di esemplare che consenta di implementare il metodo.
  20. Rispondete di nuovo alla domanda 18, aggiornando anche la variabili di esemplare che avete aggiunto rispondendo alla domanda 19.

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R3.19, R3.20 e R3.21, al termine del capitolo.

## 3.6 Variabili locali

Le variabili locali sono dichiarate nel corpo di un metodo.

In questo paragrafo parleremo del comportamento delle variabili *locali*. Una **variabile locale** (*local variable*) è una variabile che viene dichiarata all'interno del corpo di un metodo. Ad esempio, il metodo `giveChange` visto nella sezione Consigli pratici 3.1 dichiara la variabile locale `change`:

```
public double giveChange()
{
    double change = payment - purchase;
    purchase = 0;
    payment = 0;
```

```
    return change;
}
```

Le variabili parametro sono simili alle variabili locali, ma sono dichiarate nelle intestazioni dei metodi. Ad esempio, questo metodo dichiara una variabile parametro di nome `amount`:

```
public void receivePayment(double amount)
```

**Quando un metodo termina la propria esecuzione, le sue variabili locali scompaiono.**

Le variabili locali e le variabili parametro appartengono a un metodo: quando il metodo viene eseguito, queste variabili entrano in azione (“nascono”); quando il metodo termina la propria esecuzione, esse scompaiono immediatamente (“muoiono”). Ad esempio, se invocate `register.giveChange()`, viene creata una variabile di nome `change`, che scompare nel momento in cui il metodo termina la propria esecuzione.

Al contrario, le variabili di esemplare appartengono agli oggetti, non ai metodi. Quando viene costruito un oggetto, vengono create anche le sue variabili di esemplare, che rimangono in vita finché esiste almeno un metodo che usa tale oggetto. La macchina virtuale Java contiene un agente (il **garbage collector**, cioè “raccoglitrice di spazzatura”) che periodicamente elimina gli oggetti che non sono più utilizzati.

Un’importante differenza tra variabili di esemplare e variabili locali riguarda la loro *inizializzazione*. Tutte le variabili locali devono essere inizializzate: se non assegnate un valore iniziale a una variabile locale, il compilatore segnala un errore nel momento in cui cercate di utilizzarla per la prima volta (notate che le variabili parametro vengono inizializzate nel momento in cui il metodo viene invocato).

Le variabili di esemplare vengono inizializzate a un valore predefinito, mentre le variabili locali devono essere inizializzate esplicitamente.

Le variabili di esemplare vengono inizializzate a un valore predefinito, mentre le variabili locali devono essere inizializzate esplicitamente.



## Auto-valutazione

21. Cosa hanno in comune le variabili locali e le variabili parametro? In quale aspetto essenziale sono invece diverse?
22. Perché si è resa necessaria l’introduzione della variabile locale `change` nel metodo `giveChange()`? Spiegate, cioè, perché il metodo non termina semplicemente con l’enunciato

```
return payment - purchase;
```

23. Considerate un oggetto, `reg1`, di tipo `CashRegister`, la cui variabile di esemplare `payment` abbia il valore 20 e la cui variabile di esemplare `purchase` abbia il valore 19.5. Registrate sulla scheda dell’oggetto gli effetti dell’invocazione `reg1.giveChange()`, tenendo conto anche di ciò che accade alla variabile locale `change`: tracciate una X nella sua colonna quando la variabile cessa di esistere.

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi R.3.15 e R.3.16, al termine del capitolo.



## Errori comuni 3.3

---

### Duplicare le variabili di esemplare come variabili locali

Spesso i programmatori principianti aggiungono la definizione del tipo della variabile all'inizio di un enunciato di assegnazione, trasformandolo così nella dichiarazione di una variabile locale. Ad esempio:

```
public double giveChange()
{
    double change = payment - purchase;
    double purchase = 0; // ERRORE! Questo dichiara una variabile locale
    double payment = 0; // ERRORE! La variabile di esemplare non viene aggiornata
    return change;
}
```

Un altro errore frequente consiste nella dichiarazione di una variabile parametro avente lo stesso nome di una variabile di esemplare, come in questo costruttore di BankAccount:

```
public BankAccount(double balance)
{
    balance = balance; // ERRORE! Non assegna un valore alla variabile di esemplare
}
```

Questo costruttore assegna semplicemente alla variabile parametro il valore che ha già, lasciandola inalterata. Una semplice soluzione consiste nell'utilizzo di un nome diverso per la variabile parametro:

```
public BankAccount(double initialBalance)
{
    balance = initialBalance; // Così va bene
}
```



## Errori comuni 3.4

---

### Definire variabili di esemplare non necessarie

Un errore tipico dei programmatori principianti è l'uso di una variabile di esemplare quando sarebbe sufficiente una variabile locale. Considerate, ad esempio, la variabile change definita nel metodo giveChange. Non viene utilizzata in nessun altro metodo, perché è una variabile locale di quel metodo, ma cosa succede se la dichiariamo come variabile di esemplare?

```
public class CashRegister
{
    private double purchase;
    private double payment;
    private double change; // definizione inadeguata

    public double giveChange()
    {
```

```

        change = payment - purchase;
        purchase = 0;
        payment = 0;
        return change;
    }
    ...
}

```

Questa classe funzionerà, ma contiene un pericolo nascosto: altri metodi potrebbero ispezionare o modificare la variabile di esemplare `change`, e questo sarebbe probabilmente motivo di confusione.

Le variabili di esemplare vanno usate per quei valori che un oggetto deve ricordare tra due diverse invocazioni di suoi metodi, mentre le variabili locali sono adatte a contenere valori che non serve preservare quando il metodo che li usa è terminato.



## Errori comuni 3.5

### Dimenticarsi di inizializzare i riferimenti agli oggetti in un costruttore

Se dimenticarsi di inizializzare una variabile locale è un errore comune, è altrettanto facile trascurare l'inizializzazione delle variabili di esemplare. Ciascun costruttore deve garantire che tutte le variabili di esemplare assumano valori corretti.

Se non inizializzate una variabile di esemplare, il compilatore Java provvederà a farlo per voi. I numeri avranno il valore zero, ma i riferimenti a oggetti, come le variabili stringa, avranno il valore `null`.

Spesso lo zero è un comodo valore predefinito per i numeri, mentre `null` difficilmente costituisce una scelta opportuna per gli oggetti. Esamineate questo costruttore "pigro" in una versione modificata della classe `BankAccount`:

```

public class BankAccount
{
    private double balance;
    private String owner;
    ...
    public BankAccount(double initialBalance)
    {
        balance = initialBalance;
    }
}

```

In conseguenza della sua esecuzione, la variabile `balance` viene inizializzata, mentre la variabile `owner` assume il valore `null`: questo potrebbe costituire un problema in seguito, in quanto non si possono invocare metodi usando il riferimento `null`.

Per evitare questo problema, prendete l'abitudine di inizializzare tutte le variabili di esemplare:

```

public BankAccount(double initialBalance)
{
    balance = initialBalance;
    owner = "None";
}

```

### 3.7 Il riferimento this

Quando si invoca un metodo, i dati da elaborare vengono trasferiti con due modalità diverse:

- l'oggetto con cui il metodo viene invocato, e
- i parametri (o argomenti) del metodo.

Ad esempio, quando invochiamo

```
momsSavings.deposit(500);
```

il metodo `deposit` deve conoscere l'oggetto che rappresenta il conto su cui effettuare il versamento (`momsSavings`) e la quantità di denaro da versare (500).

Nell'implementazione di un metodo si definisce una variabile parametro per ogni argomento che si vuole ricevere, ma non viene analogamente definita una variabile parametro per memorizzare l'oggetto con cui il metodo verrà invocato, il cosiddetto **parametro implicito**. Per coerenza, tutte le altre variabili parametro (come `amount` in questo esempio) vengono chiamate **parametri espliciti**.

Analizziamo di nuovo il codice del metodo `deposit`:

```
public void deposit(double amount)
{
    balance = balance + amount;
}
```

Qui `amount` è un parametro esplicito, mentre il parametro implicito non è visibile: in effetti, è proprio questo che motiva il nome “implicito”. Ma che significato ha, di preciso, `balance`? Dopo tutto, un programma può avere più oggetti di tipo `BankAccount` e *ciascuno di essi* ha il proprio saldo.

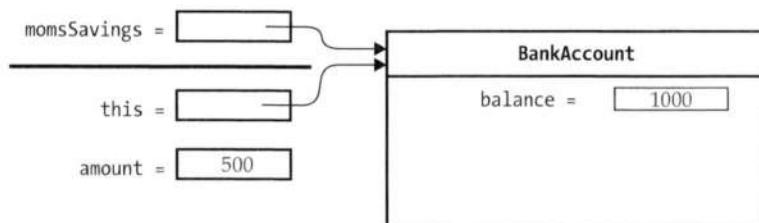
Dato che stiamo versando denaro nel conto `momsSavings`, per noi `balance` deve ovviamente avere il significato di `momsSavings.balance`. In generale, quando all'interno di un metodo si fa riferimento a una variabile di esemplare, si intende la variabile di esemplare del parametro implicito.

All'interno di qualsiasi metodo potete usare la parola riservata `this` per accedere al suo parametro implicito (cioè all'oggetto con cui è stato invocato il metodo). Ad esempio, nella precedente invocazione di metodo, `this` faceva riferimento al medesimo oggetto a cui fa riferimento `momsSavings`, come si può vedere nella Figura 7.

L'uso, all'interno di un metodo, del nome di una variabile di esemplare rappresenta la variabile di esemplare del parametro implicito.

**Figura 7**

Il parametro implicito nell'invocazione di un metodo



## L'enunciato

```
balance = balance + amount;
```

Il riferimento `this` rappresenta  
il parametro implicito.

ha il seguente significato:

```
this.balance = this.balance + amount;
```

Quando in un metodo si fa riferimento a una variabile di esemplare, il compilatore la associa automaticamente al parametro `this`. Alcuni programmatore preferiscono inserire manualmente il riferimento `this` prima di ogni variabile di esemplare, perché ritengono che ciò renda il codice più comprensibile. Ecco un esempio:

```
public BankAccount(double initialBalance)
{
    this.balance = initialBalance;
}
```

Potete provare anche questo stile e vedere se vi piace.

Il riferimento `this` può anche essere usato per distinguere una variabile di esemplare da una variabile locale o da una variabile parametro. Esaminiamo il costruttore:

```
public BankAccount(double balance)
{
    this.balance = balance;
}
```

Una variabile locale mette in ombra  
una variabile di esemplare che abbia  
lo stesso nome: si può accedere  
a quest'ultima usando  
il riferimento `this`.

L'espressione `this.balance` fa chiaramente riferimento alla variabile di esemplare `balance`, mentre l'uso, a destra, della sola espressione `balance` sembra ambiguo: potrebbe rappresentare la variabile parametro o la variabile di esemplare. Le specifiche del linguaggio Java dicono che in una situazione come questa la variabile locale prevale e “mette in ombra” l'omonima variabile di esemplare. Quindi

```
this.balance = balance;
```

significa: “Assegna alla variabile di esemplare `balance` il valore assunto dalla variabile parametro `balance`”.

Esiste un altro caso in cui è importante tener presente il parametro implicito. Esaminiamo la seguente classe `BankAccount` modificata, in cui abbiamo aggiunto un metodo che applica il canone mensile del conto:

```
public class BankAccount
{
    ...
    public void monthlyFee()
    {
        withdraw(10); // preleva 10 dollari da questo conto
    }
}
```

L'invocazione di un metodo senza usare un parametro implicito agisce sul medesimo oggetto su cui si sta operando.

Questo indica un prelievo *dallo stesso oggetto* di tipo conto bancario con cui si sta eseguendo l'operazione `monthlyFee`. In altre parole, il parametro implicito del metodo `withdraw` è il parametro implicito (invisibile) del metodo `monthlyFee`.

Se pensate che un parametro invisibile generi confusione, potete usare esplicitamente il riferimento `this`, per rendere il metodo più leggibile:

```
public class BankAccount
{
    ...
    public void monthlyFee()
    {
        this.withdraw(10); // preleva 10 dollari da questo conto
    }
}
```

A questo punto avete visto come usare oggetti e realizzare classi, oltre ad aver appreso alcuni importanti dettagli relativi a variabili e parametri di metodi. Il capitolo prosegue, ora, con argomenti relativi alla programmazione grafica, che possono essere considerati facoltativi nel flusso logico della presentazione. Nel prossimo capitolo vedrete, invece, nuove nozioni sui tipi di dati più importanti del linguaggio Java.



## Auto-valutazione

24. Quanti parametri impliciti e quanti parametri esplicativi ci sono nel metodo `withdraw` della classe `BankAccount`, e quali sono i loro nomi e tipi?
25. Qual è il significato di `this.amount` all'interno del metodo `deposit`? Oppure, se l'espressione è priva di significato, per quale motivo lo è?
26. Quanti parametri impliciti e quanti parametri esplicativi ci sono nel metodo `main` della classe `BankAccountTester`, e quali sono i loro nomi?

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi R.3.11 e R.3.13, al termine del capitolo.



## Argomenti avanzati 3.1

### Invoke a constructor from another constructor

Esaminiamo la classe `BankAccount`, che ha due costruttori: uno senza parametri, per inizializzare il saldo a zero, e un altro per fornire un saldo iniziale. Invece di assegnare esplicitamente un valore alle variabili di esemplare, un costruttore può invocare un altro costruttore della stessa classe. Per questa operazione, esiste una notazione abbreviata:

```
public class BankAccount
{
    public BankAccount(double initialBalance)
    {
        balance = initialBalance;
    }

    public BankAccount()
    { }
```

```

{
    this(0);
}
...
}

```

L'enunciato `this(0);` significa: “Invoca un altro costruttore di questa classe fornendo il valore 0 come parametro di costruzione”. Un’invocazione di costruttore di questo tipo può comparire solamente *come prima riga di un altro costruttore*.

Questa sintassi rappresenta una comodità di poco conto, quindi non la useremo nel libro, perché l’utilizzo della parola riservata `this` può generare un po’ di confusione: solitamente `this` indica un riferimento al parametro implicito di un metodo, ma, se `this` è seguito da parentesi tonde, rappresenta l’invocazione di un altro costruttore della stessa classe.

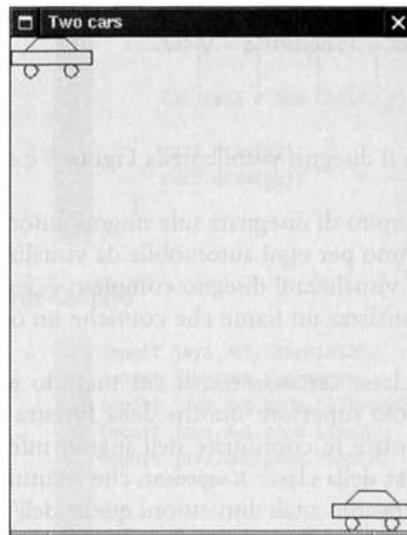
## 3.8 Classi per figure complesse

Creare una classe per ogni porzione  
di figura che possa essere utilizzata  
più volte è un’ottima idea.

In questo paragrafo proseguiamo la trattazione facoltativa degli argomenti grafici parlando di come organizzare in modo orientato agli oggetti il disegno di figure complesse.

Quando si scrive un programma che genera disegni articolati, composti di più parti, come quello di Figura 8, è bene creare una classe distinta per ogni porzione di figura. Tale classe dovrebbe avere un metodo `draw` che disegni la figura e un costruttore che ne definisca la posizione. Ecco, ad esempio, lo schema della classe `Car`, utilizzabile per disegnare un’automobile:

**Figura 8**  
Un componente che  
disegna due automobili



```

public class Car
{
    public Car(int x, int y)
    {
        // memorizza la posizione
    }
}

```

```

    }

    public void draw(Graphics2D g2)
    {
        // istruzioni per il disegno
        ...
    }
}

```

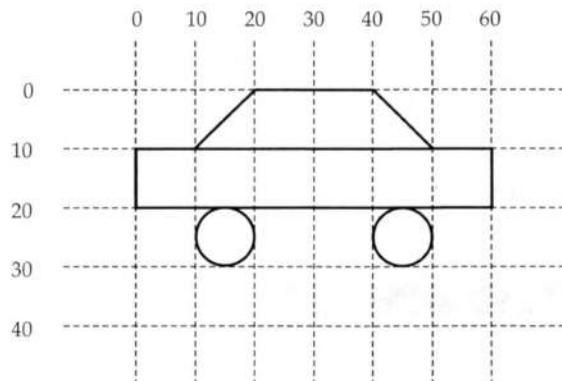
Al termine del paragrafo troverete la definizione completa della classe, nella quale noterete che il metodo `draw` contiene una lunga sequenza di istruzioni: disegna l'abitacolo, il tetto e le ruote.

**Per capire come disegnare una figura complessa, fatene uno schizzo su carta millimetrata.**

Le coordinate per le parti dell'automobile sembrano un po' casuali: per trovare i valori corretti conviene disegnare l'immagine su carta millimetrata e rilevarne le coordinate, come nella Figura 9.

**Figura 9**

Utilizzo di carta millimetrata per individuare le coordinate delle figure



Il programma che genera il disegno visibile nella Figura 8 è composto da tre classi:

- La classe `Car` ha il compito di disegnare una singola automobile. Vengono creati due oggetti di tale classe, uno per ogni automobile da visualizzare.
- La classe `CarComponent` visualizza il disegno completo.
- La classe `CarViewer` visualizza un frame che contiene un oggetto di tipo `CarComponent`.

Esaminiamo meglio la classe `CarComponent`, il cui metodo `paintComponent` disegna due automobili, una nell'angolo superiore sinistro della finestra e un'altra nel suo angolo inferiore destro. Per calcolare le coordinate dell'angolo inferiore destro invochiamo i metodi `getWidth` e `getHeight` della classe `JComponent`, che restituiscono le dimensioni di un componente grafico. Sottraiamo a tali dimensioni quelle dell'automobile, determinando così la posizione di `car2`:

```

Car car1 = new Car(0, 0);
int x = getWidth() - 60;
int y = getHeight() - 30;
Car car2 = new Car(x, y);

```

Osservate con attenzione l'invocazione di `getWidth` all'interno del metodo `paintComponent` di `CarComponent`. Non è presente alcun parametro implicito, per cui il metodo viene applicato al medesimo oggetto con cui si sta eseguendo il metodo `paintComponent`: il componente ottiene semplicemente *la propria* larghezza.

Eseguite il programma e ridimensionate la finestra: noterete che la seconda automobile si trova sempre nell'angolo in basso a destra della finestra, perché ogni volta che la finestra viene ridimensionata viene nuovamente invocato il metodo `paintComponent` e viene ricalcolata la posizione dell'automobile, in base alla nuova dimensione del componente.

### File CarComponent.java

```
1 import java.awt.Graphics;
2 import java.awt.Graphics2D;
3 import javax.swing.JComponent;
4
5 /**
6     Questo componente disegna due automobili stilizzate.
7 */
8 public class CarComponent extends JComponent
9 {
10    public void paintComponent(Graphics g)
11    {
12        Graphics2D g2 = (Graphics2D) g;
13
14        Car car1 = new Car(0, 0);
15
16        int x = getWidth() - 60;
17        int y = getHeight() - 30;
18
19        Car car2 = new Car(x, y);
20
21        car1.draw(g2);
22        car2.draw(g2);
23    }
24}
```

### File Car.java

```
1 import java.awt.Graphics2D;
2 import java.awt.Rectangle;
3 import java.awt.geom.Ellipse2D;
4 import java.awt.geom.Line2D;
5 import java.awt.geom.Point2D;
6
7 /**
8     Una forma di automobile da posizionare ovunque sullo schermo.
9 */
10 public class Car
11 {
12    private int xLeft;
13    private int yTop;
```

```

14 /**
15  * Costruisce un'automobile con l'angolo in alto a sinistra assegnato.
16  * @param x la coordinata x dell'angolo in alto a sinistra
17  * @param y la coordinata y dell'angolo in alto a sinistra
18 */
19 public Car(int x, int y)
20 {
21     xLeft = x;
22     yTop = y;
23 }
24 /**
25  * Disegna l'automobile.
26  * @param g2 il contesto grafico
27 */
28 public void draw(Graphics2D g2)
29 {
30     Rectangle body = new Rectangle(xLeft, yTop + 10, 60, 10);
31     Ellipse2D.Double frontTire
32         = new Ellipse2D.Double(xLeft + 10, yTop + 20, 10, 10);
33     Ellipse2D.Double rearTire
34         = new Ellipse2D.Double(xLeft + 40, yTop + 20, 10, 10);
35
36     // la base del parabrezza
37     Point2D.Double r1 = new Point2D.Double(xLeft + 10, yTop + 10);
38     // la parte anteriore del tettuccio
39     Point2D.Double r2 = new Point2D.Double(xLeft + 20, yTop);
40     // la parte posteriore del tettuccio
41     Point2D.Double r3 = new Point2D.Double(xLeft + 40, yTop);
42     // la base del lunotto posteriore
43     Point2D.Double r4 = new Point2D.Double(xLeft + 50, yTop + 10);
44
45     Line2D.Double frontWindshield = new Line2D.Double(r1, r2);
46     Line2D.Double roofTop = new Line2D.Double(r2, r3);
47     Line2D.Double rearWindshield = new Line2D.Double(r3, r4);
48
49     g2.draw(body);
50     g2.draw(frontTire);
51     g2.draw(rearTire);
52     g2.draw(frontWindshield);
53     g2.draw(roofTop);
54     g2.draw(rearWindshield);
55 }
56 }
57 }
58 }

```

### File CarViewer.java

```

1 import javax.swing.JFrame;
2
3 public class CarViewer
4 {
5     public static void main(String[] args)
6     {

```

```

7   JFrame frame = new JFrame();
8
9   frame.setSize(300, 400);
10  frame.setTitle("Two cars");
11  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12
13  CarComponent component = new CarComponent();
14  frame.add(component);
15
16  frame.setVisible(true);
17 }
18 }
```



## Auto-valutazione

27. Quale classe si deve modificare per visualizzare due automobili affiancate?
28. Quale classe si deve modificare per disegnare automobili con le ruote nere, e quale modifica occorre fare?
29. Come si fa a disegnare automobili di dimensione doppia?

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi E3.19 e E3.24, al termine del capitolo.



## Consigli pratici 3.2

### Disegnare forme grafiche

Si possono progettare programmi che visualizzino forme grafiche come automobili, alieni, grafici o qualsiasi altra immagine che si possa ottenere dalla combinazione di rettangoli, segmenti ed ellissi. Queste istruzioni delineano le fasi di una procedura per decomporre un disegno in parti, realizzando poi un programma che effettua il disegno completo.

**Problema.** Progettare un programma che disegni la bandiera di una nazione.

**Fase 1.** Determinare le forme che servono nel disegno completo

Potete usare le forme seguenti:

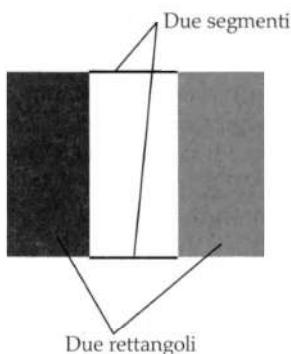
- Rettangoli e quadrati
- Ellissi e cerchi
- Segmenti

Si possono disegnare i profili di queste forme con qualsiasi colore, oppure se ne può colorare la parte interna, di nuovo con qualsiasi colore. Si può anche usare testo per etichettare parti del disegno.

Alcune bandiere nazionali sono composte da tre sezioni rettangolari di uguale ampiezza e di diversi colori, poste l'una accanto all'altra.



Potreste disegnare tale bandiera usando tre rettangoli. Se, però, il rettangolo centrale è bianco, come accade, ad esempio, nella bandiera dell'Italia (che ha i colori verde, bianco e rosso), è più semplice e di miglior effetto tracciare soltanto due segmenti orizzontali nella parte centrale, in alto e in basso:



### Fase 2. Trovare le coordinate per le forme

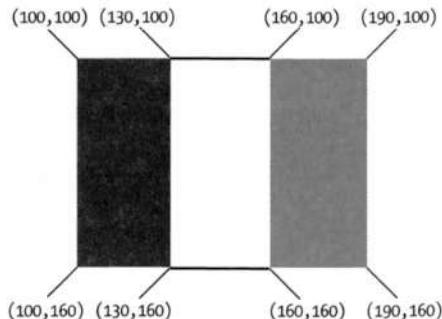
Ora occorre individuare le posizioni esatte per le singole figure geometriche.

- Per i rettangoli servono: le coordinate  $x$  e  $y$  dell'angolo superiore sinistro, la larghezza e l'altezza.
- Per le ellissi servono: per il rettangolo che le delimita, le coordinate  $x$  e  $y$  dell'angolo superiore sinistro, la larghezza e l'altezza.
- Per i segmenti servono: le coordinate  $x$  e  $y$  del punto iniziale e del punto finale.
- Per il testo servono: le coordinate  $x$  e  $y$  del punto base.

Una dimensione tipica per una finestra è, in pixel, 300 per 300. Probabilmente non vorrete avere la bandiera completamente spostata verso l'alto, per cui forse l'angolo superiore sinistro della bandiera dovrebbe trovarsi nel punto (100, 100).

Molte bandiere, come la bandiera dell'Italia, hanno un rapporto 3:2 tra larghezza e altezza (spesso è possibile trovare le proporzioni esatte per una particolare bandiera facendo un po' di ricerche in Internet, in uno dei tanti siti che si occupano di "bandiere del mondo"). Ad esempio, se volete che la bandiera sia larga 90 pixel, l'altezza dovrebbe essere pari a 60 pixel (perché non farla ampia 100 pixel? perché in tal caso l'altezza dovrebbe essere  $100 \cdot 2 / 3 = 67$ , che sembra essere più scomodo).

A questo punto possiamo calcolare le coordinate di tutti i punti significativi della figura:



### Fase 3. Scrivere gli enunciati Java per disegnare le forme

Nel nostro esempio ci sono due rettangoli e due segmenti:

```
Rectangle leftRectangle = new Rectangle(100, 100, 30, 60);
Rectangle rightRectangle = new Rectangle(160, 100, 30, 60);
Line2D.Double topLine = new Line2D.Double(130, 100, 160, 100);
Line2D.Double bottomLine = new Line2D.Double(130, 160, 160, 160);
```

Se siete più ambiziosi potete esprimere le coordinate in funzione di alcune variabili. Nel caso della bandiera, abbiamo scelto arbitrariamente la posizione dell'angolo superiore sinistro e la larghezza: tutte le altre coordinate discendono da tali scelte. Se decidete di seguire l'approccio ambizioso, allora i rettangoli e i segmenti vengono disegnati nel modo seguente:

```
Rectangle leftRectangle = new Rectangle(
    xLeft, yTop,
    width / 3, width * 2 / 3);
Rectangle rightRectangle = new Rectangle(
    xLeft + width * 2 / 3, yTop,
    width / 3, width * 2 / 3);
Line2D.Double topLine = new Line2D.Double(
    xLeft + width / 3, yTop,
    xLeft + width * 2 / 3, yTop);
Line2D.Double bottomLine = new Line2D.Double(
    xLeft + width / 3, yTop + width * 2 / 3,
    xLeft + width * 2 / 3, yTop + width * 2 / 3);
```

Ora bisogna colorare i rettangoli e disegnare i segmenti. Per la bandiera dell'Italia, il rettangolo sinistro è verde e quello destro è rosso. Ricordate di modificare i colori prima delle operazioni di riempimento e di disegno:

```
g2.setColor(Color.GREEN);
g2.fill(leftRectangle);
g2.setColor(Color.RED);
g2.fill(rightRectangle);
g2.setColor(Color.BLACK);
g2.draw(topLine);
g2.draw(bottomLine);
```

**Fase 4.** Combinare gli enunciati di disegno con il solito schema del componente

```
public class MyComponent extends JPanel
{
    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;
        // qui vanno inserite le istruzioni per disegnare
        . .
    }
}
```

Nel nostro esempio possiamo semplicemente aggiungere direttamente nel metodo `paintComponent` le istruzioni per disegnare tutte le forme:

```
public class ItalianFlagComponent extends JPanel
{
    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;
        Rectangle leftRectangle = new Rectangle(100, 100, 30, 60);
        . .
        g2.setColor(Color.GREEN);
        g2.fill(leftRectangle);
        . .
    }
}
```

Questo approccio è accettabile per disegni semplici, ma non è molto orientato agli oggetti: dopotutto, una bandiera è un oggetto. Sarebbe meglio creare una classe apposita per la bandiera, dopodiché si possono disegnare diverse bandiere in diverse posizioni e con diverse dimensioni, specificandone le dimensioni in un costruttore e fornendo un metodo `draw`:

```
public class ItalianFlag
{
    private int xLeft;
    private int yTop;
    private int width;

    public ItalianFlag(int x, int y, int aWidth)
    {
        xLeft = x;
        yTop = y;
        width = aWidth;
    }

    public void draw(Graphics2D g2)
    {
        Rectangle leftRectangle = new Rectangle(
            xLeft, yTop,
            width / 3, width * 2 / 3);
        . .
        g2.setColor(Color.GREEN);
        g2.fill(leftRectangle);
    }
}
```

```
    } . . .  
}
```

Serve, come prima, una classe separata per il componente, ma diventa molto semplice:

```
public class ItalianFlagComponent extends JComponent  
{  
    public void paintComponent(Graphics g)  
    {  
        Graphics2D g2 = (Graphics2D) g;  
        ItalianFlag flag = new ItalianFlag(100, 100, 90);  
        flag.draw(g2);  
    }  
}
```

#### Fase 5. Scrivere la classe che visualizza il disegno

Progettate una classe di visualizzazione, il cui metodo `main` costruisce un frame, vi aggiunge il vostro componente e rende visibile il frame stesso. Tale classe è praticamente sempre la stessa: per visualizzare un diverso componente dovete modificare una sola riga di codice.

```
public class ItalianFlagViewer  
{  
    public static void main(String[] args)  
    {  
        JFrame frame = new JFrame();  
  
        frame.setSize(300, 400);  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        ItalianFlagComponent component = new ItalianFlagComponent();  
        frame.add(component);  
  
        frame.setVisible(true);  
    }  
}
```

## Riepilogo degli obiettivi di apprendimento

### Variabili di esemplare e metodi che le usano

- Un oggetto usa variabili di esemplare per memorizzare i dati necessari per l'esecuzione dei propri metodi.
- Ciascun oggetto di una classe ha il proprio insieme di variabili di esemplare.
- Alle variabili di esemplare private possono accedere soltanto i metodi appartenenti alla medesima classe.
- L'incapsulamento prevede di nascondere i dettagli realizzativi, fornendo metodi per l'accesso ai dati.
- L'incapsulamento consente ai programmatore di usare una classe senza doverne conoscere i dettagli realizzativi.

- L'incapsulamento agevola l'individuazione di errori e la modifica di dettagli realizzativi di una classe.

#### **Intestazioni di metodi e costruttori per descrivere l'interfaccia pubblica di una classe**

- Per realizzare una classe, occorre prima individuare quali metodi siano necessari.
- I costruttori impostano i valori iniziali per i dati degli oggetti.
- Il nome di un costruttore è sempre uguale al nome della classe.
- Per descrivere le classi e i metodi pubblici dei programmi si usano i commenti di documentazione.
- Scrivete commenti di documentazione per ogni classe, ogni metodo, ogni parametro e ogni valore restituito.

#### **Implementazione di una classe**

- L'implementazione privata di una classe comprende le variabili di esemplare e il corpo di costruttori e metodi.

#### **Progetto di un collaudo che verifichi il corretto funzionamento di una classe**

- Un collaudo di unità verifica il corretto funzionamento di una classe a sé stante, senza che sia inserita in un programma completo.
- Per collaudare una classe si usa un ambiente interattivo oppure si scrive una classe di test che esegua istruzioni di collaudo.

#### **Tenere traccia dell'esecuzione per visualizzare il comportamento degli oggetti**

- Scrivete i metodi su una faccia di una scheda e le variabili di esemplare sull'altra faccia.
- Quando viene invocato un metodo modificatore, aggiornate le variabili di esemplare.

#### **Confronto tra periodo vitale e inizializzazione di variabili di esemplare, locali e parametro**

- Le variabili locali sono dichiarate nel corpo di un metodo.
- Quando un metodo termina la propria esecuzione, le sue variabili locali scompaiono.
- Le variabili di esemplare vengono inizializzate a un valore predefinito, mentre le variabili locali devono essere inizializzate esplicitamente.

#### **Uso del parametro implicito nella dichiarazione di metodi**

- L'uso, all'interno di un metodo, del nome di una variabile di esemplare rappresenta la variabile di esemplare del parametro隐式的.
- Il riferimento `this` rappresenta il parametro implicito.
- Una variabile locale mette in ombra una variabile di esemplare che abbia lo stesso nome: si può accedere a quest'ultima usando il riferimento `this`.
- L'invocazione di un metodo senza usare un parametro implicito agisce sul medesimo oggetto su cui si sta operando.

#### **Realizzazione di classi che disegnano forme complesse**

- Creare una classe per ogni porzione di figura che possa essere utilizzata più volte è un'ottima idea.
- Per capire come disegnare una figura complessa, fatene uno schizzo su carta millimetrata.

---

## **Esercizi di riepilogo e approfondimento**

- \* **R3.1.** Cos'è l'interfaccia pubblica di una classe? In cosa differisce dall'implementazione della classe stessa?

- \* **R3.2.** Cos'è l'incapsulamento? Perché è utile?
- \* **R3.3.** Le variabili di esemplare fanno parte dell'implementazione privata di una classe, ma sono visibili a quei programmati che abbiano a disposizione il codice sorgente della classe stessa. Spiegate in che senso la parola riservata `private` provvede a "nascondere" l'informazione.
- \* **R3.4.** Considerate una classe `Grade` che rappresenti un voto, in lettere, come A+ o B, secondo le consuetudini anglosassoni. Delineate due diverse scelte per le variabili di esemplare da usare per realizzarla.
- \*\* **R3.5.** Considerate una classe `Time` che rappresenti un istante di tempo, come le 9 del mattino (9 a.m., nel mondo anglosassone) o le 3.30 del pomeriggio (3.30 p.m.). Delineate due diversi insiemi di variabili di esemplare che si potrebbero usare per realizzarla.
- \* **R3.6.** Immaginate che il progettista della classe `Time` dell'esercizio precedente passi da una strategia di realizzazione all'altra, senza modificare l'interfaccia pubblica. Cosa devono fare i programmati che utilizzano la classe `Time`?
- \*\* **R3.7.** La variabile di esemplare `value` della classe `Counter` può essere ispezionata mediante il metodo d'accesso `getValue`. Dovrebbe essere presente un metodo `setValue` per modificare tale variabile? Motivate la risposta.
- \*\* **R3.8.**
  - Spiegate perché il costruttore `BankAccount(double initialBalance)` non è strettamente necessario. Conseguentemente, se lo togliamo dall'interfaccia pubblica, come si può ottenere un oggetto di tipo `BankAccount` con il saldo iniziale voluto?
  - Al contrario, potremmo togliere il costruttore `BankAccount()` e mettere a disposizione soltanto `BankAccount(double initialBalance)`?
- \*\* **R3.9.** Perché la classe `BankAccount` non dispone di un metodo `reset`, che ne riporti lo stato alle condizioni iniziali?
- \* **R3.10.** Cosa succede, nella nostra realizzazione della classe `BankAccount`, quando da un conto bancario viene prelevata una quantità di denaro maggiore del saldo disponibile in quel momento?
- \*\* **R3.11.** Cos'è il riferimento `this`? Per quale motivo lo usereste?
- \*\* **R3.12.** Tra i metodi della classe `CashRegister` vista nella sezione Esempi completi 3.1, quali sono metodi d'accesso e quali, invece, metodi modificatori?
- \*\* **R3.13.** Cosa fa il metodo seguente? Fornite un esempio di come si potrebbe invocare il metodo.

```
public class BankAccount
{
    public void mystery(BankAccount that, double amount)
    {
        this.balance = this.balance - amount;
        that.balance = that.balance + amount;
    }
    . . . // gli altri metodi del conto bancario
}
```

- \*\* **R3.14.** Immaginate di voler realizzare la classe `TimeDepositAccount` che rappresenti un conto bancario avente un tasso di interesse attivo fisso, il cui valore viene determinato al momento della costruzione dell'oggetto, insieme al saldo iniziale. Progettate un metodo che fornisca il valore del saldo. Progettate, poi, un metodo che accrediti sul conto gli interessi maturati, senza aver bisogno di parametri, dal momento che il tasso di interesse è noto, e senza restituire alcunché, visto che

esiste un altro metodo che fornisce il valore del saldo. In un conto di questo tipo non è possibile versare ulteriori fondi dopo l'apertura. Infine, occorre progettare un metodo `withdraw` che preleva l'intero ammontare del saldo: non sono consentiti prelievi parziali.

- \* **R3.15.** Analizzate questa classe, che vuole rappresentare un modello di forme quadrate:

```
public class Square
{
    private int sideLength;
    private int area; // non è una buona idea

    public Square(int length)
    {
        sideLength = length;
    }

    public int getArea()
    {
        area = sideLength * sideLength;
        return area;
    }
}
```

Perché non è una buona idea usare una variabile di esemplare per memorizzare l'area del quadrato? Modificate la classe in modo che `area` sia una variabile locale.

- \*\* **R3.16.** Analizzate questa classe, che vuole rappresentare un modello di forme quadrate:

```
public class Square
{
    private int sideLength;
    private int area;

    public Square(int initialLength)
    {
        sideLength = initialLength;
        area = sideLength * sideLength;
    }

    public int getArea() { return area; }
    public void grow() { sideLength = 2 * sideLength; }
}
```

Che errore c'è in questa classe? Come lo correggereste?

- \*\* **R3.17 (collaudo).** Progettate una classe per il collaudo di unità della classe `Counter` vista nel Paragrafo 3.1.
- \*\* **R3.18 (collaudo).** Leggete l'esercizio E3.12, senza implementare, per il momento, la classe `Car`. Scrivete una classe di collaudo che simuli uno scenario di questo tipo: si aggiunge carburante all'automobile, si guida un po', si aggiunge di nuovo un po' di carburante e, infine, si guida di nuovo. Visualizzate la quantità di carburante presente nel serbatoio, oltre al suo valore previsto.
- \* **R3.19.** Usando la tecnica di analisi descritta nel Paragrafo 3.5, studiate il programma presentato alla fine del Paragrafo 3.4.
- \*\* **R3.20.** Usando la tecnica di analisi descritta nel Paragrafo 3.5, studiate il programma presentato nella sezione Consigli pratici 3.1.

- ★★ **R3.21.** Usando la tecnica di analisi descritta nel Paragrafo 3.5, studiate il programma presentato nella sezione Esempi completi 3.1.
- ★★★ **R3.22.** Progettate una modifica della classe `BankAccount` che la porti ad avere questo comportamento: ogni mese, le prime cinque transazioni sono gratuite, mentre ciascuna delle successive comporta un addebito di \$1. Aggiungete un metodo che addebiti complessivamente tali commissioni alla fine di ogni mese. Quali ulteriori variabili di esemplare vi servono? Usando la tecnica di analisi descritta nel Paragrafo 3.5, studiate uno scenario che evidenzi come vengono calcolate le commissioni su un arco temporale di due mesi.
- ★★ **R3.23 (grafica).** Nell'ipotesi di voler estendere il programma visualizzatore di automobili visto nel Paragrafo 3.8 in modo che mostri uno scenario urbano, con diverse automobili e case, di quali classi avreste bisogno?
- ★★★ **R3.24 (grafica).** Spiegate per quale motivo nella classe `CarComponent` le invocazioni dei metodi `getWidth` e `getHeight` non hanno parametri espliciti.
- ★★ **R3.25 (grafica).** Come modifichereste la classe `Car` per poter visualizzare automobili di dimensioni diverse?

## Esercizi di programmazione

---

- ★ **E3.1.** Vogliamo aggiungere al conta-persone visto nel Paragrafo 3.1 un pulsante che consenta di annullare il conteggio di una persona avvenuto per sbaglio (operazione “annulla” o, in inglese, *undo*). Progettate il metodo

```
public void undo()
```

che simuli l'effetto di tale nuovo pulsante. A titolo precauzionale, garantite che non ci siano effetti anomali nel caso in cui la pressione del pulsante “annulla” sia più frequente di quella del pulsante di conteggio (*suggerimento*: l'invocazione `Math.max(n, 0)` restituisce il valore di `n` se `n` è maggiore di zero, altrimenti restituisce zero).

- ★ **E3.2.** Dovete simulare l'utilizzo di un conta-persone in uno scenario in cui il numero di persone che possono entrare in un luogo è limitato. Per prima cosa si imposta il limite massimo di persone, invocando il metodo

```
public void setLimit(int maximum)
```

Se il pulsante di conteggio viene premuto dopo il superamento del limite impostato, non ha alcun effetto sul conteggio memorizzato nel dispositivo (*suggerimento*: l'invocazione `Math.min(n, limit)` restituisce il valore di `n` se `n` è minore di `limit`, altrimenti restituisce `limit`).

- ★★ **E3.3.** Simulate un circuito elettrico che controlla la lampada di un corridoio, con interruttori alle due estremità del corridoio stesso. Ciascun interruttore può trovarsi in una di due posizioni (su o giù, *up* o *down*) e la lampada può essere accesa o spenta (*on* o *off*). Il cambiamento di stato di uno qualsiasi dei due interruttori provoca il cambiamento di stato della lampada. Progettate i metodi:

```
public int getFirstSwitchState() // ispeziona il primo interruttore, 0 = giù, 1 = su
public int getSecondSwitchState() // ispeziona il secondo interruttore
public int getLampState() // ispeziona la lampada, 0 = spenta, 1 = accesa
public void toggleFirstSwitch() // cambia lo stato del primo interruttore
public void toggleSecondSwitch() // cambia lo stato del secondo interruttore
```

- \* **E3.4 (collaudo).** Scrivete la classe `CircuitTester` che collaudi tutte le possibili combinazioni di interruttori previste dall'esercizio precedente, visualizzando lo stato previsto e effettivo della lampada e di ciascun interruttore.
- \*\*\* **E3.5.** Modificate l'interfaccia pubblica della classe dell'esercizio E3.3, che rappresenta un circuito elettrico, in modo che metta a disposizione i metodi seguenti:

```
public int getSwitchState(int switch) // ispeziona un interruttore
public int getLampState()
public void toggleSwitch(int switch) // cambia lo stato di un interruttore
```

Risolvete il problema usando soltanto le caratteristiche del linguaggio che sono state presentate fino ad ora: l'aspetto complicato del progetto è l'individuazione di una rappresentazione dei dati che consenta di ricostruire lo stato in cui si trovano i due interruttori.

- \* **E3.6 (collaudo).** Scrivete la classe `BankAccountTester` il cui metodo `main` costruisca un conto bancario, effettui un versamento di \$1000 seguito dai prelievi di \$500 e \$400 e, infine, visualizzi il saldo rimanente, seguito dal suo valore previsto.
- \* **E3.7.** Aggiungete alla classe `BankAccount` un metodo

```
public void addInterest(double rate)
```

che aggiunga al saldo del conto gli interessi, calcolati con il tasso fornito come parametro. Ad esempio, dopo l'esecuzione di questi enunciati

```
BankAccount momSavings = new BankAccount(1000);
momSavings.addInterest(10); // interessi al 10%
```

il saldo di `momSavings` è \$1100. Progettate anche la classe `BankAccountTester` che visualizzi il saldo finale e il suo valore previsto.

- \* **E3.8.** Scrivete la classe `SavingsAccount` (*conto di risparmio*), del tutto simile alla classe `BankAccount`, tranne per la presenza di un'ulteriore variabile di esemplare, `interest`. Fornite un costruttore che assegna un valore sia al saldo iniziale sia al tasso di interesse. Fornite anche un metodo, `addInterest` (privo di parametri esplicativi), che accrediti sul conto gli interessi maturati, calcolati applicando il tasso d'interesse `interest` al saldo del conto nel momento in cui il metodo viene invocato. Scrivete, poi, la classe `SavingsAccountTester` che costruisca uno di tali "conti di risparmio" con saldo iniziale di \$1000 e tasso di interesse del 10%. Applicate, infine, per cinque volte il metodo `addInterest`, visualizzando il saldo finale e il suo valore previsto, dopo averlo calcolato a mano.

- \*\*\* **E3.9.** Aggiungete alla classe `CashRegister` il metodo `printReceipt` che visualizzi i prezzi di ciascun articolo acquistato e l'importo totale dovuto dal cliente. *Suggerimento:* dovete costruire una stringa che contenga tutte le informazioni, usando il metodo `concat` della classe `String` per aggiungervi i singoli articoli, uno dopo l'altro; per trasformare un prezzo in una stringa usate l'invocazione `String.valueOf(prezzo)`.
- \* **E3.10.** Dopo la chiusura, il gestore di un negozio vorrebbe conoscere il volume totale di vendite effettuate nella giornata: modificate la classe `CashRegister` in modo che sia in grado di farlo, dotandola dei metodi aggiuntivi `getSalesTotal` e `getSalesCount`, che restituiscono, rispettivamente, l'incasso totale e il numero totale di scontrini emessi, oltre al metodo `reset` che azzera tutti i contatori utilizzati, in modo che le operazioni funzionino correttamente il giorno successivo.
- \*\* **E3.11.** Realizzate la classe `Employee` (*dipendente*). Ogni dipendente ha un nome (una stringa) e uno stipendio (di tipo `double`). Scrivete un costruttore con due parametri:

```
public Employee(String employeeName, double currentSalary)
```

e i metodi:

```
public String getName()
public double getSalary()
public void raiseSalary(double byPercent)
```

Tali metodi forniscono, nell'ordine, il nome e lo stipendio del dipendente e ne aumentano il salario della percentuale indicata. Ecco un esempio di utilizzo:

```
Employee harry = new Employee("Hacker, Harry", 50000);
harry.raiseSalary(10); // Harry ottiene un aumento di stipendio del 10%
```

Progettate anche la classe `EmployeeTester` che collaudi tutti i metodi.

- ★★ **E3.12.** Realizzate la classe `Car (automobile)`, con queste proprietà. Un'automobile ha una determinata resa del carburante (misurata in miglia/gallone o in litri/chilometro, a vostra scelta) e una certa quantità di carburante nel serbatoio. La resa è specificata nel costruttore e inizialmente il serbatoio è vuoto. Progettate: un metodo `drive` per simulare il percorso di un'automobile per una data distanza, riducendo conseguentemente il livello di carburante nel suo serbatoio; un metodo `getGasInTank`, per ispezionare il livello del carburante nel serbatoio; un metodo `addGas`, per fare rifornimento. Ecco un esempio di utilizzo:

```
Car myHybrid = new Car(50); // 50 miglia per gallone
myHybrid.addGas(20); // aggiungi 20 galloni di carburante
myHybrid.drive(100); // viaggia per 100 miglia
double gasLeft = myHybrid.getGasInTank(); // carburante rimasto
```

Potete ipotizzare che il metodo `drive` non venga mai invocato per una distanza maggiore di quella percorribile con il carburante disponibile. Progettate anche la classe `CarTester` che collaudi tutti i metodi.

- \* **E3.13.** Realizzate la classe `Product (prodotto)`. Ciascun prodotto ha un nome e un prezzo, descritti nel costruttore: `new Product("Toaster", 29.95)`. Progettate i seguenti metodi: `getName`, per conoscere il nome del prodotto; `getPrice`, per conoscerne il prezzo; `reducePrice`, per ridurne il prezzo. Scrivete un programma che crei due prodotti e ne visualizzi il nome e il prezzo, per poi ridurne i prezzi di \$5.00 e visualizzarli nuovamente.
- ★★ **E3.14.** Realizzate una classe che impagini una semplice lettera. Il costruttore riceve come parametri il nome del mittente (*from*) e quello del destinatario (*to*):

```
public Letter(String from, String to)
```

Progettate un metodo per aggiungere una riga di testo al contenuto della lettera, in fondo:

```
public void addLine(String line)
```

Progettate un altro metodo che restituisca l'intero testo della lettera:

```
public String getText()
```

Il testo della lettera ha il seguente formato:

```
Dear destinatario:  
riga vuota  
prima riga del contenuto della lettera  
seconda riga del contenuto della lettera
```

*ultima riga del contenuto della lettera*

*riga vuota*

*Sincerely,*

*riga vuota*

*mittente*

Progettate anche il programma di collaudo `LetterPrinter` che visualizzi questa lettera:

Dear John:

I am sorry we must part.  
I wish you all the best.

Sincerely,

Mary

In esso, costruite un esemplare della classe `Letter` e invocate due volte il metodo `addLine`.

*Suggerimenti:* (1) Usate il metodo `concat` per costruire una stringa più lunga a partire da due stringhe più corte. (2) La stringa speciale "`\n`" rappresenta il carattere speciale (chiamato *newline*) che si usa per andare a capo. Ad esempio, il seguente enunciato

```
body = body.concat("Sincerely,").concat("\n");
```

aggiunge al contenuto della lettera la riga "Sincerely,".

- \*\* **E3.15.** Realizzate la classe `Bug` che rappresenti un insetto che si sposta lungo una linea orizzontale, verso sinistra o verso destra. Inizialmente si sposta verso destra, ma può cambiare direzione; ogni volta che si sposta, la sua posizione lungo la linea cambia di un'unità verso la direzione più recente. Dotate la classe di un costruttore:

```
public Bug(int initialPosition) // riceve la posizione iniziale
```

e dei metodi:

```
public void turn() // cambia direzione
public void move() // si sposta di un'unità nella direzione attuale
public int getPosition() // ispeziona la posizione
```

Ecco un esempio di utilizzo:

```
Bug bugsy = new Bug(10);
bugsy.move(); // ora si trova nella posizione 11
bugsy.turn(); // cambia direzione
bugsy.move(); // ora si trova nella posizione 10
```

La classe `BugTester` deve costruire un insetto, farlo muovere e girare alcune volte, poi visualizzarne la posizione effettiva e quella prevista.

- \*\* **E3.16.** Realizzate una classe `Moth` che rappresenti una falena che si sposta lungo una linea retta, ricordando la propria posizione e la distanza da un'origine prefissata. Quando si sposta verso una sorgente luminosa, la sua nuova posizione viene a trovarsi a metà strada tra quella precedente e la posizione della sorgente luminosa. Dotate la classe di un costruttore:

```
public Moth(double initialPosition) // riceve la posizione iniziale
```

e dei metodi:

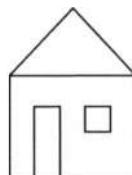
```
public void moveToLight(double lightPosition) // va verso la luce indicata
public double getPosition() // ispeziona la posizione
```

La classe `MothTester` deve costruire una falena, farla muovere verso un paio di sorgenti luminose, poi visualizzarne la posizione effettiva e quella prevista.

- \*\*\* **E3.17 (grafica).** Scrivete un programma che riempia una finestra con una grande ellisse, avente il contorno nero e la superficie interna del colore che preferite. L'ellisse deve avere la finestra come rettangolo di delimitazione, anche se questa viene ridimensionata: all'interno del metodo `paintComponent` invocate i metodi `getWidth` e `getHeight` della classe `JComponent`.
- \*\* **E3.18 (grafica).** Disegnate un bersaglio ("bull's eye"), vale a dire una serie di cerchi concentrici, alternando i colori bianco e nero. Il programma deve essere costituito dalle classi `Target` (*bersaglio*), `TargetComponent` e `TargetViewer`. Suggerimento: Colorate un cerchio nero nella sua intera superficie, quindi sovrapponete un cerchio bianco più piccolo, e così via.



- \*\* **E3.19 (grafica).** Scrivete un programma che tracci lo schizzo di una casa, semplice come la figura qui riportata oppure più elaborata, se preferite (potete utilizzare una vista prospettica, disegnare un grattacielo, colonne di marmo nell'ingresso o qualsiasi cosa). Progettate la classe `House` e dotatela del metodo `draw(Graphics2D g2)` che disegni la casa.



- \*\* **E3.20 (grafica).** Migliorate l'esercizio precedente dotando la classe `House` di un costruttore che consenta di specificare posizione e dimensione della casa da visualizzare, quindi popolate lo schermo con un po' di case di varie dimensioni.
- \*\* **E3.21 (grafica).** Modificate il programma che disegna automobili visto nel Paragrafo 3.8 in modo che le automobili vengano visualizzate in colori diversi. Ogni oggetto di tipo `Car` deve memorizzare il proprio colore. Scrivete le classi `Car` e `CarComponent` modificate.
- \*\* **E3.22 (grafica).** Modificate la classe `Car` in modo che la dimensione di un'automobile possa essere specificata nel costruttore. Modificate la classe `CarComponent` in modo che una delle automobili abbia dimensioni doppie rispetto alla versione originale.
- \*\* **E3.23 (grafica).** Scrivete un programma che disegni la stringa "HELLO" utilizzando solo linee e cerchi. Non invocate il metodo `drawString` e non usate `System.out`. Progettate le classi `LetterH`, `LetterE`, `LetterL` e `LetterO`.
- \*\* **E3.24 (grafica).** Scrivete un programma per visualizzare gli anelli olimpici. Colorate gli anelli con i colori corretti (blu, nero, rosso, giallo e verde, procedendo da sinistra a destra e dall'alto in basso), progettando le classi `OlympicRing`, `OlympicRingViewer` e `OlympicRingComponent`.



\*\* E3.25 (grafica). Costruite un grafico a barre per rappresentare i dati seguenti:

Nome del ponte	Campata maggiore (misurata in piedi)
Golden Gate	4200
Brooklyn	1595
Delaware Memorial	2150
Mackinac	3800

Disponete le barre orizzontalmente, per facilitare l'inserimento delle etichette in corrispondenza di ciascuna barra, e progettate le classi `BarChartViewer` e `BarChartComponent`.

Sul sito web dedicato al libro si trova una raccolta di progetti di programmazione più complessi.

# 4

## Tipi di dati fondamentali



### Obiettivi del capitolo

---

- Apprendere l'utilizzo di numeri interi e di numeri in virgola mobile
- Individuare i limiti dei tipi numerici
- Acquisire consapevolezza degli errori di trabocco e di arrotondamento
- Apprendere il corretto utilizzo delle costanti
- Scrivere espressioni aritmetiche in Java
- Usare il tipo `String` per manipolare sequenze di caratteri
- Imparare ad acquisire dati e a visualizzare dati impaginati

Questo capitolo insegna a manipolare numeri e sequenze di caratteri, con l'obiettivo di raggiungere una salda conoscenza di tali tipi di dati fondamentali nei programmi scritti in Java. Vedrete come manipolare numeri e stringhe, per scrivere programmi semplici ma utili. Infine, tratteremo l'importante argomento dell'acquisizione dei dati in ingresso e della loro presentazione in uscita, per consentirvi di realizzare programmi interattivi.

## 4.1 Numeri

Iniziamo questo capitolo parlando di numeri: i paragrafi che seguono vi spiegheranno come scegliere i tipi di numeri più appropriati per i valori numerici che dovete elaborare; inoltre, imparerete a lavorare con le costanti, cioè valori numerici che non cambiano.

### 4.1.1 Tipi di numeri

Java ha otto tipi primitivi, fra i quali quattro tipi numerici interi e due tipi numerici in virgola mobile.

Ogni valore, in Java, è un riferimento a un oggetto oppure appartiene a uno degli otto **tipi primitivi** elencati nella Tabella 1.

Sei di tali tipi primitivi sono numerici, quattro dei quali rappresentano numeri interi e due rappresentano numeri in virgola mobile.

Ciascuno dei tipi numerici interi ha un diverso intervallo di variabilità (l'Appendice D spiega per quale motivo i limiti degli intervalli siano correlati a potenze di due). Il più grande numero rappresentabile con un valore di tipo int si indica con `Integer.MAX_VALUE` ed è circa 2.14 miliardi, mentre il valore più piccolo è `Integer.MIN_VALUE`, circa -2.14 miliardi.

**Tabella 1**  
Tipi primitivi

Tipo	Descrizione	Dimensione
int	Tipo intero con intervallo -2147483648 ( <code>Integer.MIN_VALUE</code> )...2147483647 ( <code>Integer.MAX_VALUE</code> ) (circa 2 miliardi)	4 byte
byte	Tipo che descrive un singolo byte, con intervallo -128...127	1 byte
short	Tipo intero "corto", con intervallo -32768...32767	2 byte
long	Tipo intero "lungo", con intervallo -9223372036854775808... 9223372036854775807	8 byte
double	Tipo in virgola mobile a doppia precisione, con intervallo circa $\pm 10^{308}$ e circa 15 cifre decimali significative	8 byte
float	Tipo in virgola mobile a singola precisione, con intervallo circa $\pm 10^{38}$ e circa 7 cifre decimali significative	4 byte
char	Tipo che rappresenta caratteri codificati secondo lo schema Unicode (Computer e società 4.2)	2 byte
boolean	Tipo per i due valori logici true e false (Capitolo 5)	1 bit

Quando, in un programma Java, compare un valore numerico come 6 o 0.335, lo chiamiamo **numero letterale** (*number literal*). Se un numero letterale ha il punto decimale si tratta di un numero in virgola mobile, altrimenti è un numero intero. La Tabella 2 mostra come si possano scrivere numeri interi e in virgola mobile in Java.

In generale, per rappresentare numeri interi useremo il tipo di dato int, anche se, a volte, i calcoli che coinvolgono numeri interi possono dar luogo a un fenomeno chiamato *trabocco* ("overflow"). Ciò accade quando il risultato di un calcolo non rientra nell'intervalle di variabilità del tipo numerico, come in questo esempio:

```
int n = 1000000;
System.out.println(n * n); // Visualizza -727379968, risultato ovviamente errato
```

Un calcolo numerico trabocca se il risultato esce dall'intervallo caratteristico del tipo numerico.

**Tabella 2**  
Numeri letterali in Java

Numero	Tipo	Commento
6	int	Un numero intero non ha parte frazionaria.
-6	int	I numeri interi possono essere negativi.
0	int	Zero è un numero intero.
0.5	double	Un numero con parte frazionaria è di tipo double.
1.0	double	Un numero intero con parte frazionaria .0 è di tipo double.
1E6	double	Un numero in notazione esponenziale: $1 \times 10^6$ o 1000000. I numeri in notazione esponenziale sono sempre di tipo double.
2.96E-2	double	Eponente negativo: $2.96 \times 10^{-2} = 2.96 / 100 = 0.0296$ .
10000L	long	Il suffisso L indica un letterale “lungo”.
 100,000		<b>Errore:</b> come separatore decimale non si può usare la virgola.
 100_000	int	Nei numeri letterali si possono usare i caratteri di sottolineatura, per chiarezza.
 3 1/2		<b>Errore:</b> non si possono usare frazioni, bisogna usare la notazione decimale, in questo caso 3.5.

Il prodotto  $n * n$  vale  $10^{12}$ , un valore maggiore del massimo numero intero rappresentabile, che è circa  $2 \times 10^9$ . Il risultato viene quindi troncato per trovar posto in un int, generando un valore completamente errato: sfortunatamente, quando avvengono fenomeni di trabocco nell’elaborazione di numeri interi non viene data nessuna segnalazione d’errore.

Se incorrette in questo problema, la soluzione più semplice consiste nell’utilizzare il tipo long. La sezione Argomenti avanzati 4.1 vi mostrerà come usare il tipo a precisione arbitraria BigInteger nella rara circostanza in cui anche il tipo long dia luogo a trabocco.

Solitamente il trabocco non si manifesta quando si elaborano numeri in virgola mobile in doppia precisione, perché il tipo double ha un intervallo di variabilità di circa  $\pm 10^{308}$ . Il problema dei numeri in virgola mobile è diverso: la precisione limitata. Il tipo double ha circa 15 cifre decimali significative, e sono molti i valori che non possono essere rappresentati adeguatamente con tale tipo di dato.

Quando un valore non può essere rappresentato in modo esatto, viene arrotondato al valore più vicino, come in questo esempio:

```
double f = 4.35;
System.out.println(100 * f); // Visualizza 434.99999999999994
```

Quando non è possibile trovare una rappresentazione numerica esatta di un numero in virgola mobile si ha un errore di arrotondamento.

Questo problema è dovuto al fatto che i calcolatori rappresentano i numeri con il sistema di numerazione binario, nel quale non esiste una rappresentazione esatta della frazione 1/10, esattamente come nel sistema di numerazione decimale non esiste una rappresentazione esatta della frazione 1/3 (troverete maggiori informazioni nell’Appendice D).

Per questo motivo il tipo double non è adatto alle elaborazioni finanziarie. In questo libro continueremo a usare valori di tipo double per i saldi bancari e per altre quantità di tipo finanziario, al fine di semplificare al massimo i programmi qui presentati, ma programmi professionali devono usare, per tali ambiti, il tipo BigDecimal (descritto nella sezione Argomenti avanzati 4.1).

In Java è perfettamente lecito assegnare un valore numerico intero a una variabile in virgola mobile:

```
int dollars = 100;
double balance = dollars; // va bene
```

ma l'assegnazione inversa è errata. Non si può assegnare un valore in virgola mobile a una variabile intera:

```
double balance = 13.75;
int dollars = balance; // ERRORE
```

Nella parte conclusiva del Paragrafo 4.2.5 vedrete come convertire un valore di tipo `double` in un valore numerico intero.

In questo libro non useremo il tipo di dato `float`: ha meno di 7 cifre significative, cosa che aumenta notevolmente il rischio di errori di arrotondamento. Alcuni programmati usano il tipo `float` per risparmiare memoria quando hanno bisogno di memorizzare un'enorme insieme di numeri, per i quali la richiesta di precisione non sia molto elevata.

### 4.1.2 Costanti

In molti programmi si usano **costanti** numeriche, cioè valori che non vengono modificati e che hanno un significato ben preciso all'interno dell'elaborazione.

Come esempio tipico di elaborazione che coinvolge valori costanti consideriamo i valori delle monete, come nel caso seguente:

```
payment = dollars + quarters * 0.25 + dimes * 0.1
          + nickels * 0.05 + pennies * 0.01;
```

La maggior parte del codice si documenta da sé, ma i quattro valori numerici compaiono nell'espressione aritmetica senza alcuna spiegazione: 0.25, 0.1, 0.05 e 0.01. Ovviamente in questo caso tutti gli americani sanno che il valore di un "nickel" è cinque centesimi, giustificando così il numero 0.05, e così via, ma la persona che modificherà questo codice potrebbe vivere in un altro Paese e potrebbe non sapere che un "nickel" vale cinque centesimi di dollaro.

Per questo motivo è bene usare nomi simbolici per tutti i valori, anche quelli che sembrano ovvi. Ecco una versione più chiara del calcolo precedente:

```
double quarterValue = 0.25;
double dimeValue = 0.1;
double nickelValue = 0.05;
double pennyValue = 0.01;
return dollars + quarters * quarterValue + dimes * dimeValue
          + nickels * nickelValue + pennies * pennyValue;
```

C'è un altro miglioramento che possiamo apportare, osservando che esiste una differenza tra le variabili `nickels` e `nickelValue`: la variabile `nickels` può veramente assumere valori diversi nel corso dell'esecuzione del programma, quando calcoliamo diversi pagamenti, ma `nickelValue` vale sempre 0.05.

Una variabile `final` è una costante:  
dopo aver ricevuto un valore, non può  
più essere modificata.

Date un nome alle costanti,  
per rendere i vostri programmi  
più facili da leggere e da modificare.

In Java le costanti sono identificate dalla parola riservata `final`. Dopo aver ricevuto il suo valore iniziale, una variabile con l'attributo `final` non può più essere modificata: se provate a modificarne il valore, il compilatore segnalerà un errore e il vostro programma non verrà compilato.

Molti programmatore usano per le costanti (cioè per le variabili `final`) nomi con tutte le lettere maiuscole, come `NICKEL_VALUE`, perché così si distinguono facilmente dalle variabili vere e proprie, i cui nomi hanno lettere in maggior parte minuscole. Anche in questo libro useremo questa convenzione, ma è soltanto un buono stile di programmazione, non un requisito del linguaggio Java: il compilatore non protesterà se date a una variabile `final` un nome con lettere minuscole.

Ecco una versione migliorata del codice che calcola l'importo di un pagamento:

```
final double QUARTER_VALUE = 0.25;
final double DIME_VALUE = 0.1;
final double NICKEL_VALUE = 0.05;
final double PENNY_VALUE = 0.01;
return dollars + quarters * QUARTER_VALUE + dimes * DIME_VALUE
    + nickels * NICKEL_VALUE + pennies * PENNY_VALUE;
```

Spesso le costanti vengono usate in più metodi, per cui è comodo dichiararle insieme alle variabili di esemplare della classe, aggiungendo l'attributo `static` a quello `final`. Di nuovo, `final` significa che il valore è costante, mentre il significato della parola riservata `static` verrà spiegato con maggiore dettaglio nel Capitolo 8: in estrema sintesi, significa che la costante appartiene alla classe.

```
public class CashRegister
{
    // costanti
    public static final double QUARTER_VALUE = 0.25;
    public static final double DIME_VALUE = 0.1;
    public static final double NICKEL_VALUE = 0.05;
    public static final double PENNY_VALUE = 0.01;

    // variabili di esemplare
    private double purchase;
    private double payment;

    // metodi
    ...
}
```

Abbiamo qui definito le costanti con l'attributo `public`: ciò non rappresenta un pericolo, perché le costanti non possono essere modificate. Metodi di altre classi possono accedere a una costante pubblica indicando il nome della classe in cui essa è dichiarata, seguito da un punto e dal nome della costante stessa, come `CashRegister.NICKEL_VALUE`.

La classe `Math` della libreria standard definisce un paio di utili costanti:

```
public class Math
{
    ...
    public static final double E = 2.7182818284590452354;
    public static final double PI = 3.14159265358979323846;
}
```

## Sintassi di Java

### 4.1 Dichiarazione di costante

#### Sintassi

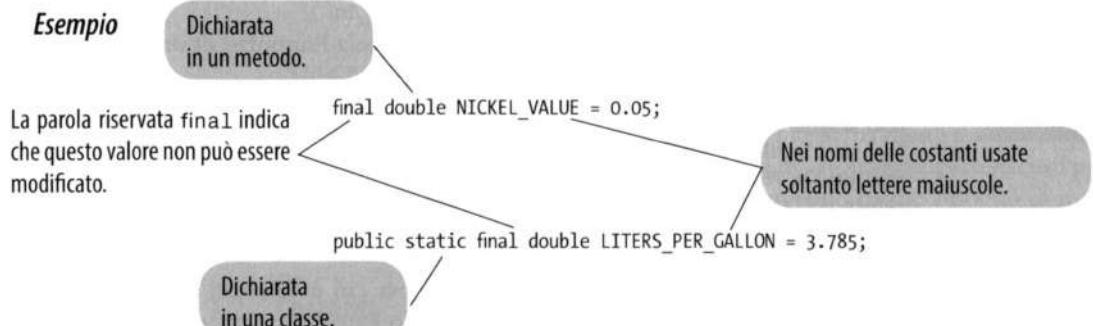
Dichiarata in un metodo:

```
final nomeTipo nomeVariabile = espressione;
```

Dichiarata in una classe:

```
modalitàDiAccesso static final nomeTipo nomeVariabile = espressione;
```

#### Esempio



In qualunque vostro metodo potete fare riferimento a queste costanti, `Math.PI` e `Math.E`. Ad esempio

```
double circumference = Math.PI * diameter;
```

Il programma che trovate come esempio al termine di questo paragrafo usa le costanti e propone una versione migliorata della classe `CashRegister` che avete visto nella sezione Consigli pratici 3.1: la sua interfaccia pubblica è stata modificata per risolvere un problema commerciale molto comune.

I cassieri molto indaffarati compiono spesso errori nel sommare monete: la nostra classe `CashRegister` mette ora a disposizione un metodo che ha come valori di ingresso *conteggi di monete*. Ad esempio, l'invocazione

```
register.receivePayment(1, 2, 1, 1, 4);
```

registra un pagamento che consiste, nell'ordine da sinistra a destra, di un dollaro, due quarti di dollaro (“quarter”), un decimo di dollaro (“dime”), un “nickelino” (“nickel”) e quattro centesimi (“penny”). Il metodo `receivePayment` calcola l’importo totale pagato, cioè 1.69 dollari. Come potete vedere esaminando il codice, il metodo usa costanti per rappresentare i valori delle singole monete.

#### File `CashRegister.java`

```
1 /**
2  * Un registratore di cassa somma i prezzi e calcola il resto dovuto.
3 */
```

```
public class CashRegister
{
    public static final double QUARTER_VALUE = 0.25;
    public static final double DIME_VALUE = 0.1;
    public static final double NICKEL_VALUE = 0.05;
    public static final double PENNY_VALUE = 0.01;

    private double purchase;
    private double payment;

    /**
     * Costruisce un registratore di cassa senza soldi nel cassetto.
     */
    public CashRegister()
    {
        purchase = 0;
        payment = 0;
    }

    /**
     * Registra la vendita di un articolo.
     * @param amount il prezzo dell'articolo
     */
    public void recordPurchase(double amount)
    {
        purchase = purchase + amount;
    }

    /**
     * Riceve un pagamento dal cliente e lo registra.
     * @param dollars il numero di dollari
     * @param quarters il numero di quarti di dollaro
     * @param dimes il numero di decimi di dollaro
     * @param nickels il numero di nichelini
     * @param pennies il numero di centesimi di dollaro
     */
    public void receivePayment(int dollars, int quarters,
                               int dimes, int nickels, int pennies)
    {
        payment = dollars + quarters * QUARTER_VALUE + dimes * DIME_VALUE
                  + nickels * NICKEL_VALUE + pennies * PENNY_VALUE;
    }

    /**
     * Calcola il resto dovuto al cliente e azzerà il conto per il successivo.
     * @return il resto dovuto al cliente
     */
    public double giveChange()
    {
        double change = payment - purchase;
        purchase = 0;
        payment = 0;
        return change;
    }
}
```

**File CashRegisterTester.java**

```

1  /**
2   * Questa classe collauda la classe CashRegister.
3  */
4  public class CashRegisterTester
5  {
6      public static void main(String[] args)
7      {
8          CashRegister register = new CashRegister();
9
10         register.recordPurchase(0.75);
11         register.recordPurchase(1.50);
12         register.receivePayment(2, 0, 5, 0, 0);
13         System.out.print("Change: ");
14         System.out.println(register.giveChange());
15         System.out.println("Expected: 0.25");
16
17         register.recordPurchase(2.25);
18         register.recordPurchase(19.25);
19         register.receivePayment(23, 2, 0, 0, 0);
20         System.out.print("Change: ");
21         System.out.println(register.giveChange());
22         System.out.println("Expected: 2.0");
23     }
24 }
```

**Esecuzione del programma**

Change: 0.25  
 Expected: 0.25  
 Change: 2.0  
 Expected: 2.0

**Auto-valutazione**

1. Quali sono i tipi numerici più utilizzati in Java?
2. Immaginate di voler scrivere un programma che elabora dati relativi alla popolazione di diverse nazioni. In Java, quale tipo di dati usereste?
3. Quali di queste inizializzazioni sono corrette, e perché?
  - a. `int dollars = 100.0;`
  - b. `double balance = 100;`
4. Qual è la differenza tra i due enunciati seguenti?  
`final double CM_PER_INCH = 2.54;`  
`public static final double CM_PER_INCH = 2.54;`
5. Cosa c'è di sbagliato in questa sequenza di enunciati?  
`double diameter = . . .;`  
`double circumference = 3.14 * diameter;`

**Per far pratica**

A questo punto si consiglia di svolgere gli esercizi R.4.1, R.4.27 e E4.21, al termine del capitolo.

Argomenti avanzati 4.1

### Numeri grandi

Se intendete eseguire calcoli con numeri veramente grandi potete utilizzare oggetti che rappresentano grandi numeri: si tratta di oggetti delle classi `BigInteger` (numeri interi grandi) e `BigDecimal` (numeri frazionari grandi) appartenenti al pacchetto `java.math`. Diversamente dai tipi numerici quali `int` o `double`, i "numeri grandi" non hanno sostanzialmente alcun limite di dimensione e di precisione. Tuttavia, i calcoli effettuati con tali oggetti sono molto più lenti di quelli che elaborano tipi numerici e, forse ancora più importante, con questi oggetti non è possibile utilizzare i consueti operatori aritmetici (`+` - `*` / `%`): al loro posto bisogna invocare metodi, come `add`, `subtract` o `multiply`. Ecco un esempio di come si creano oggetti di tipo `BigInteger` e di come si invoca il metodo `multiply`:

```
BigInteger n = new BigInteger("1000000");
BigInteger r = n.multiply(n);
System.out.println(r); // Visualizza 10000000000000
```

Il tipo `BigDecimal` consente di effettuare calcoli in virgola mobile senza errori di arrotondamento, come in questo esempio:

```
BigDecimal d = new BigDecimal("4.35");
BigDecimal e = new BigDecimal("100");
BigDecimal f = d.multiply(e);
System.out.println(f); // Visualizza 435.00
```

#### **Suggerimenti per la programmazione 4.1**

### **Non usare numeri magici**

Un **numero magico** è un numero che compare nel codice senza spiegazioni. Per esempio, considerate il codice seguente che compare nel sorgente della libreria di Java:

$h = 31 * h + ch;$

Perché 31? Il numero di giorni nel mese di gennaio? Il numero di bit in un numero intero, meno uno? In realtà, questo codice calcola il “codice di hash” (*hash code*) di una stringa, un numero calcolato in base ai caratteri presenti nella stringa in modo che stringhe diverse generino con buona probabilità codici di hash diversi: il valore 31 ha l’effetto di “mescolare” in modo adeguato i valori dei caratteri.

Una soluzione migliore prevede di usare, invece, una costante con nome:

```
final int HASH_MULTIPLIER = 31;  
h = HASH_MULTIPLIER * h + ch;
```

Non si dovrebbero mai usare numeri magici nel codice: qualsiasi numero che non sia completamente auto-esplicativo dovrebbe essere dichiarato come costante, con un proprio nome. Anche la costante ritenuta più universale è destinata a cambiare, prima o

poi. Credete che in un anno vi siano 365 giorni? I vostri clienti su Marte saranno molto delusi dal vostro stupido pregiudizio. È meglio creare una costante come questa:

```
final int DAYS_PER_YEAR = 365;
```

## 4.2 Aritmetica

In questo paragrafo vedrete come eseguire calcoli aritmetici in Java.

### 4.2.1 Operatori aritmetici

Java consente l'esecuzione delle stesse quattro operazioni aritmetiche basilari che si trovano in una calcolatrice: addizione, sottrazione, moltiplicazione e divisione. Come avete già visto, per l'addizione e la sottrazione si usano i consueti operatori + e -, mentre per la moltiplicazione bisogna usare l'operatore \*, che, diversamente da quanto avviene in matematica, non può essere implicito: non si può scrivere  $a \cdot b$ , né  $a \cdot b$ , né  $a \times b$ . Analogamente, l'operazione di divisione si indica con /, non con l'operatore  $\div$  né usando il simbolo della frazione. Per esempio, la frazione

$$\frac{a+b}{2}$$

diventa

$$(a + b) / 2$$

La combinazione di variabili, numeri letterali, operatori e invocazioni di metodi si chiama **espressione** (*expression*). Ad esempio,  $(a + b) / 2$  è un'espressione.

Le parentesi si usano, esattamente come in algebra, per indicare in quale ordine vanno calcolate le espressioni parziali. Per esempio, nell'espressione  $(a + b) / 2$ , la somma  $a + b$  è calcolata per prima e soltanto in seguito il risultato dell'addizione viene diviso per 2. Al contrario, nell'espressione:

$$a + b / 2$$

soltanto  $b$  è diviso per 2, poi viene calcolata la somma tra  $a$  e  $b / 2$ . Proprio come nella normale notazione algebrica, moltiplicazione e divisione *hanno la precedenza* rispetto all'addizione e alla sottrazione. Per esempio, nell'espressione  $a + b / 2$ , la divisione / viene eseguita per prima, nonostante l'addizione + si trovi più a sinistra, cioè compaia più a sinistra nell'espressione. L'Appendice B riporta ulteriori dettagli sulla precedenza tra operatori.

Se un'espressione aritmetica contiene alcuni valori interi e altri in virgola mobile, il risultato è un valore in virgola mobile. Ad esempio, l'espressione  $7 + 4.0$  ha come risultato il valore in virgola mobile  $11.0$ .

Se un'espressione aritmetica contiene  
alcuni valori interi e altri in virgola  
mobile, il risultato è un valore  
in virgola mobile.

## 4.2.2 Incremento e decremento

Gli operatori `++` e `--` incrementano e, rispettivamente, decrementano di un'unità il valore di una variabile.

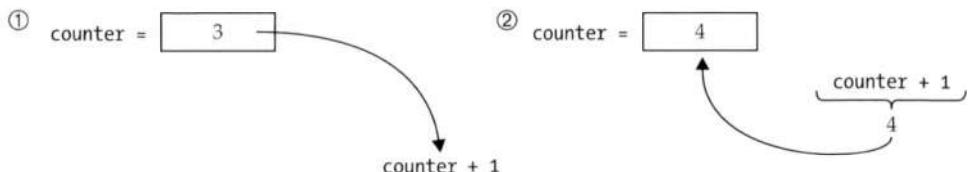
L'operazione di incremento (cioè di “aggiunta di un'unità a un valore”) è talmente comune nella scrittura dei programmi che ne esiste una speciale forma abbreviata, ovvero:

```
counter++; // aggiunge 1 alla variabile counter
```

Analogamente, esiste anche un operatore per il decremento, che è `--`:

```
counter--; // sottrae 1 alla variabile counter
```

**Figura 1**  
Incrementare una variabile



## 4.2.3 Divisione intera e resto

Se entrambi gli argomenti dell'operatore `/` sono di tipo intero, il risultato è un numero intero e il resto viene ignorato.

La divisione funziona come ci si aspetta che funzioni, a patto che almeno uno dei numeri coinvolti sia in virgola mobile. Quindi, tutte le divisioni seguenti hanno come risultato 1.75:

```
7.0 / 4.0  
7 / 4.0  
7.0 / 4
```

Tuttavia, se *entrambi* i numeri sono di tipo intero, il risultato della divisione è sempre un numero intero, e il resto viene ignorato (si parla di **divisione intera**). Quindi

```
7 / 4
```

vale 1, perché 7 diviso per 4 dà come risultato 1 (con resto 3, che viene ignorato). Trascurare questo resto può essere utile in alcune situazioni, ma altre volte può causare insidiosi errori di programmazione (come discusso nella sezione Errori comuni 4.1).

Se interessa solamente il resto della divisione, si usa l'operatore `%`. Il risultato calcolato per l'espressione

```
7 % 4
```

L'operatore `%` calcola il resto di una divisione intera.

è 3, cioè il resto della divisione intera di 7 per 4. Il simbolo `%` non ha analogie in algebra ed è stato scelto perché è simile alla barra `/`, dal momento che l'operazione di calcolo del resto è connessa con la divisione. A volte si parla di “operatore modulo” e, comunque, non ha alcuna relazione con l'operazione di calcolo della percentuale che corrisponde a tale simbolo in molte calcolatrici tascabili.

Vediamo un tipico utilizzo della divisione intera / e di calcolo del resto mediante l'operatore %. Supponiamo di conoscere il valore di alcune monete contenute in un salvadanaio, espresso in centesimi:

```
int pennies = 1729;
```

Vogliamo determinare il valore corrispondente espresso in dollari e centesimi. Il numero di dollari si può calcolare con una divisione intera per 100:

```
int dollars = pennies / 100; // assegna 17 a dollars
```

La divisione intera ignora il resto. Per conoscerlo, usiamo l'operatore %:

```
int cents = pennies % 100; // assegna 29 a cents
```

La Tabella 3 riporta ulteriori esempi.

**Tabella 3**  
Divisione intera e resto

Espressione (dove n = 1729)	Valore	Commento
n % 10	9	n % 10 è sempre l'ultima cifra di n.
n / 10	172	Questa espressione è sempre n senza la sua ultima cifra.
n % 100	29	Le ultime due cifre di n.
n / 10.0	172.9	Dato che 10.0 è un numero in virgola mobile, la parte frazionaria del risultato non viene ignorata.
-n % 10	-9	Dato che il primo operando è negativo, anche il resto è negativo.
n % 2	1	n % 2 è 0 se n è pari, è 1 o -1 se n è dispari.

#### 4.2.4 Potenze e radici

La libreria di Java contiene molte funzioni matematiche; ad esempio, Math.sqrt e Math.pow calcolano, rispettivamente, radici quadrate e potenze.

In Java non esistono simboli per esprimere l'elevamento a potenza o l'estrazione di radice: per calcolare queste funzioni è necessario invocare metodi. Per estrarre la radice quadrata di un numero si usa il metodo Math.sqrt: per estrarre la radice quadrata di x si scrive Math.sqrt(x). Per calcolare  $x^n$  si invoca il metodo Math.pow(x, n).

In algebra si usano frazioni, esponenti e radici per comporre espressioni in una forma bidimensionale compatta, mentre in Java bisogna scrivere tutte le espressioni secondo una disposizione lineare. Per esempio, l'espressione:

$$b \cdot \left(1 + \frac{r}{100}\right)^n$$

in Java, diventa:

```
b * Math.pow(1 + r / 100, n)
```

La Figura 2 mostra come viene analizzata un'espressione simile e la Tabella 4 riporta ulteriori metodi matematici.

**Figura 2**  $b * \text{Math.pow}(1 + r / 100, n)$   
Analisi di un'espressione

$$\begin{aligned}
 & b \times \underbrace{\left(1 + \frac{r}{100}\right)^n}_{\text{calcolo della potenza}} \\
 & \quad \underbrace{\left(1 + \frac{r}{100}\right)}_{\text{calcolo del fattore}}^n \\
 & \quad \underbrace{\frac{r}{100}}_{\text{calcolo del rapporto}}
 \end{aligned}$$

**Tabella 4**

Metodi matematici

Math.sqrt(x)	radice quadrata di $x$ ( $\geq 0$ )
Math.pow(x, y)	$x^y$ ( $x > 0$ , oppure $x = 0$ e $y > 0$ , oppure $x < 0$ e $y$ è intero)
Math.sin(x)	seno di $x$ ( $x$ in radienti)
Math.cos(x)	coseno di $x$
Math.tan(x)	tangente di $x$
Math.toRadians(x)	converte $x$ da gradi a radienti (cioè restituisce $x \cdot \pi/180$ )
Math.toDegrees(x)	converte $x$ da radienti a gradi (cioè restituisce $x \cdot 180/\pi$ )
Math.exp(x)	$e^x$
Math.log(x)	logaritmo naturale ( $\ln(x)$ , $x > 0$ )
Math.log10(x)	logaritmo decimale ( $\log_{10}(x)$ , $x > 0$ )
Math.round(x)	il numero intero (di tipo <code>long</code> ) più prossimo a $x$
Math.ceil(x)	il più piccolo numero intero (di tipo <code>double</code> ) che sia $\geq x$
Math.floor(x)	il più grande numero intero (di tipo <code>double</code> ) che sia $\leq x$
Math.abs(x)	valore assoluto, $ x $
Math.max(x, y)	il numero maggiore tra $x$ e $y$
Math.min(x, y)	il numero minore tra $x$ e $y$

### 4.2.5 Conversione e arrotondamento

A volte si vuole convertire un valore di tipo `double` in un valore di tipo `int`, ma l'assegnazione di un valore in virgola mobile a una variabile intera è un errore:

```
double balance = total + tax;
int dollars = balance; // ERRORE: non si può assegnare un double a un int
```

Il compilatore non consente questa assegnazione perché è potenzialmente pericolosa:

- La parte frazionaria si perde.

- Il valore assoluto del numero può essere troppo grande (il massimo valore assoluto per un valore di tipo `int` è circa 2 miliardi, ma un valore di tipo `double` può essere molto più grande).

Per convertire un valore in un tipo diverso si usa il cast (`nomeTipo`).

Per risolvere questo problema si usa l'operatore **cast** ("forzatura"), che, in questo caso, si esprime in questo modo: `(int)`. L'operatore cast va scritto prima dell'espressione che deve convertire:

```
double balance = total + tax;
int dollars = (int) balance;
```

Il cast `(int)` trasforma in un numero intero il valore in virgola mobile contenuto nella variabile `balance`, ignorandone la parte frazionaria. Ad esempio, se `balance` vale 13.75, a `dollars` viene assegnato il valore 13.

Se si vuole applicare l'operatore cast al risultato prodotto dalla valutazione di un'espressione aritmetica, bisogna inserire l'espressione tra parentesi:

```
int dollars = (int) (total + tax);
```

Non sempre si vuole eliminare la parte frazionaria ignorandone il valore: a volte si vuole arrotondare un valore in virgola mobile al numero intero più vicino, risultato che si può ottenere usando il metodo `Math.round`, che restituisce un valore intero di tipo `long`, perché i numeri in virgola mobile più grandi non trovano posto in una variabile di tipo `int`.

```
long rounded = Math.round(balance);
```

Se `balance` vale 13.75, a `rounded` viene assegnato il valore 14.

Se, però, sapete che il risultato può essere memorizzato in una variabile di tipo `int` e non serve una variabile di tipo `long`, potete usare un cast:

```
int rounded = (int) Math.round(balance);
```

## Sintassi di Java

## 4.2 Cast

### Sintassi

`(nomeTipo) espressione`

### Esempio

Questo è il tipo dell'espressione dopo la conversione.

Queste parentesi tonde fanno parte dell'operatore cast.

`(int) (balance * 100)`

Qui usate le parentesi se il cast va applicato a un'espressione che contiene operatori aritmetici.

**Tabella 5**  
Espressioni aritmetiche

Espressione matematica	Espressione in Java	Commenti
$\frac{x+y}{2}$	$(x + y) / 2$	Le parentesi sono necessarie: $x + y / 2$ calcolerebbe $x + \frac{y}{2}$ .
$\frac{xy}{2}$	$x * y / 2$	Le parentesi non sono necessarie: operatori aventi la medesima precedenza vengono valutati da sinistra a destra.
$\left(1 + \frac{r}{100}\right)^n$	<code>Math.pow(1 + r / 100, n)</code>	Per calcolare $x^n$ si usa <code>Math.pow(x, n)</code> .
$\sqrt{a^2 + b^2}$	<code>Math.sqrt(a * a + b * b)</code>	$a * a$ è più semplice di <code>Math.pow(a, 2)</code> .
$\frac{i+j+k}{3}$	$(i + j + k) / 3.0$	Se $i, j$ e $k$ sono valori interi, l'uso di 3.0 come denominatore produce una divisione in virgola mobile.
$\pi$	<code>Math.PI</code>	<code>Math.PI</code> è una costante dichiarata nella classe <code>Math</code> .



## Auto-valutazione

6. In conto bancario vengono accreditati gli interessi una volta all'anno. In Java, come si calcolano gli interessi maturati durante il primo anno? Ipotizzate che le variabili `percent` e `balance`, di tipo `double`, siano già state dichiarate.
7. Come si calcola, in Java, la lunghezza del lato di un quadrato la cui area è memorizzata nella variabile `area`?
8. Scrivere, in Java, l'espressione che calcola il volume di una sfera il cui raggio sia memorizzato nella variabile `radius` di tipo `double`.
9. Qual è il valore delle espressioni `1729 / 100` e `1729 % 100`?
10. Se `n` è un numero intero positivo, cosa rappresenta `(n / 10) % 10`?

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi R4.4, R4.8, E4.4 e E4.24, al termine del capitolo.



## Errori comuni 4.1

### Divisione intera inconsapevole

È una sfortuna che Java usi lo stesso simbolo, la barra `/`, per entrambi i tipi di divisione, quella fra numeri interi e quella fra numeri in virgola mobile, perché si tratta di operazioni davvero molto diverse. In conseguenza di ciò, usare la divisione fra interi in modo inconsapevole è un errore comune. Esaminiamo questa porzione di programma che calcola la media fra tre numeri interi:

```
int score1 = 10;
int score2 = 4;
int score3 = 9;
double average = (score1 + score2 + score3) / 3; // Errore
System.out.print("Average score: " + average); // Visualizza 7.0, non 7.6666666666666667
```

Che cosa può esservi di sbagliato? Ovviamente, la media di `score1`, `score2` e `score3` è:

$$\frac{\text{score1} + \text{score2} + \text{score3}}{3}$$

Nell'esempio, però, la barra / non indica una divisione nel senso matematico, ma indica una divisione fra numeri interi, perché sia 3 sia la somma `score1 + score2 + score3` sono valori interi. Dato che la somma dei punteggi è 23, la media viene calcolata come 7, vale a dire il risultato della divisione intera di 23 per 3. In conclusione, nella variabile in virgola mobile `average` viene memorizzato il numero intero 7. La soluzione consiste nell'utilizzare come numeratore o come denominatore un numero in virgola mobile:

```
double total = score1 + score2 + score3;
double average = total / 3;
```

oppure:

```
double average = (score1 + score2 + score3) / 3.0;
```



## Errori comuni 4.2

### Parentesi non accoppiate

Esaminiamo l'espressione:

$$((a + b) * t / 2 * (1 - t)$$

Cosa c'è che non va? Contate le parentesi: ci sono tre parentesi aperte, ma soltanto due parentesi chiuse, per cui *le parentesi non sono accoppiate correttamente*. Questo tipo di errore di digitazione è molto comune nelle espressioni complicate. Ora, considerate questa espressione:

$$(a + b) * t) / (2 * (1 - t)$$

L'espressione presenta tre parentesi aperte e tre parentesi chiuse, ma di nuovo non è corretta. Nella parte centrale dell'espressione

$$(a + b) * t) / (2 * (1 - t)$$

↑

c'è soltanto una parentesi aperta ma due parentesi chiuse, da cui l'errore. In qualsiasi punto interno a un'espressione, il numero di parentesi aperte dall'inizio dell'espressione deve essere maggiore o uguale a quello delle parentesi chiuse; inoltre, al termine dell'espressione i due conteggi devono equivalersi.

Ecco un semplice trucco per contare le parentesi senza usare carta e matita. Mentalmente è difficile seguire due numerazioni contemporaneamente, quindi, quando controllate l'espressione, usate un solo conteggio: iniziate da 1 quando trovate la prima parentesi aperta e sommate 1 ogni volta che vedete un'altra parentesi aperta, mentre sottrate 1 se trovate una parentesi chiusa. Scandite i numeri ad alta voce, mentre scorrete l'espressione da sinistra a destra: se in qualche punto il conteggio scende sotto lo zero

oppure se alla fine dell'espressione non è uguale a zero, le parentesi non sono bilanciate. Per esempio, mentre scorrete l'espressione precedente, dovreste mormorare:

$$\begin{array}{ccccccc} & (a + b) * t) / (2 * (1 - t) \\ & \quad \quad \quad 1 \quad \quad \quad 0 \quad \quad \quad -1 \end{array}$$

e, quindi, trovare l'errore.



## Suggerimenti per la programmazione 4.2

### Spaziature nelle espressioni

È più facile leggere:

```
x1 = (-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a);
```

piuttosto che:

```
x1=(-b+Math.sqrt(b*b-4*a*c))/(2*a);
```

È sufficiente inserire spazi ai lati degli operatori + - \* / % =, senza, però, inserire uno spazio dopo un segno meno “unario”, ovvero un segno meno usato per convertire un singolo valore nel suo corrispondente valore negativo, come nell'espressione -b. In questa forma, sarà più facile distinguerlo dal segno meno “binario”, quello che si trova nell'espressione a - b.

Di solito, poi, non si mette uno spazio dopo il nome di un metodo, cioè si scrive `Math.sqrt(x)` e non `Math.sqrt (x)`.



## Note per Java 8 4.1

### Evitare resti negativi

Se il primo operando è negativo, l'operatore % fornisce un risultato negativo, una caratteristica spesso scomoda. Pensate, ad esempio, a un robot che memorizzi la propria direzione, `direction`, mediante un angolo compreso tra 0 e 359 gradi. Se compie una rotazione di un certo numero di gradi (`turn`), non si può calcolare la nuova direzione usando semplicemente l'espressione `(direction + turn) % 360`, perché si potrebbe ottenere un risultato negativo (come si può vedere nell'Esercizio R4.7). Nella versione 8 di Java si può invocare il metodo:

```
Math.floorMod(direction + turn, 360)
```

Questo calcola il resto in modo corretto: il risultato di `Math.floorMod(m, n)` è sempre positivo, se `n` è positivo.



## Argomenti avanzati 4.2

### Combinare assegnazioni e operatori aritmetici

In Java, possiamo combinare operatori aritmetici e assegnazioni. Per esempio, l'enunciato:

```
balance += amount;
```



## Computer e società 4.1

### Errore nel calcolo in virgola mobile nel Pentium

Nel 1994, Intel Corporation mise in vendita quello che poi divenne il suo processore più potente, il primo della serie Pentium. Diversamente dai processori Intel delle generazioni precedenti, il Pentium aveva un'unità molto veloce dedicata al calcolo in virgola mobile. L'obiettivo di Intel era quello di competere aggressivamente con i costruttori dei processori di livello avanzato per workstation professionali e il Pentium ebbe immediatamente un successo enorme.

Nell'estate di quell'anno, il Dott. Thomas Nicely del Lynchburg College, in Virginia, eseguì una lunga serie di calcoli per analizzare le proprietà delle somme dei reciproci di determinate sequenze di numeri primi: i risultati non sempre corrispondevano a quelli previsti dalla sua teoria, sia pure considerando gli inevitabili errori di arrotondamento. Successivamente, il Dott. Nicely notò che lo stesso programma produceva il risultato corretto quando veniva eseguito sul più lento processore 486, che aveva preceduto il Pentium nella produzione Intel: questo non doveva succedere. Il metodo di arrotondamento migliore, nelle operazioni in

virgola mobile, era stato reso standard da IEEE (Institute of Electrical and Electronics Engineers) e Intel affermava di aderire agli standard IEEE in entrambi i processori, 486 e Pentium. In seguito a ulteriori controlli, il Dott. Nicely scoprì che, in verità, esisteva un insieme di numeri molto limitato per i quali il prodotto di due numeri veniva calcolato in modo diverso dai due processori. Per esempio, l'espressione seguente

$$4195835 - ((4195835/3145727) \times 3145727)$$

deve, ovviamente, avere il valore zero e un processore 486 lo confermava. Tuttavia, eseguendo il calcolo con un processore Pentium, il risultato era 256.

Come si seppe poi, Intel aveva scoperto l'errore indipendentemente, nei suoi collaudi, e aveva iniziato a produrre chip che ne erano esenti. Il malfunzionamento era causato da un errore nella tabella che serviva per accelerare l'algoritmo usato dal processore per moltiplicare i numeri in virgola mobile: Intel decise che il problema era piuttosto raro e dichiarò che un utilizzatore tipico, in condizioni operative normali, avreb-

be riscontrato l'errore solo una volta ogni 27000 anni. Sfortunatamente per Intel, il Dott. Nicely non era un utente normale.

A quel punto Intel si trovò a fronteggiare un grosso problema: calcolò che sostituire tutti i processori Pentium già venduti sarebbe costato una grossa somma di denaro; inoltre, Intel aveva già più ordinazioni di chip di quanti poteva produrne e sarebbe stato particolarmente irritante dover utilizzare le scorte, già limitate, per la sostituzione gratuita, invece di destinarle alla vendita. Inizialmente la direzione di Intel offrì la sostituzione dei processori soltanto ai clienti in grado di dimostrare che la loro attività richiedeva precisione assoluta nei calcoli matematici. Naturalmente, la cosa non andò giù alle centinaia di migliaia di clienti che avevano pagato anche più di 700 dollari per un Pentium e non volevano vivere con la fastidiosa sensazione che forse, un giorno, il loro programma per il calcolo delle imposte avrebbe prodotto una dichiarazione dei redditi sbagliata.

Alla fine Intel fu costretta a cedere alla richiesta generale e a sostituire tutti i chip difettosi, al costo di circa 475 milioni di dollari.

è un'abbreviazione per:

```
balance = balance + amount;
```

Analogamente,

```
total *= 2;
```

è un modo alternativo per scrivere:

```
total = total * 2;
```

Molti programmatori ritengono queste forme una comoda scorciatoia. Se vi piacciono, non esitate a utilizzarle nel vostro codice, ma in questo libro non le useremo, per semplicità.

## Argomenti avanzati 4.3

### Metodi di esemplare e metodi statici

Nel paragrafo precedente abbiamo visto la classe `Math`, che contiene molti metodi utili per eseguire calcoli matematici. Si tratta di metodi che non agiscono su un oggetto, infatti non vengono invocati in questo modo:

```
double root = 2.sqrt(); // ERRORE
```

I numeri, in Java, non sono oggetti, per cui non vedrete mai un metodo invocato usando un numero. Per fare in modo che un metodo elabori un numero, lo si passa come argomento (esplicito):

```
double root = Math.sqrt(2);
```

Questi metodi sono chiamati **metodi statici** (*static method*), dove il termine “statico” è stato ereditato dai linguaggi di programmazione C e C++, ma non ha niente a che vedere con il consueto significato della parola.

I metodi statici non agiscono su oggetti, ma sono comunque dichiarati all’interno di una classe. Quando si invoca uno di tali metodi, si indica la classe a cui appartiene, in questo modo:

Nome della classe      Nome del metodo statico  
                          ↓  
                          Math.sqrt(2)

Al contrario, un metodo che viene invocato usando un oggetto come parametro implicito si chiama **metodo di esemplare** o di istanza (*instance method*). Come regola pratica, quando elaborate numeri usate metodi statici, ma nel Capitolo 8 approfondirete le differenze tra metodi statici e metodi di esemplare.

## 4.3 Dati in ingresso e in uscita

Nei prossimi paragrafi vedrete come acquisire dati in ingresso, forniti dall’utente, e come controllare il formato delle informazioni visualizzate dal programma come prodotto della propria elaborazione.

### 4.3.1 Acquisire dati

I programmi possono essere resi molto più versatili facendo in modo che elaborino dati forniti dall’utente piuttosto che valori prefissati. Considerate, ad esempio, un programma che elabori prezzi e quantità di contenitori di bibite, due valori che sono soggetti a fluttuazioni: dovrebbe essere l’utente del programma a fornirli come dati in ingresso.

Quando un programma chiede dati all'utente, per prima cosa dovrebbe comunicare ciò che si aspetta di ricevere, visualizzando un messaggio che solitamente viene chiamato **prompt** ("sollecito" o "suggerimento"):

```
System.out.print("Please enter the number of bottles: "); // prompt
```

Per visualizzare il prompt è meglio usare il metodo `print`, non il metodo `println`: in questo modo i dati forniti in ingresso verranno scritti subito dopo il carattere "due punti", non sulla riga seguente. Inoltre, è anche bene ricordare di inserire uno spazio dopo i "due punti".

Dal momento che i dati in uscita vengono inviati a `System.out`, potreste pensare di usare `System.in` per ricevere dati in ingresso: sfortunatamente la cosa non è così semplice. Quando fu progettato il linguaggio Java, non venne riposta molta attenzione alle interazioni con la tastiera, perché si ipotizzò che tutti i programmati avrebbero scritto applicazioni dotate di interfaccia grafica per l'interazione con l'utente, con menu e campi di testo. All'oggetto `System.in` venne associato un insieme di funzionalità davvero minimali e, per essere utilizzabile, in pratica deve essere combinato con altre classi.

Per acquisire dati tramite la tastiera si usa la classe `Scanner`. Per costruire un oggetto di tipo `Scanner` si usa questo enunciato:

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

Dopo aver creato uno scanner ("scansionatore"), potete usare il suo metodo `nextInt` per leggere un numero intero:

```
System.out.print("Please enter the number of bottles: ");
int bottles = in.nextInt();
```

Quando viene invocato il metodo `nextInt`, il programma si arresta in attesa che l'utente scriva un numero e prema sulla tastiera il tasto "Enter" (o "Invio"): poi, il numero acquisito viene memorizzato nella variabile `bottles` e il programma procede.

Per leggere un numero in virgola mobile, si usa invece il metodo `nextDouble`:

```
System.out.print("Enter price: ");
double price = in.nextDouble();
```

La classe `Scanner` appartiene al pacchetto `java.util`. Quando usate la classe `Scanner`, importatela inserendo all'inizio del file sorgente questa dichiarazione:

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

### 4.3.2 Controllare il formato di visualizzazione

Quando si visualizzano i risultati di un'elaborazione, spesso se ne vuole controllare il formato e l'impaginazione in modo preciso. Ad esempio, quando si visualizza una quantità di denaro in dollari e centesimi, solitamente si vogliono visualizzare due sole cifre decimali nella parte frazionaria, arrotondando il valore in modo opportuno. Si vuole, cioè visualizzare questo:

```
Price per liter: 1.22
```

invece di:

## Sintassi di Java

### 4.3 Acquisizione di dati

Inserendo questa riga si può usare la classe Scanner.

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

Crea un oggetto di tipo Scanner per leggere dalla tastiera.

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

Visualizza un prompt nella finestra di console.

```
:  
System.out.print("Please enter the number of bottles: ");
```

Definisce una variabile per memorizzare il valore acquisito.

```
int bottles = in.nextInt();
```

Qui non usare println.

Il programma attende che l'utente digiti il dato, poi lo memorizza nella variabile.

Price per liter: 1.215962441314554

Per specificare come vanno visualizzati i valori si usa il metodo printf.

L'enunciato seguente visualizza il prezzo (price) con due cifre dopo il punto decimale:

```
System.out.printf("%.2f", price);
```

È anche possibile specificare un'ampiezza per il campo (*field width*) in cui viene visualizzato il dato:

```
System.out.printf("%10.2f", price);
```

In questo caso il prezzo viene visualizzato usando dieci caratteri: sei spazi seguiti dai quattro caratteri 1.22.

La stringa "%10.2f" è una *specifica di formato (format specifier)* che descrive come va visualizzato un valore. La lettera f finale segnala che il valore da visualizzare è un numero in virgola mobile: si usa d per i numeri interi e s per le stringhe (la Tabella 6 riporta altri esempi).

Il primo parametro del metodo printf è, più in generale, una *stringa di controllo del formato (format string)*, che può contenere caratteri da stampare così come sono ("caratteri letterali") e caratteri che fungono da *specifiche di formato*. Qualsiasi carattere che non faccia parte di una specifica di formato viene visualizzato così com'è. Ad esempio:

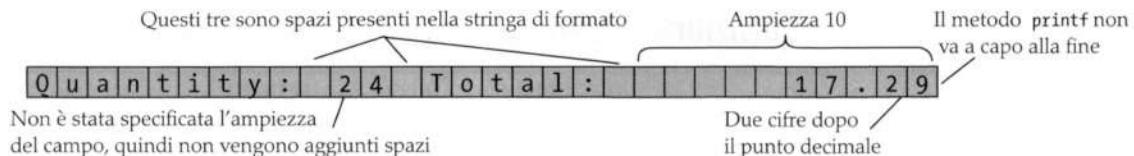
```
System.out.printf("Price per liter:%10.2f", price);
```

visualizza

Price per liter: 1.22

Con un'unica invocazione di printf si possono anche visualizzare più valori. Ecco un esempio tipico:

```
System.out.printf("Quantity: %d Total: %10.2f", quantity, total);
```



**Tabella 6**  
Esempi di specifiche di formato

Stringa di formato	Esempio di visualizzazione	Commento
"%d"	24	Per un numero intero si usa d.
"%5d"	24	Vengono aggiunti spazi a sinistra in modo che il campo visualizzato abbia lunghezza 5.
"Quantity:%d"	Quantity: 24	I caratteri presenti nella stringa ma al di fuori di una specifica di formato vengono visualizzati.
"%f"	1.21997	Per un numero in virgola mobile si usa f.
"%.2f"	1.22	Visualizza due cifre dopo il punto separatore decimale.
"%7.2f"	1.22	Vengono aggiunti spazi a sinistra in modo che il campo visualizzato abbia lunghezza 7.
"%s"	Hello	Per una stringa si usa s.
"%d %.2f"	24 1.22	Si possono visualizzare più valori con un'unica stringa di controllo del formato.

Il metodo `printf`, proprio come il metodo `print`, non va a capo al termine della visualizzazione prodotta: se volete che i caratteri successivi compaiano su una nuova riga, dovete invocare `System.out.println()`. In alternativa, nel Paragrafo 4.5.4 vedrete come aggiungere un “carattere che va a capo” (*newline*) all’interno della stringa che controlla il formato.

Il prossimo programma chiede all’utente di fornire il prezzo di una confezione di sei bottiglie di bibita e quello di una singola bottiglia da due litri, poi visualizza il prezzo per litro nei due casi, mettendo in pratica quanto avete appreso in relazione all’acquisizione dei dati e alla loro visualizzazione.

### File Volume.java

```

1 import java.util.Scanner
2
3 /**
4  * Questo programma visualizza il prezzo al litro nel caso di una
5  * confezione da sei bottiglie e di una singola bottiglia da due litri.
6 */
7 public class Volume
8 {
9     public static void main(String[] args)
10    {
11        // legge il prezzo di una confezione da sei bottiglie
12
13        Scanner in = new Scanner(System.in);
14
15        System.out.print("Please enter the price for a six-pack: ");
16        double packPrice = in.nextDouble();

```

```

17
18 // legge il prezzo di una bottiglia da due litri
19
20 System.out.print("Please enter the price for a two-liter bottle: ");
21 double bottlePrice = in.nextDouble();
22
23 final double CANS_PER_PACK = 6;
24 final double CAN_VOLUME = 0.335; // 12 oz. = 0.335 l
25 final double BOTTLE_VOLUME = 2;
26
27 // calcola e visualizza i prezzi per litro
28
29 double packPricePerLiter = packPrice / (CANS_PER_PACK * CAN_VOLUME);
30 double bottlePricePerLiter = bottlePrice / BOTTLE_VOLUME;
31
32 System.out.printf("Pack price per liter: %8.2f", packPricePerLiter);
33 System.out.println();
34
35 System.out.printf("Bottle price per liter: %8.2f", bottlePricePerLiter);
36 System.out.println();
37 }
38 }
```

### Esecuzione del programma

```

Please enter the price for a six-pack: 2.95
Please enter the price for a two-liter bottle: 2.85
Pack price per liter:      1.38
Bottle price per liter:    1.43
```



### Auto-valutazione

11. Scrivete enunciati che chiedano all'utente l'età e la acquisiscano usando la variabile `in` di tipo `Scanner`.
12. Che errore c'è in questa sequenza di enunciati?  
`System.out.print("Please enter the unit price: ");`  
`double unitPrice = in.nextDouble();`  
`int quantity = in.nextInt();`
13. Che problema c'è in questa sequenza di enunciati?  
`System.out.print("Please enter the unit price: ");`  
`double unitPrice = in.nextInt();`
14. Che problema c'è in questa sequenza di enunciati?  
`System.out.print("Please enter the number of cans");`  
`int cans = in.nextInt();`
15. Cosa visualizza questa sequenza di enunciati?  
`int volume = 10;`  
`System.out.printf("The volume is %5d", volume);`
16. Usando il metodo `printf`, visualizzate i valori delle variabili intere `bottles` e `cans`, con questa impaginazione:  
 Bottles: 8  
 Cans: 24

I numeri devono essere incolonnati a destra (ipotizzando che entrambi abbiano al massimo 8 cifre).

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R4.13, E4.6 e E4.7, al termine del capitolo.



## Consigli pratici 4.1

### Effettuare calcoli

Molti problemi di programmazione richiedono l'utilizzo di calcoli aritmetici: questa sezione mostra come trasformare l'enunciazione di un problema in una soluzione descritta mediante pseudocodice e, infine, in un programma Java.

**Problema.** Dovete scrivere un programma che simuli un distributore automatico. Un cliente seleziona un articolo da acquistare e inserisce una banconota nel distributore, che consegna l'articolo acquistato e il resto. Ipotizziamo che tutti i prezzi siano multipli di 25 centesimi e che la macchina dia il resto con monete da un dollaro e da un quarto di dollaro (cioè 25 centesimi). Il programma deve, quindi, calcolare quante monete di ciascun tipo debbano essere consegnate come resto.

**Fase 1.** Analizzare il problema. Quali sono i dati da acquisire? E quali i dati da produrre?

In questo problema ci sono due dati in ingresso:

- L'importo della banconota inserita dal cliente
- Il prezzo dell'articolo acquistato

e ci sono due risultati da produrre:

- Il numero di monete da un dollaro che la macchina deve fornire
- Il numero di monete da un quarto di dollaro che la macchina deve fornire

**Fase 2.** Risolvere alcuni esempi a mano

Questo è un passo molto importante: se non riuscite a calcolare un paio di soluzioni a mano, è assai poco probabile che possiate scrivere un programma che automatizzi il calcolo.

Ipotizziamo che un cliente acquisti un articolo che costa \$2.25, inserendo una banconota da \$5. Il resto è, quindi, \$2.75, cioè 2 monete da un dollaro e 3 monete da un quarto di dollaro.

È un calcolo facile, ma come può un programma Java giungere alle stesse conclusioni? La soluzione consiste nel fare i calcoli usando i centesimi, non i dollari. Il resto da dare al cliente è pari a 275 centesimi. Dividendo tale valore per 100 si ottiene 2, il numero di monete da un dollaro. Dividendo, poi, il resto (cioè 75 centesimi) per 25, si ottiene 3, il numero di monete da un quarto di dollaro.

**Fase 3.** Scrivere pseudocodice che calcoli le risposte

Nella fase precedente abbiamo ragionato su un esempio specifico, ora dobbiamo trovare un metodo di soluzione generale, che funzioni sempre.

Data una banconota e un prezzo, come si può calcolare il numero di monete da fornire come resto? Per prima cosa calcoliamo il resto in centesimi:

$$\text{resto in centesimi} = 100 \times \text{banconota} - \text{prezzo in centesimi}$$

Per trovare il numero di monete da un dollaro che compongono il resto, dividiamo per 100 e ignoriamo il resto:

$$\text{monete da un dollaro} = \text{resto in centesimi} / 100 \text{ (ignorando il resto)}$$

La parte rimanente del resto dovuto si può calcolare in due modi. Usando l'operatore modulo, si può semplicemente scrivere:

$$\text{resto in centesimi} = \text{resto in centesimi} \% 100$$

Altrimenti si può sottrarre dal resto in centesimi il valore in centesimi delle monete da un dollaro:

$$\text{resto in centesimi} = \text{resto in centesimi} - 100 \times \text{monete da un dollaro}$$

Infine, si divide per 25, ottenendo il numero di monete da un quarto di dollaro:

$$\text{monete da un quarto di dollaro} = \text{resto in centesimi} / 25$$

**Fase 4.** Dichiare le variabili e le costanti che servono, specificandone i tipi

Fin qui abbiamo individuato cinque variabili:

- `billValue` (valore della banconota)
- `itemPrice` (prezzo dell'articolo acquistato)
- `changeDue` (resto dovuto)
- `dollarCoins` (numero di monete da un dollaro)
- `quarters` (numero di monete da un quarto di dollaro)

È opportuno introdurre costanti che spieghino il significato dei numeri 100 e 25, ad esempio `PENNIES_PER_DOLLAR` e `PENNIES_PER_QUARTER`. Questo agevolerebbe l'adattamento del programma a un diverso mercato, per cui è meglio farlo.

È molto importante che `changeDue` e `PENNIES_PER_DOLLAR` siano di tipo `int`, perché il calcolo di `dollarCoins` usa la divisione intera, e lo stesso dicasi per le altre variabili.

**Fase 5.** Trasformare lo pseudocodice in enunciati Java

Se avete fatto un buon lavoro nella scrittura dello pseudocodice questa fase dovrebbe essere semplice: ovviamente dovete sapere come si esprimono in Java le operazioni matematiche (come l'elevamento a potenza o la divisione intera).

```
changeDue = PENNIES_PER_DOLLAR * billValue - itemPrice;  
dollarCoins = changeDue / PENNIES_PER_DOLLAR;
```

```
changeDue = changeDue % PENNIES_PER_DOLLAR;
quarters = changeDue / PENNIES_PER_QUARTER;
```

**Fase 6.** Scrivere codice per acquisire i dati e visualizzare i risultati

Prima di iniziare l'elaborazione chiediamo all'utente il valore della banconota e il prezzo dell'articolo acquistato:

```
System.out.print("Enter bill value (1 = $1 bill, 5 = $5 bill, etc.): ");
billValue = in.nextInt();
System.out.print("Enter item price in pennies: ");
itemPrice = in.nextInt();
```

Finiti i calcoli, visualizziamo il risultato, usando il metodo `printf` per garantire un corretto allineamento.

```
System.out.printf("Dollar coins: %6d", dollarCoins);
System.out.println();
System.out.printf("Quarters:      %6d", quarters);
System.out.println();
```

**Fase 7.** Scrivere il metodo `main` nella classe

I calcoli da eseguire devono essere inseriti in una classe, per la quale bisogna individuare un nome adeguato, che descriva i calcoli che eseguirà. In questo caso sceglieremo il nome `VendingMachine`.

Nella classe occorre definire il metodo `main`, nel quale vanno dichiarate le variabili e le costanti individuate nella Fase 4, con gli enunciati di calcolo (Fase 5) e di gestione dei dati in ingresso e in uscita (Fase 6). Chiaramente la prima parte del metodo si occuperà dell'acquisizione dei dati, poi verranno eseguiti i calcoli e, infine, verranno visualizzati i risultati. Le costanti vanno dichiarate all'inizio del metodo, mentre le variabili si dichiarano nel punto in cui si rendono necessarie per la prima volta.

- Ecco il programma completo.

```
import java.util.Scanner;

/**
 * Questo programma simula un distributore automatico che dà il resto.
 */
public class VendingMachine
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);

        final int PENNIES_PER_DOLLAR = 100;
        final int PENNIES_PER_QUARTER = 25;

        System.out.print("Enter bill value (1 = $1 bill, 5 = $5 bill, etc.): ");
        billValue = in.nextInt();
        System.out.print("Enter item price in pennies: ");
        itemPrice = in.nextInt();

        // calcola il resto dovuto
```

```

changeDue = PENNIES_PER_DOLLAR * billValue - itemPrice;
dollarCoins = changeDue / PENNIES_PER_DOLLAR;
changeDue = changeDue % PENNIES_PER_DOLLAR;
quarters = changeDue / PENNIES_PER_QUARTER;

// visualizza il resto dovuto

System.out.printf("Dollar coins: %d", dollarCoins);
System.out.println();
System.out.printf("Quarters:      %d", quarters);
System.out.println();
}
}

```

### Esecuzione del programma

```

Enter bill value (1 = $1 bill, 5 = $5 bill, etc.): 5
Enter item price in pennies: 225
Dollar coins:      2
Quarters:         3

```

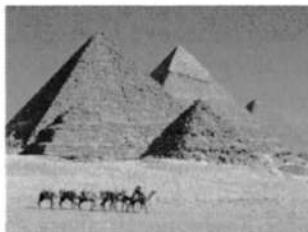


## Esempi completi 4.1

### Calcolare il volume e l'area della superficie di una piramide

In questo esempio completo svilupperemo la soluzione per un problema di calcolo.

**Problema.** Immaginate di voler essere d'aiuto agli archeologi che fanno ricerche sulle piramidi egizie e di esservi assunti il compito di scrivere un metodo che determini il volume e l'area della superficie di una piramide, di cui sia nota la misura della base e dell'altezza.



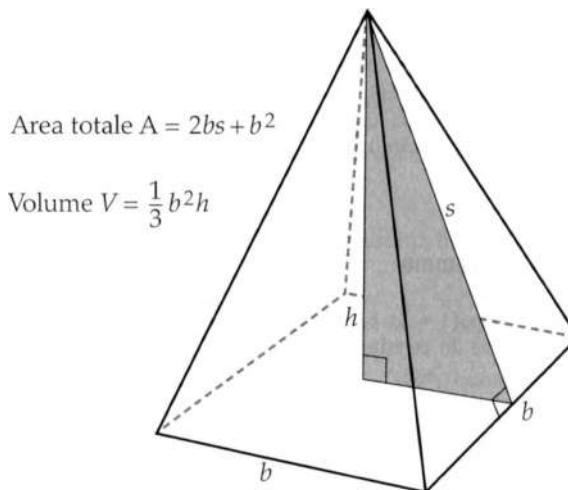
**Fase 1.** Analizzare il problema. Quali sono i dati da acquisire? E quali i dati da produrre?

Fate un elenco di tutti i valori che possono essere considerati variabili: uno degli errori più frequenti dei principianti consiste nella realizzazione di classi troppo specifiche. Ad esempio, potreste sapere che la grande piramide di Giza, la più grande delle piramidi egizie, è alta 146 metri e ha una base lunga 230 metri: *non dovete usare questi numeri* nel vostro progetto, anche se le specifiche che vi hanno fornito riguardano soltanto la grande piramide. Scrivere una classe che descrive *qualsiasi* piramide è altrettanto semplice ed estremamente più utile.

Nel nostro caso, una piramide è descritta dalla sua altezza e dalla lunghezza del lato della sua base (quadrata). I valori da calcolare sono il volume e l'area della superficie.

**Fase 2.** Risolvere alcuni esempi a mano

Una ricerca in Internet vi fornisce questo schema per i calcoli geometrici che riguardano una piramide a base quadrata:



Il calcolo del volume è davvero semplice. Considerate una piramide avente base e altezza di 10 cm ciascuna: il volume è, quindi,  $1/3 \times 10^2 \times 10 = 333.3$  cm<sup>3</sup>, cioè un terzo del volume di un cubo con lato di 10 cm. La cosa non vi meraviglia se conoscete la famosa scomposizione di un cubo in tre piramidi, individuata da Archimede.

Il calcolo della superficie è meno banale. Guardando la formula,  $A = 2bs + b^2$ , notiamo che calcola l'area dell'intera superficie, compreso il quadrato di base: è il valore che vi interessa se, ad esempio, volete calcolare quanta vernice vi serve per colorare una piramide di carta. Ma i ricercatori con cui collaborate sono realmente interessati alla base, che non è visibile? Dovete confrontarvi con loro e immaginiamo che vi rispondano che il loro interesse è relativo solamente alla parte di piramide visibile, al di fuori del suolo: in questo caso, la formula diventa  $A = 2bs$ .

Sfortunatamente, il valore  $s$  non è tra i nostri dati di ingresso, per cui lo dobbiamo calcolare. Osservate il triangolo grigio nella figura: è un triangolo rettangolo con lati  $s$ ,  $h$  e  $b/2$ . Il teorema di Pitagora ci dice che  $s^2 = h^2 + (b/2)^2$ .

Esercitiamoci nuovamente. Se  $h$  e  $b$  valgono 10, allora  $s^2$  è uguale a  $10^2 + 5^2 = 125$  e  $s$  è, quindi, uguale alla radice quadrata di 125. L'area è, quindi,  $A = 2bs = 20 \times 125^{1/2}$ , circa 224: un risultato plausibile, dato che quattro facce di un cubo con lato 10 hanno area complessivamente pari a 400, valore che ci aspettiamo sia ragionevolmente maggiore della somma delle aree delle quattro facce triangolari della nostra piramide.

Calcolato tutto questo a mano, siamo ora pronti per descrivere i calcoli in Java.

**Fase 3.** Progettare una classe che svolga i calcoli

Come visto nei Consigli pratici 3.1, dobbiamo identificare i metodi e le variabili di esemplare della nostra classe. In questo caso l'enunciazione del problema descrive questo costruttore e questi metodi:

- `public Pyramid(double height, double baseLength)`
- `public double getVolume()`
- `public double getSurfaceArea()`

Per individuare le variabili di esemplare dobbiamo ragionare un po'. Esaminate queste alternative:

- Una piramide memorizza la lunghezza della propria altezza e della propria base. Il volume e l'area vengono calcolati nei metodi `getVolume` e `getSurfaceArea`, quando necessario.
- Una piramide memorizza il proprio volume e l'area della propria superficie, eseguendo i calcoli nel costruttore a partire dai valori di `height` e `baseLength`, che vengono poi dimenticati.

Nel nostro caso entrambi gli approcci sarebbero adeguati e non c'è alcuna regola semplice per decidere quale scelta sia migliore. Per affrontare il problema, è bene valutare come possa evolvere la classe `Pyramid`: si potrebbero aggiungere ulteriori metodi per effettuare calcoli geometrici (ad esempio per conoscerne gli angoli) oppure metodi per modificarne le dimensioni. La prima delle due alternative individuate rende più semplice la realizzazione di questi scenari evolutivi e, inoltre, ci sembra maggiormente orientata agli oggetti: una piramide è descritta dalla sua base e dalla sua altezza, non dal volume e dall'area della superficie laterale.

#### Fase 4. Scrivere pseudocodice che descriva i metodi

Come già visto, il volume è semplicemente:

$$\text{volume} = (\text{base} \times \text{base} \times \text{altezza}) / 3$$

Per calcolare l'area, ci serve prima di tutto l'altezza di una faccia:

$$\text{altezza di una faccia} = \text{radice quadrata di} (\text{altezza} \times \text{altezza} + \text{base} \times \text{base} / 4)$$

Abbiamo, quindi:

$$\text{area} = 2 \times \text{base} \times \text{altezza di una faccia}$$

#### Fase 5. Realizzare la classe

Per le decisioni prese nella Fase 3, le variabili di esemplare sono l'altezza e la base:

```
public class Pyramid
{
    private double height;
    private double baseLength;
    ...
}
```

A questo punto, scrivere i metodi per il calcolo del volume e dell'area è veramente banale.

```
public double getVolume()
{
    return height * baseLength * baseLength / 3;
}

public double getSurfaceArea()
{
    double sideLength = Math.sqrt(height * height
        + baseLength * baseLength / 4);
    return 2 * baseLength * sideLength;
}
```

C'è un piccolo problema in merito al costruttore. Come detto nella Fase 3, i parametri del costruttore hanno il medesimo significato delle corrispondenti variabili di esemplare:

```
public Pyramid(double height, double baseLength)
```

Una delle possibili soluzioni prevede di modificare i nomi dei parametri del costruttore:

```
public Pyramid(double aHeight, double aBaseLength)
{
    height = aHeight;
    baseLength = aBaseLength;
}
```

Questo approccio ha un piccolo svantaggio: questi strani nomi dei parametri vanno a finire nella documentazione API:

```
/**
 * Costruisce una piramide con altezza e base assegnate.
 * @param aHeight l'altezza
 * @param aBaseLength la lunghezza di uno dei lati del quadrato di base
 */
```

Se preferite, potete aggirare il problema usando il riferimento `this`, in questo modo:

```
public Pyramid(double height, double baseLength)
{
    this.height = height;
    this.baseLength = baseLength;
}
```

A questo punto la realizzazione della classe è completa: nel pacchetto dei file scaricabili per questo libro, la cartella `worked_example_1` del Capitolo 4 contiene il codice sorgente completo della classe.

#### Fase 6. Collaudare la classe

Come programma di collaudo possiamo usare i calcoli visti nella Fase 2:

```
Pyramid sample = new Pyramid(10, 10);
```

```
System.out.println(sample.getVolume());
System.out.println("Expected: 333.33");
System.out.println(sample.getSurfaceArea());
System.out.println("Expected: 224");
```

Sarebbe, poi, preferibile collaudare la classe anche in un caso in cui l'altezza e la base hanno valori diversi, per verificare che il costruttore li utilizzi nell'ordine corretto. Una ricerca in Internet ci dice che il volume della piramide di Giza è circa due milioni e mezzo di metri cubi.

```
Pyramid gizeh = new Pyramid(146, 230);
System.out.println(gizeh.getVolume());
System.out.println("Expected: 2500000");
```

L'esecuzione del programma visualizza:

```
333.333333333333
Expected: 333.33
223.60679774997897
Expected: 224
2574466.6666666665
Expected: 2500000
```

I risultati sono molto simili a quelli previsti, per cui decidiamo che il collaudo ha avuto successo.

## 4.4 Problem Solving: prima si risolve a mano

Una fase molto importante nello sviluppo di un algoritmo consiste nel fare, per prima cosa, i calcoli *a mano*. Se non siete in grado di farlo autonomamente, è veramente improbabile che riusciate a scrivere un programma che automatizzi i calcoli.

Per illustrare questo principio, consideriamo questo problema: si vuole posizionare una striscia di piastrelle nere e bianche lungo un muro e, per motivi estetici, l'architetto ha richiesto che la prima e l'ultima piastrella siano nere.

Dovete calcolare il numero di piastrelle necessarie e lo spazio che rimane vuoto a ciascuna estremità della striscia, dato lo spazio totale disponibile e la larghezza di ciascuna piastrella.



Per fare i calcoli a mano, scegliete i valori effettivi in un caso tipico.

Per concretizzare il problema, facciamo le seguenti ipotesi sulle dimensioni:

- Ampiezza totale: 100 pollici.
- Larghezza di una piastrella: 5 pollici.

La soluzione più ovvia sarebbe quella di riempire l'intero spazio disponibile con 20 piastrelle, ma in tal modo le due piastrelle, iniziale e finale, avrebbero colori diversi, mentre è richiesto che siano uguali (ed entrambe nere).

Consideriamo, invece, il problema sotto questo punto di vista: la prima piastrella deve sempre essere nera, dopodiché aggiungiamo un certo numero di coppie di piastrelle, una bianca e una nera.



La prima piastrella occupa uno spazio di 5 pollici, lasciando 95 pollici da ricoprire con le coppie di piastrelle. Ciascuna coppia occupa uno spazio di 10 pollici, quindi il numero di coppie è  $95 / 10 = 9.5$ . Da questo numero dobbiamo ovviamente eliminare la parte frazionaria, perché non possiamo usare frazioni di coppie.

Useremo, quindi, 9 coppie di piastrelle, cioè 18 piastrelle, oltre a quella iniziale, nera: 19 piastrelle in totale.

Le piastrelle occupano uno spazio totale di  $19 \times 5 = 95$  pollici, lasciando uno spazio vuoto complessivo di  $100 - 19 \times 5 = 5$  pollici, che va distribuito equamente alle due estremità: lo spazio vuoto a ciascuna estremità è, quindi,  $(100 - 19 \times 5) / 2 = 2.5$  pollici.

Questo calcolo ci fornisce informazioni sufficienti per individuare un algoritmo che possa funzionare con qualunque valore per i due dati di partenza, lo spazio totale da coprire e la larghezza di una piastrella.

$$\text{numero di coppie} = \text{parte intera di} (\text{spazio totale} - \text{lorgh. piastr.}) / (2 \times \text{lorgh. piastr.})$$

$$\text{numero di piastr.} = 1 + 2 \times \text{numero di coppie}$$

$$\text{ciascuno spazio vuoto} = (\text{spazio totale} - \text{numero di piastr.} \times \text{lorgh. piastr.}) / 2$$

Come potete vedere, l'esecuzione manuale dei calcoli ci ha consentito di analizzare il problema al punto tale da rendere veramente facile lo sviluppo di un algoritmo generale.



### Auto-valutazione

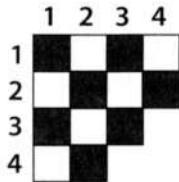
17. Traducete in Java lo pseudocodice che abbiamo scritto nel paragrafo precedente per il calcolo del numero di piastrelle e della larghezza di ciascuno spazio vuoto alle due estremità.
18. Ipotizzate che, nella situazione precedente, l'architetto avesse richiesto uno schema con piastrelle nere, grigie e bianche, come questo:



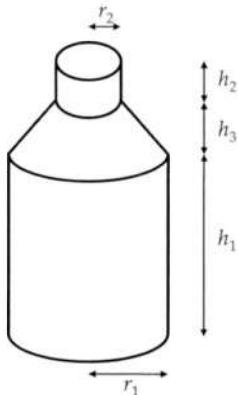
Anche in questo caso, però, la prima e l'ultima piastrella devono essere nere. Come va modificato l'algoritmo?

19. Un robot deve ricoprire un pavimento con piastrelle nere e bianche, secondo uno schema a scacchiera. Progettate un algoritmo che, data una posizione del pavimento identificata da indici di colonna e di riga, come nella figura, calcoli il colore della piastrella (0 per il

nero e 1 per il bianco). Iniziate facendo i calcoli per alcuni valori specifici degli indici, per poi generalizzare la soluzione.



20. Per una determinata automobile, i costi annuali di manutenzione sono valutati in \$100 per il primo anno, arrivando a \$1500 per il decimo anno. Ipotizzando che tali costi aumentino ogni anno della stessa quantità, progettate pseudocodice che calcoli il costo di manutenzione previsto per il terzo anno, generalizzando poi la soluzione per l'anno  $n$ -esimo.
21. La forma di una bottiglia si può approssimare con due cilindri di raggio  $r_1$  e  $r_2$ , con altezze  $h_1$  e  $h_2$ , collegati da un tronco di cono di altezza  $h_3$ .



Per calcolare il volume di un cilindro usate la formula  $V = \pi hr^2$ , mentre per il tronco di cono usate questa:

$$V = \pi \frac{(r_1^2 + r_1 r_2 + r_2^2)h}{3}$$

Progettate, mediante pseudocodice, un algoritmo che calcoli il volume di una bottiglia. Usando come esempio una vera bottiglia di cui conoscete il volume, fate i calcoli a mano usando lo pseudocodice che avete sviluppato.

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R4.18, R4.22 e R4.23, al termine del capitolo.



## Esempi completi 4.2

### Calcolare il tempo necessario per compiere un percorso

In questo esempio completo calcoleremo manualmente il tempo di viaggio per un percorso, che poi utilizzeremo per scrivere pseudocodice che esegua gli stessi calcoli, traducendolo, infine, in un programma Java.

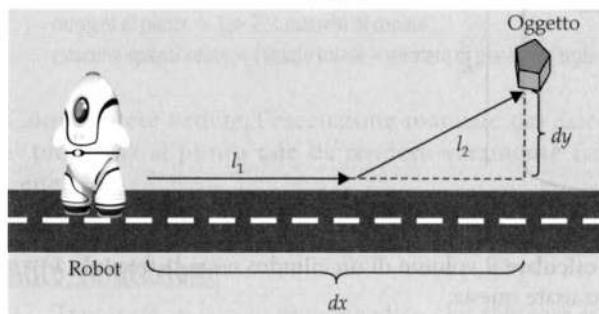
**Problema.** Un robot deve recuperare un oggetto che si trova in un terreno roccioso adiacente a una strada. Il robot, ovviamente, può procedere più velocemente sulla strada che sul terreno, per cui vuole compiere un certo percorso sulla strada, prima di abbandonarla per spostarsi in linea retta sul terreno roccioso, in direzione dell'oggetto. L'obiettivo è quello di calcolare il tempo totale necessario perché il robot raggiunga la sua destinazione finale.

**Fase 1.** Analizzare il problema. Quali sono i dati da acquisire? E quali i dati da produrre?

Questi sono i dati forniti in ingresso:

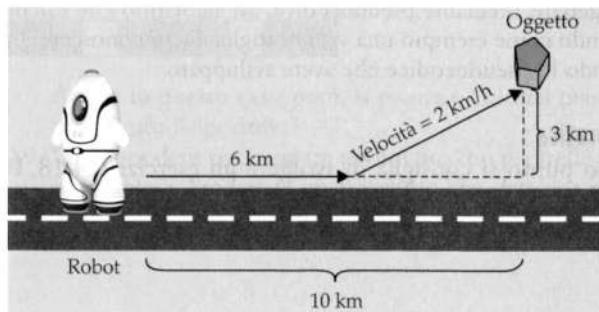
- La distanza tra il robot e l'oggetto, nelle due direzioni,  $x$  e  $y$  (rispettivamente,  $dx$  e  $dy$ ).
- La velocità del robot sulla strada ( $s_1$ ) e sul terreno roccioso ( $s_2$ ).
- La lunghezza  $l_1$  del primo tratto, percorso sulla strada.

È richiesto il calcolo del tempo di spostamento totale.



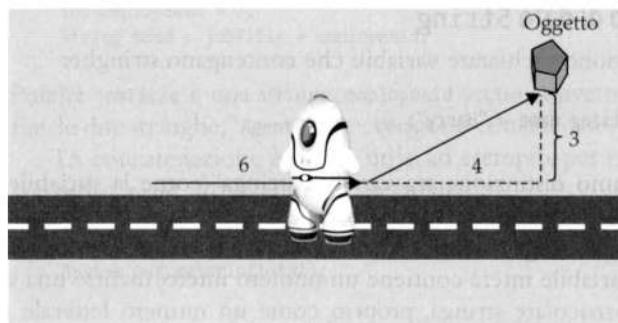
**Fase 2.** Risolvere alcuni esempi a mano

Per calcolare un esempio a mano, usiamo i valori indicati in figura:



Il tempo totale è la somma dei tempi necessari a fare i due tratti. Il tempo richiesto per il primo tratto, su strada, è semplicemente uguale alla lunghezza  $l_1$  divisa per la velocità  $s_1$ ; 6 km diviso per 5 km/h, cioè 1.2 ore.

Per il calcolo del tempo richiesto per lo spostamento sul terreno roccioso, dobbiamo prima calcolare la sua lunghezza, che è l'ipotenusa di un triangolo rettangolo avente i cateti di lunghezza 3 km e 4 km.



La sua lunghezza è, quindi, uguale alla radice quadrata di  $(3^2 + 4^2)$ , cioè 5 km. Alla velocità di 2 km/h, tale percorso richiede un tempo di 2.5 ore, portando il tempo totale a 3.7 ore.

#### Fase 3. Scrivere pseudocodice che realizzi i calcoli

Diamo ancora un'occhiata ai calcoli che abbiamo fatto: le singole fasi non dipendono dai valori utilizzati come dati, quindi li possiamo formulare come pseudocodice sostituendo i valori con variabili aventi nomi appropriati:

```
tempo1 = l1 / s1
lunghezza2 = radice quadrata di ((dx - l1)^2 + dy^2)
tempo2 = lunghezza2 / s2
tempo totale = tempo1 + tempo2
```

#### Fase 4. Tradurre in Java lo pseudocodice

Facendo i calcoli a mano è comodo usare nomi brevi per le variabili, come  $dx$  o  $s_1$ , mentre in un programma è bene modificarli in modo che siano più lunghi e descrittivi.

Tradotti in Java, i calcoli da eseguire sono:

```
double segment1Time = segment1Length / segment1Speed;
double segment2Time = Math.sqrt(
    Math.pow(xDistance - segment1Length, 2)
    + Math.pow(yDistance, 2));
double segment2Time = segment2Length / segment2Speed;
double totalTime = segment1Time + segment2Time;
```

Nel pacchetto dei file scaricabili per questo libro, la cartella `worked_example_2` del Capitolo 4 contiene il codice sorgente completo della classe.

## 4.5 Stringhe

Una stringa è una sequenza di caratteri.

Molti programmi elaborano testi, non numeri. In generale, i testi sono composti da **caratteri** (*character*): lettere, numeri, segni di punteggiatura, spazi e così via. Una **stringa** è una sequenza di caratteri, come "Harry", che è una sequenza di cinque caratteri.

### 4.5.1 Il tipo di dato String

Si possono dichiarare variabili che contengano stringhe:

```
String name = "Harry";
```

Facciamo distinzione tra variabili stringa (come la variabile `name` appena dichiarata) e stringhe **letterali**, che sono sequenze di caratteri racchiuse tra virgolette, come "Harry". Una variabile stringa è semplicemente una variabile che contiene una stringa, esattamente come una variabile intera contiene un numero intero, mentre una stringa letterale rappresenta una particolare stringa, proprio come un numero letterale (come 2) rappresenta uno specifico numero.

Il metodo `length` restituisce il numero di caratteri presenti in una stringa.

In una stringa, il numero di caratteri ne costituisce la *lunghezza*. Per esempio, la lunghezza di "Harry" è 5. Come avete visto nel Paragrafo 2.3, la lunghezza di una stringa si può calcolare usando il metodo `length`.

```
int n = name.length();
```

Una stringa di lunghezza zero, che non contiene caratteri, si chiama *stringa vuota* ("empty string") e si scrive "".

### 4.5.2 Concatenazione

Usando l'operatore + si possono concatenare stringhe, cioè porle una di seguito all'altra per formare una stringa più lunga.

Date due stringhe, come "Harry" e "Morgan", le si possono **concatenare** per formare una stringa più lunga. Il risultato consiste in una stringa contenente tutti i caratteri della prima stringa, seguiti da tutti i caratteri della seconda stringa. In Java, per concatenare due stringhe si usa l'operatore +.

Ad esempio:

```
String fName = "Harry";
String lName = "Morgan";
String name = fName + lName;
```

Si ottiene la stringa:

"HarryMorgan"

E se avessimo voluto separare con uno spazio il primo nome dal secondo? Nessun problema:

```
String name = fName + " " + lName;
```

Questo enunciato concatena tre stringhe: `fName`, la stringa letterale " " e `lName`. Il risultato è:

"Harry Morgan"

Quando l'espressione che si trova a sinistra o a destra dell'operatore + è una stringa, l'altra espressione viene automaticamente convertita in una stringa, per poi concatenarle.

Per esempio, esaminiamo questo codice:

```
String jobTitle = "Agent";
int employeeId = 7;
String bond = jobTitle + employeeId;
```

Se uno degli argomenti dell'operatore + è una stringa, l'altro viene convertito in una stringa.

Poiché jobTitle è una stringa, employeeId viene convertito dal numero 7 alla stringa "7". Poi, le due stringhe, "Agent" e "7", vengono concatenate, dando luogo alla stringa "Agent7".

La concatenazione è molto utile, ad esempio, per ridurre il numero degli enunciati System.out.print. I due enunciati seguenti:

```
System.out.print("The total is ");
System.out.println(total);
```

possono essere combinati nella singola invocazione:

```
System.out.println("The total is " + total);
```

La concatenazione "The total is " + total restituisce un'unica stringa, formata dalla stringa "The total is " seguita dalla stringa corrispondente al numero contenuto in total.

### 4.5.3 Acquisire stringhe in ingresso

Per acquisire una stringa contenente un'unica parola si usa il metodo next della classe Scanner.

Si può leggere una stringa dalla tastiera, in questo modo:

```
System.out.print("Please enter your name: ");
String name = in.next();
```

Quando si legge una stringa usando il metodo next, viene letta una sola parola. Supponiamo, ad esempio, che l'utente, in risposta alla richiesta, digitò:

Harry Morgan

Il dato in ingresso è costituito da due parole, ma l'invocazione in.next() restituisce la stringa "Harry". Per leggere la seconda parola si può usare una seconda invocazione di in.next().

### 4.5.4 Sequenze di escape

Per inserire delle virgolette all'interno di una stringa letterale, fatele precedere da un carattere di barra rovesciata (*backslash*, \), in questo modo:

```
"He said \"Hello\""
```

Il carattere *backslash* non viene inserito nella stringa: serve soltanto a segnalare che le virgolette immediatamente seguenti fanno parte della stringa e non ne contrassegnano la fine, come invece normalmente avviene. La sequenza \" si chiama **"sequenza di escape"**.

Per inserire, invece, il carattere *backslash* all'interno di una stringa, si usa un'altra sequenza di escape, `\`, in questo modo:

```
" C:\\Temp\\\\Secret.txt"
```

Un'altra sequenza di escape utilizzata spesso è `\n`, che rappresenta un carattere (chiamato **newline**, cioè “riga nuova”) la cui “visualizzazione” sullo schermo produce l'effetto di “andare a capo” e iniziare a scrivere nella riga successiva. Per esempio, l'enunciato:

```
System.out.print("*\\n**\\n***\\n");
```

visualizza quanto segue:

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

su tre righe separate.

Spesso capita anche di voler aggiungere un carattere *newline* alla fine della stringa che controlla il formato in un'invocazione di `System.out.printf`, come in questo esempio:

```
System.out.printf("Price: %10.2f\\n", price);
```

### 4.5.5 Stringhe e caratteri

In Java, le stringhe sono sequenze di caratteri **Unicode** (si veda la sezione Computer e società 4.2) e un **carattere** è un valore di tipo `char`.

I caratteri hanno valori numerici: ad esempio, consultando l'Appendice A, dove trovate i caratteri usati nelle lingue occidentali, potete verificare che il carattere 'H' viene codificato con il numero 72.

I caratteri letterali sono delimitati da singoli apici e non vanno confusi con le stringhe:

- 'H' è un carattere, cioè un valore di tipo `char`.
- "H" è una stringa (cioè un oggetto di tipo `String`) che contiene un solo carattere.

Le posizioni all'interno di una stringa sono numerate a partire da zero.

Il metodo `charAt` restituisce un valore di tipo `char` estratto da una stringa. La prima posizione all'interno della stringa è associata all'indice 0, la seconda posizione all'indice 1, e così via.

H	a	r	r	y
0	1	2	3	4

L'indice numerico che rappresenta la posizione dell'ultimo carattere (4 nella stringa "Harry") è sempre inferiore di un'unità rispetto alla lunghezza della stringa.

Ad esempio, gli enunciati seguenti assegnano a `start` il valore 'H' e a `last` il valore 'y'.

```
String name = "Harry";
```

```
char start = name.charAt(0);
char last = name.charAt(4);
```

### 4.5.6 Sottostringhe

Per estrarre una porzione  
di una stringa  
si usa il metodo `substring`.

Data una stringa, è possibile estrarne sottostringhe usando il metodo `substring`. L'invocazione

```
str.substring(start, pastEnd)
```

restituisce una stringa costituita da caratteri consecutivi della stringa `str`, iniziando dal carattere che si trova nella posizione `start`, incluso, e terminando con il carattere che si trova nella posizione `pastEnd`, escluso. Ecco un esempio:

```
String greeting = "Hello, World!";
String sub = greeting.substring(0, 5); // sub contiene "Hello"
```

In questo caso l'operazione `substring` costruisce una stringa formata da caratteri copiati dalla stringa `greeting`: i suoi primi cinque caratteri.

H	e	l	l	o	,	W	o	r	l	d	!	
0	1	2	3	4	5	6	7	8	9	10	11	12

Vediamo come si estraе, invece, la sottostringa "World". Il conteggio dei caratteri inizia da 0, non da 1, e potete constatare che `W`, l'ottavo carattere, si trova nella posizione 7. Il primo carattere che *non si vuole* copiare, il punto esclamativo, è nella posizione 12. Pertanto, l'enunciato corretto per l'estrazione della sottostringa desiderata è:

```
String sub2 = greeting.substring(7, 12);
```

H	e	l	l	o	,	W	o	r	l	d	!	
0	1	2	3	4	5	6	7	8	9	10	11	12

5

È curioso che si debba specificare la posizione del primo carattere desiderato e, poi, quella del primo che non si vuole. Questa strategia ha un vantaggio: potete calcolare facilmente la lunghezza della sottostringa mediante l'operazione `pastEnd - start`. Per esempio, la stringa "World" ha lunghezza  $12 - 7 = 5$ .

Se nell'invocazione del metodo `substring` evitate di fornire il secondo parametro, vengono copiati tutti caratteri dalla posizione iniziale indicata come unico parametro fino alla fine della stringa. Per esempio, dopo l'esecuzione dell'enunciato:

```
String tail = greeting.substring(7); // copia dalla posizione 7 alla fine
```

`tail` contiene la stringa "World!".

Nel seguito presentiamo un semplice programma che sfrutta i concetti appena visti: il programma chiede il vostro nome e quello del vostro partner, poi visualizza le vostre iniziali.

L'operazione `first.substring(0, 1)` costruisce una stringa costituita da un unico carattere, il carattere iniziale di `first`, dopodiché il programma compie la stessa elaborazione

con la stringa `second`, per poi concatenare le due stringhe di lunghezza unitaria in modo da ottenere una stringa unica, `initials`, come si può vedere nella Figura 3.

**Figura 3**

Costruzione della stringa  
initials

```
first = R | o | d | o | l | f | o
      0   1   2   3   4   5   6

second = S | a | l | l | y
        0   1   2   3   4

initials = R | & | S |
          0   1   2
```

### File Initials.java

```
1 import java.util.Scanner;
2
3 /**
4     Programma che visualizza una coppia di iniziali.
5 */
6 public class Initials
7 {
8     public static void main(String[] args)
9     {
10         Scanner in = new Scanner(System.in);
11
12         // leggi i nomi della coppia
13
14         System.out.print("Enter your first name: ");
15         String first = in.next();
16         System.out.print("Enter your significant other's first name: ");
17         String second = in.next();
18
19         // calcola e visualizza le iniziali
20
21         String initials = first.substring(0, 1)
22             + "&" + second.substring(0, 1);
23         System.out.println(initials);
24     }
25 }
```

### Esecuzione del programma

```
Enter your first name: Rodolfo
Enter your significant other's first name: Sally
R&S
```



### Auto-valutazione

22. Che lunghezza ha la stringa "Java Program"?
23. Considerate la seguente variabile stringa:  
`String str = "Java Program";`  
Scrivete un'invocazione del metodo `substring` che restituisca la sottostringa "gram".

24. Usate la concatenazione per trasformare la variabile stringa str della domanda precedente nella stringa "Java Programming".
25. Cosa visualizza questa sequenza di enunciati?
- ```
String str = "Harry";
int n = str.length();
String mystery = str.substring(0, 1) + str.substring(n - 1, n);
System.out.println(mystery);
```
26. Scrivete una sequenza di enunciati che sia in grado di acquisire un nome come "John Q. Public".

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R4.10, R4.14, E4.15 e P4.7, al termine del capitolo.

**Tabella 7** Operazioni con stringhe

| Enunciati                                                                              | Risultato                                                       | Commento                                                                                                            |
|----------------------------------------------------------------------------------------|-----------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|
| string str = "Ja";<br>str = str + "va";                                                | Assegna "Java" a str.                                           | Quando viene applicato a stringhe, l'operatore + indica concatenazione.                                             |
| System.out.println("Please" + " enter your name: ");                                   | Visualizza<br>Please enter your name:                           | Concatenazione usata per scomporre una stringa che non trovava posto in una sola riga.                              |
| team = 49 + "ers"                                                                      | Assegna "49ers" a team.                                         | Dato che "ers" è una stringa, 49 viene convertito in una stringa.                                                   |
| String first = in.next();<br>String last = in.next();<br>(Utente digita: Harry Morgan) | Assegna "Harry" a first<br>e "Morgan" a second.                 | Il metodo next memorizza in una variabile stringa la parola successiva.                                             |
| String greeting = "H & S";<br>int n = greeting.length();                               | Assegna 5 a n.                                                  | Anche gli spazi vengono contati come caratteri della stringa.                                                       |
| String str = "Sally";<br>char ch = str.charAt(1);                                      | Assegna 'a' a ch.                                               | È un valore di tipo char, non una stringa. Si noti che la posizione iniziale ha indice 0.                           |
| String str = "Sally";<br>String str2 = str.substring(1, 4);                            | Assegna "all" a str2.                                           | Estrae la sottostringa che inizia nella posizione 1 e termina prima della posizione 4.                              |
| String str = "Sally";<br>String str2 = str.substring(1);                               | Assegna "ally" a str2.                                          | Se si omette la posizione finale, vengono presi tutti i caratteri dalla posizione indicata alla fine della stringa. |
| String str = "Sally";<br>String str2 = str.substring(1, 2);                            | Assegna "a" a str2.                                             | Estrae una sottostringa di lunghezza 1: confrontate con str.charAt(1).                                              |
| String last = str.substring(str.length() - 1);                                         | Assegna a last la stringa contenente l'ultimo carattere di str. | L'ultimo carattere ha sempre posizione uguale a str.length() - 1.                                                   |



### Suggerimenti per la programmazione 4.3

#### Leggere le segnalazioni di eccezioni

Vi capiterà spesso che i vostri programmi terminino visualizzando un messaggio d'errore come questo:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
        String index out of range: -4
        at java.lang.String.substring(String.java:1444)
        at Homework1.main(Homework1.java:16)
```

Se vi succede, non dite "non funziona", oppure "il mio programma è morto": piuttosto, leggete il messaggio d'errore. A dire il vero, il formato della segnalazione di eccezione non è molto amichevole, ma in realtà è facile decifrarlo.

Guardando attentamente il messaggio d'errore, noterete due informazioni importanti:

1. Il nome dell'eccezione, in questo caso `StringIndexOutOfBoundsException`
2. Il numero della riga del file sorgente che contiene l'enunciato che ha causato l'eccezione, in questo caso `Homework1.java:16`

Il nome dell'eccezione è sempre nella prima riga della segnalazione e termina con `Exception`. Se la segnalazione è `StringIndexOutOfBoundsException`, allora c'è stato un problema relativamente all'accesso a una posizione non valida all'interno di una stringa: sicuramente un'informazione utile.

Il numero della riga errata nel file sorgente è un po' più difficile da determinare, perché la segnalazione di eccezione contiene **l'intera traccia della pila di esecuzione** (*stack trace*), cioè il nome di tutti i metodi la cui esecuzione era sospesa nel momento in cui si è verificata l'eccezione. La prima riga della traccia riguarda il metodo che ha effettivamente generato l'eccezione, mentre l'ultima riga della traccia fa riferimento a una riga del metodo `main`. Capita spesso che l'eccezione venga lanciata da un metodo che si trova nella libreria standard: dovete cercare la prima riga del *vostro codice* che appare nella segnalazione di eccezione. Ad esempio, saltate la riga che si riferisce a

```
java.lang.String.substring(String.java:1444)
```

La riga successiva, nel nostro esempio, riporta un numero di riga relativo al vostro codice, `Homework1.java`. Dopo aver individuato il numero di riga nel vostro codice, aprirete il file, andate a quella riga e osservatela! E guardate anche il nome dell'eccezione. Nella maggioranza dei casi, queste due informazioni rendono assolutamente evidente il problema, permettendovi di correggerlo facilmente.

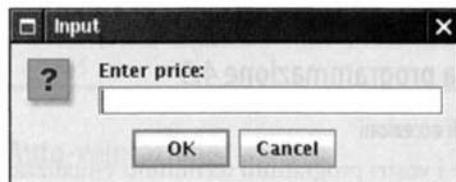


## Argomenti avanzati 4.4

### Acquisizione e visualizzazione di dati con una finestra di dialogo

La maggior parte degli utenti di programmi ritiene che le finestre di console siano decisamente obsolete: l'alternativa più semplice consiste nella creazione di una diversa finestra di dialogo per ogni valore da richiedere in ingresso.

Una finestra di dialogo per acquisire dati



Invocate il metodo statico `showInputDialog` della classe `JOptionPane`, fornendo come parametro la stringa da usare come `prompt` per chiedere un dato all'utente. Ad esempio:

```
String input = JOptionPane.showInputDialog("Enter price:");
```

Tale metodo restituisce un oggetto di tipo `String`, contenente quanto scritto dall'utente. Dato che spesso si ha bisogno di ricevere dati numerici, usate poi i metodi `Integer.parseInt` o `Double.parseDouble` per convertire la stringa in un numero:

```
double price = Double.parseDouble(input);
```

È anche possibile visualizzare dati in una finestra dello stesso tipo:

```
JOptionPane.showMessageDialog(null, "Price: " + price);
```



## Computer e società 4.2

### Alfabeti internazionali e Unicode

L'alfabeto inglese è piuttosto semplice: contiene le lettere dalla *a* alla *z*, maiuscole e minuscole. Altre lingue europee usano accenti e caratteri speciali: per esempio, il tedesco ha tre caratteri cosiddetti *umlaut* (ä, ö, ü) e una doppia s, il carattere ß. Non si tratta di fronzoli opzionali: non potreste mai scrivere una pagina in tedesco senza usare questi caratteri e, infatti, le tastiere dei computer tedeschi hanno tasti appositi per questi caratteri.



Una tastiera tedesca

Molti paesi non usano nemmeno l'alfabeto latino. I caratteri russi, greci, ebraici, arabi e thai, per citarne solo alcuni, hanno una forma completamente diversa. Per complicare le cose, lingue come l'ebraico e l'arabo si scrivono da destra a sinistra invece



Ebraico, arabo e inglese

che da sinistra a destra. Ciascuno di questi alfabeti ha un numero di caratteri simile a quello dell'alfabeto inglese.



Ideogrammi cinesi

A partire dal 1987, un consorzio di produttori di hardware e di software ha sviluppato uno schema uniforme di codifica chiamato **Unicode**, appositamente progettato per essere in grado di codificare un testo in tutte le lingue scritte del mondo. Una prima versione di Unicode usava 16 bit per codificare ciascun carattere e il tipo `char` in Java corrisponde a tale codifica.

Oggi Unicode è cresciuto fino a utilizzare un codice a 21 bit, con definizioni per oltre 100000 caratteri diversi ([www.unicode.org](http://www.unicode.org)). Si pensa, poi, di aggiungere la codifica di lingue estinte, come i geroglifici egizi. Sfortunatamente, questo significa che, in Java, un carattere Unicode non sempre corrisponde a un valore di tipo `char`: alcuni caratteri di lingue come il cinese o l'antico egizio occupano due valori di tipo `char`.

Le lingue giapponese, coreana e cinese usano caratteri ideografici: un carattere rappresenta un concetto o una cosa e non un singolo suono, con parole composte da uno o più di tali caratteri. Sono noti oltre 70000 ideogrammi.

## Riepilogo degli obiettivi di apprendimento

### Scelta dei tipi appropriati per la rappresentazione di dati numerici

- Java ha otto tipi primitivi, fra i quali quattro tipi numerici interi e due tipi numerici in virgola mobile.
- Un calcolo numerico trabocca se il risultato esce dall'intervallo caratteristico del tipo numerico.
- Quando non è possibile trovare una rappresentazione numerica esatta di un numero in virgola mobile si ha un errore di arrotondamento.
- Una variabile final è una costante: dopo aver ricevuto un valore, non può più essere modificata.
- Date un nome alle costanti, per rendere i vostri programmi più facili da leggere e da modificare.

### Espressioni aritmetiche in Java

- Se un'espressione aritmetica contiene alcuni valori interi e altri in virgola mobile, il risultato è un valore in virgola mobile.
- Gli operatori ++ e -- incrementano e, rispettivamente, decrementano di un'unità il valore di una variabile.
- Se entrambi gli argomenti dell'operatore / sono di tipo intero, il risultato è un numero intero e il resto viene ignorato.
- L'operatore % calcola il resto di una divisione intera.
- La libreria di Java contiene molte funzioni matematiche; ad esempio, Math.sqrt e Math.pow calcolano, rispettivamente, radici quadrate e potenze.
- Per convertire un valore in un tipo diverso si usa il cast (*nomeTipo*).

### Acquisire e visualizzare dati

- Per acquisire dati tramite la tastiera, all'interno di una finestra di console, si usa la classe Scanner.
- Per specificare come vanno visualizzati i valori si usa il metodo printf.

### Fare calcoli a mano per lo sviluppo di un algoritmo

- Per fare i calcoli a mano, scegliete i valori effettivi in un caso tipico.

### Elaborazione di stringhe

- Una stringa è una sequenza di caratteri.
- Il metodo length restituisce il numero di caratteri presenti in una stringa.
- Usando l'operatore + si possono concatenare stringhe, cioè porle una di seguito all'altra per formare una stringa più lunga.
- Se uno degli argomenti dell'operatore + è una stringa, l'altro viene convertito in una stringa.
- Per acquisire una stringa contenente un'unica parola si usa il metodo next della classe Scanner.
- Le posizioni all'interno di una stringa sono numerate a partire da zero.
- Per estrarre una porzione di una stringa si usa il metodo substring.

## Elementi di libreria presentati in questo capitolo

|                     |          |
|---------------------|----------|
| java.io.PrintStream | abs      |
| printf              | ceil     |
| java.lang.Double    | cos      |
| parseDouble         | exp      |
| java.lang.Integer   | floor    |
| MAX_VALUE           | floorMod |
| MIN_VALUE           | log      |
| parseInt            | log10    |
| java.lang.Math      | max      |
| PI                  | min      |

|                      |                         |
|----------------------|-------------------------|
| pow                  | add                     |
| round                | multiply                |
| sin                  | subtract                |
| sqrt                 | java.math.BigInteger    |
| tan                  | add                     |
| toDegrees            | multiply                |
| toRadians            | subtract                |
| java.lang.String     | java.util.Scanner       |
| charAt               | next                    |
| length               | nextDouble              |
| substring            | nextInt                 |
| java.lang.System     | javax.swing.JOptionPane |
| in                   | showInputDialog         |
| java.math.BigDecimal | showMessageDialog       |

## Esercizi di riepilogo e approfondimento

- ★ **R4.1.** Scrivete in Java enunciati che dichiarino variabili adatte a memorizzare i dati descritti nel seguito, scegliendo in modo appropriato tra numeri interi e numeri in virgola mobile ed eventualmente dichiarando costanti quando tale scelta sia ritenuta adeguata.
- Il numero di giorni in una settimana
  - Il numero di giorni che mancano alla fine del semestre
  - Il numero di centimetri in un pollice
  - L'altezza in centimetri della persona più alta della classe
- ★ **R4.2.** Che valore ha `mystery` dopo l'esecuzione di questa sequenza di enunciati?

```
int mystery = 1;
mystery = 1 - 2 * mystery;
mystery = mystery + 1;
```

- ★ **R4.3.** Che cosa c'è di sbagliato in questa sequenza di enunciati?

```
int mystery = 1;
mystery = mystery + 1;
int mystery = 1 - 2 * mystery;
```

- ★★ **R4.4.** Scrivete in notazione matematica le seguenti espressioni Java.

- $dm = m * (\text{Math.sqrt}(1 + v / c) / \text{Math.sqrt}(1 - v / c) - 1);$
- $volume = \text{Math.PI} * r * r * h;$
- $volume = 4 * \text{Math.PI} * \text{Math.pow}(r, 3) / 3;$
- $z = \text{Math.sqrt}(x * x + y * y);$

- ★★ **R4.5.** Scrivete in Java le seguenti espressioni matematiche.

$$s = s_0 + v_0 t + \frac{1}{2} g t^2$$

$$G = 4\pi^2 \frac{a^3}{P^2(m_1 + m_2)}$$

$$FV = PV \cdot \left(1 + \frac{\text{INT}}{100}\right)^{\text{YRS}}$$

$$c = \sqrt{a^2 + b^2 - 2ab \cos \gamma}$$

- \*\* R4.6. Ipotizzando che `a` e `b` siano variabili di tipo `int`, completate la tabella.

| <code>a</code> | <code>b</code> | <code>Math.pow(a, b)</code> | <code>Math.max(a, b)</code> | <code>a / b</code> | <code>a % b</code> | <code>Math.floorMod(a, b)</code> |
|----------------|----------------|-----------------------------|-----------------------------|--------------------|--------------------|----------------------------------|
| 2              | 3              |                             |                             |                    |                    |                                  |
| 3              | 2              |                             |                             |                    |                    |                                  |
| 2              | -3             |                             |                             |                    |                    |                                  |
| 3              | -2             |                             |                             |                    |                    |                                  |
| -3             | 2              |                             |                             |                    |                    |                                  |
| -3             | -2             |                             |                             |                    |                    |                                  |

- \*\* R4.7. Immaginate che `direction` sia la misura di un angolo, espressa come numero intero compreso tra 0 e 359 gradi. Dopo aver compiuto una rotazione di `turn` gradi, la variabile `direction` viene aggiornata in questo modo:

```
direction = (direction + turn) % 360;
```

In quali situazioni si ottiene un risultato sbagliato? Come si può correggere l'errore senza usare il metodo `Math.floorMod` descritto nella sezione Note per Java 8 4.1?

- \*\* R4.8. Quali sono i valori delle seguenti espressioni? Per ciascuna espressione usate questi valori:

```
double x = 2.5;
double y = -1.5;
int m = 18;
int n = 4;
```

- a.  $x + n * y - (x + n) * y$
- b.  $m / n + m \% n$
- c.  $5 * x - n / 5$
- d.  $1 - (1 - (1 - (1 - (1 - n))))$
- e. `Math.sqrt(Math.sqrt(n))`

- \* R4.9. Ipotizzando che `n` abbia il valore 17 e che `m` abbia il valore 18, quali sono i valori delle seguenti espressioni?

- a.  $n / 10 + n \% 10$
- b.  $n \% 2 + m \% 2$
- c.  $(m + n) / 2$
- d.  $(m + n) / 2.0$
- e. `(int) (0.5 * (m + n))`
- f. `(int) Math.round(0.5 * (m + n))`

- \*\* R4.10. Quali sono i valori delle seguenti espressioni? Per ciascuna espressione usate questi valori:

```
String s = "Hello";
String t = "World";
```

- a. `s.length() + t.length()`
- b. `s.substring(1, 2)`
- c. `s.substring(s.length() / 2, s.length())`
- d. `s + t`
- e. `t + s`

- \* R4.11. Individuate almeno cinque errori di compilazione nel programma seguente.

```
public class HasErrors
{
    public static void main();
```

```

{
    System.out.print("Please enter two numbers:");
    x = in.readDouble();
    y = in.readDouble();
    System.out.println("The sum is " + x + y);
}
}

```

- \*\* **R4.12.** Individuate almeno tre errori di esecuzione nel programma seguente.

```

public class HasErrors
{
    public static void main(String[] args)
    {
        int x = 0;
        int y = 0;
        Scanner in = new Scanner("System.in");
        System.out.print("Please enter an integer:");
        x = in.readInt();
        System.out.print("Please enter another integer: ");
        x = in.readInt();
        System.out.println("The sum is " + x + y);
    }
}

```

- \*\* **R4.13.** Esaminate il codice seguente:

```

CashRegister register = new CashRegister();
register.recordPurchase(19.93);
register.receivePayment(20, 0, 0, 0, 0);
System.out.print("Change: ");
System.out.println(register.giveChange());

```

Il programma visualizza il resto come 0.0700000000000028: spiegate perché. Date consigli per migliorare il programma in modo che gli utenti non rimangano perplessi.

- \* **R4.14.** Spiegate la differenza fra 2, 2.0, '2', "2" e "2.0".
- \* **R4.15.** Spiegate cosa calcolano i due seguenti frammenti di programma:
  - a. x = 2;  
y = x + x;
  - b. s = "2";  
t = s + s;
- \*\* **R4.16.** Scrivete pseudocodice per un programma che legga una parola e ne visualizzi il primo carattere, l'ultimo carattere e i caratteri che si trovano tra i due estremi. Ad esempio, se la parola acquisita in ingresso fosse Harry, il programma dovrebbe visualizzare H y arr.
- \*\* **R4.17.** Scrivete pseudocodice per un programma che legga un nome (come Harold James Morgan) e visualizzi un monogramma costituito dalle lettere iniziali del primo nome, del secondo nome e del cognome (ad esempio HJM).
- \*\*\* **R4.18.** Scrivete pseudocodice per un programma che calcoli la prima e l'ultima cifra di un numero. Ad esempio, se il numero è 23456, il programma deve visualizzare 2 e 6. Suggerimento: usate % e Math.log10.
- \* **R4.19.** Modificate lo pseudocodice del programma sviluppato nella sezione Consigli pratici 4.1 in modo che fornisca il resto usando monete da un quarto di dollaro (*quarter*), da dieci centesimi

(dime) e da cinque centesimi (nickel), ipotizzando che il prezzo sia un multiplo di 5 centesimi. Per sviluppare lo pseudocodice fate prima un paio di esempi elaborando valori specifici.

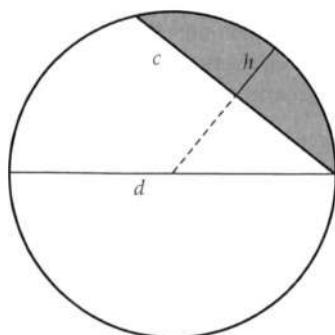
- \*\*\* **R4.20.** Per utilizzare il programma sviluppato nella sezione Esempi completi 4.1, è abbastanza facile misurare la lunghezza del lato di base di una piramide. Per misurarne l'altezza senza arrampicarsi fino in cima si può usare un teodolite e determinare l'angolo tra il terreno e la linea ideale che congiunge la posizione del teodolite e la cima della piramide. Quali altre informazioni servono per calcolare l'area della superficie della piramide?
- \*\*\* **R4.21.** Nell'ipotesi che un'antica civiltà abbia costruito piramidi a base circolare (cioè coni), scrivete pseudocodice per un programma che ne determini l'area superficiale a partire da misurazioni che si possano effettuare dal suolo.
- \*\* **R4.22.** Uno shaker per cocktail si può considerare costituito da tre tronchi di cono. Usando valori realistici per i raggi e le altezze, calcolate il volume totale del contenitore, usando la formula presentata nella domanda di auto-valutazione 21 per un singolo tronco di cono, poi sviluppate un algoritmo che funzioni con qualsiasi insieme di misure.



- \*\*\* **R4.23.** Si vuol tagliare una fetta di torta, come indicato in figura, dove  $c$  è la lunghezza del taglio (tecnicamente, una corda della circonferenza) e  $h$  è la "larghezza" della fetta. Per calcolarne l'area si può usare la seguente formula, approssimata:

$$A = \frac{2}{3}ch + \frac{h^3}{2c}$$

Va notato, però, che  $h$  non è facile da misurare, mentre solitamente il diametro  $d$  della torta è noto. Calcolate l'area della fetta nel caso in cui il diametro sia 12 pollici e la corda sia 10 pollici, poi generalizzate l'algoritmo in modo che calcoli l'area per qualsiasi valore del diametro e della corda.



- \*\* **R4.24.** Lo pseudocodice seguente descrive come calcolare il nome di un giorno in inglese, dato il suo indice (0 = Sunday, 1 = Monday, e così via).

Definire la stringa names = "SunMonTueWedThuFriSat"  
 Calcolare la posizione iniziale pos =  $3 \cdot$  indice del giorno  
 Estrarre da names la sottostringa di lunghezza 3 a partire da pos

Verificate la correttezza di questo pseudocodice usando il valore 4 come indice del giorno. Tracciate uno schema della stringa che viene calcolata, in analogia con la Figura 3.

- \*\*\* **R4.25.** Lo pseudocodice seguente descrive come scambiare due lettere in una parola.

Dati: la stringa str e due posizioni, i e j (i precede j)  
 first = sottostringa di str dall'inizio fino alla posizione che precede i  
 charl = sottostringa di str contenente il solo carattere in posizione i  
 middle = sottostringa di str da  $i + 1$  a  $j - 1$   
 charJ = sottostringa di str contenente il solo carattere in posizione j  
 last = sottostringa di str da  $j + 1$  fino alla fine della stringa  
 Concatenare, nell'ordine: first, charJ, middle, charl, last

Verificate la correttezza di questo pseudocodice usando la stringa "Gateway" e le posizioni 2 e 4. Tracciate uno schema della stringa che viene calcolata, in analogia con la Figura 3.

- \*\* **R4.26.** Come si ottiene il primo carattere di una stringa? E l'ultimo? Come si elimina il primo carattere da una stringa? E l'ultimo?

- \*\* **R4.27.** Per ciascuno dei seguenti calcoli eseguiti in Java, determinate se il risultato sarà esatto oppure se si verificherà un overflow o un errore di arrotondamento:

- $2.0 - 1.1$
- $1.0E6 * 1.0E6$
- $65536 * 65536$
- $1_000_000L * 1_000_000L$

- \*\*\* **R4.28.** Scrivete un programma che visualizzi il risultato delle espressioni seguenti e fornite spiegazioni per quanto accade:

$$\begin{aligned} & 3 * 1000 * 1000 * 1000 \\ & 3.0 * 1000 * 1000 * 1000 \end{aligned}$$

## Esercizi di programmazione

- \* **E4.1.** Scrivete un programma che visualizzi le dimensioni in millimetri di un foglio di carta in formato "letter" (8.5 · 11 pollici, un pollice = 25.4 millimetri). Usate le costanti opportune e commentate adeguatamente il codice.
- \* **E4.2.** Scrivete un programma che calcoli e visualizzi il perimetro e la lunghezza della diagonale di un foglio di carta in formato "letter" (8.5 · 11 pollici).
- \* **E4.3.** Scrivete un programma che legga un numero e ne visualizzi il quadrato, il cubo e la quarta potenza. Usate il metodo `Math.pow` soltanto per calcolare la quarta potenza.
- \*\* **E4.4.** Scrivete un programma che chieda all'utente di fornire due numeri interi e che poi ne stampi:

- La somma
- La differenza
- Il prodotto
- Il valore medio
- La distanza (cioè il valore assoluto della differenza)
- Il valore massimo (cioè il valore più grande dei due)
- Il valore minimo (cioè il valore più piccolo dei due)

*Suggerimento:* nella classe `Math` sono presenti i metodi `max` e `min`.

- \*\* **E4.5.** Migliorate la soluzione dell'esercizio precedente in modo che i numeri vengano visualizzati con l'incolonnamento opportuno, come in questo esempio:

|             |       |
|-------------|-------|
| Sum:        | 45    |
| Difference: | -5    |
| Product:    | 500   |
| Average:    | 22.50 |
| Distance:   | 5     |
| Maximum:    | 25    |
| Minimum:    | 20    |

- \*\* **E4.6.** Scrivete un programma che chieda all'utente una misura in metri e che poi la converta in miglia, piedi e pollici.

- \* **E4.7.** Scrivete un programma che chieda all'utente la lunghezza del raggio e visualizzi:

- L'area e la circonferenza di un cerchio avente tale raggio
- Il volume e l'area superficiale di una sfera avente tale raggio

- \*\* **E4.8.** Scrivete un programma che chieda all'utente le lunghezze dei lati di un rettangolo e visualizzi:

- L'area e il perimetro del rettangolo
- La lunghezza della sua diagonale (usando il teorema di Pitagora)

- \* **E4.9.** Migliorate il programma presentato nella sezione Consigli pratici 4.1 in modo che consenta all'utente di introdurre anche monete da un quarto di dollaro e non soltanto banconote.

- \*\* **E4.10.** Scrivete un programma che chieda all'utente di fornire:

- Il numero di galloni di carburante presenti nel serbatoio di un'automobile
- L'efficienza del motore in miglia percorse per gallone
- Il prezzo del carburante (in dollari al gallone)

Poi, visualizzate la spesa necessaria per percorrere 100 miglia e la distanza percorribile con il carburante rimasto ancora nel serbatoio.

- \*\* **E4.11.** Modificate la classe `Menu` vista nella sezione Esempi Completati 3.1 in modo che le opzioni del menu abbiano come etichette A, B, C e così via. *Suggerimento:* costruite una stringa con le etichette.

- \* **E4.12. Nomi di file e loro estensioni.** Scrivete un programma che chieda all'utente di fornire la lettera che identifica il disco su cui si trova un file (ad esempio C), il percorso da seguire per trovarlo nel *file system* (ad esempio `\Windows\System`), il nome del file (come `Readme`) e la sua estensione (`.txt`). Poi, visualizzate il nome completo del file, che in questo esempio è `C:\Windows\System\Readme.txt`

(se usate un sistema Unix o Macintosh, ignorate la lettera che identifica il disco e usate / invece di \ per separare tra loro i nomi delle cartelle).

- \*\*\* **E4.13.** Scrivete un programma che legga un numero compreso tra 1000 e 999999, dopo averlo chiesto all'utente, il quale inserirà il numero usando la virgola per separare le migliaia, secondo l'uso anglosassone (es. 10,000 oppure 99,999). Poi, visualizzate il numero senza la virgola. Per chiarire meglio il problema, vediamo un esempio di dialogo tra il programma e l'utente (i dati forniti dall'utente sono riportati in grassetto):

```
Please enter an integer between 1,000 and 999,999: 23,456
23456
```

*Suggerimento.* Acquisite il dato sotto forma di stringa e misuratela. Diciamo che contenga  $n$  caratteri: estraete le sottostringhe contenenti i primi  $n - 4$  caratteri e gli ultimi tre caratteri.

- \*\* **E4.14.** Scrivete un programma che legga un numero compreso tra 1000 e 999999, dopo averlo chiesto all'utente, per poi visualizzarlo con una virgola come separatore delle migliaia, secondo l'uso anglosassone. Per chiarire meglio il problema, vediamo un esempio di dialogo tra il programma e l'utente (i dati forniti dall'utente sono riportati in grassetto):

```
Please enter an integer between 1000 and 999999: 23456
23,456
```

- \* **E4.15.** *Visualizzare una scacchiera.* Scrivete un programma che visualizzi una scacchiera per giocare a tris (in inglese, *tic-tac-toe*), come questa:

```
+---+---+
| | | |
+---+---+
| | | |
+---+---+
| | | |
+---+---+
```

Potreste ovviamente risolvere il problema in modo banale, scrivendo sette enunciati di questo tipo:

```
System.out.println("+-+-+-+");
```

ma cercate, invece, di farlo in modo più elegante e flessibile. Dichiarate variabili stringa che memorizzino due schemi diversi: una specie di pettine (come quello costituito dalle prime due righe della scacchiera) e l'ultima riga. Visualizzate per tre volte lo schema a pettine, concludendo con una visualizzazione dell'ultima riga.

- \*\* **E4.16.** Scrivete un programma che legga un numero intero di cinque cifre e lo scomponga in una sequenza delle sue singole cifre. Ad esempio, ricevendo in ingresso il numero 16384, il programma deve visualizzare:

```
1 6 3 8 4
```

Potete ipotizzare che il numero fornito non sia negativo e non abbia più di cinque cifre.

- \*\* **E4.17.** Scrivete un programma che legga due orari in formato militare (ad esempio, 0900 o 1730) e visualizzi il numero di ore e di minuti che li separano nel tempo, come nel seguente esempio di esecuzione (con i dati introdotti dall'utente in grassetto):

```
Please enter the first time: 0900
Please enter the second time: 1730
8 hours 30 minutes
```

Fate attenzione alla gestione dei casi in cui il primo orario è inferiore al secondo:

```
Please enter the first time: 1730
Please enter the second time: 0900
15 hours 30 minutes
```

- \*\*\* E4.18. *Scrivere lettere giganti.* Una lettera H gigante si può visualizzare in questo modo:

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

e può essere ottenuta dichiarando una stringa letterale come questa:

```
final String LETTER_H = "*\n* *\n*****\n* *\n* \n";
```

(ricordate che la sequenza di escape `\n` rappresenta il carattere *newline*, dopo il quale i caratteri verranno scritti a partire da una riga nuova). Definite stringhe simili per le lettere E, L e O. Poi, scrivete il seguente messaggio con lettere giganti:

```
H
E
L
L
O
```

- \*\* E4.19. Scrivete un programma che traduca i numeri da 1 a 12 nei nomi dei mesi corrispondenti, in inglese: January, February, March, ..., December. Suggerimento. Create una stringa molto lunga che contenga i nomi di tutti i mesi ("January February March..."), nella quale inserirete spazi in modo che ciascun nome di mese abbia *la stessa lunghezza*. Poi, usate il metodo `substring` per estrarre da questa i soli caratteri del mese richiesto.
- \*\* E4.20. Scrivete un programma che visualizzi un albero di Natale, come questo (ricordatevi di usare le sequenze di escape):

```
      ^
     / \
    /   \
   /     \
  -----
  " "
  " "
  " "
```

- \*\* E4.21. Migliorate la classe `CashRegister` aggiungendo i metodi `enterDollars`, `enterQuarters`, `enterDimes`, `enterNickels` e `enterPennies`. Usate questa classe di collaudo:

```
public class CashRegisterTester
{
    public static void main(String[] args)
    {
        CashRegister register = new CashRegister();
        register.recordPurchase(20.37);
        register.enterDollars(20);
        register.enterQuarters(2);
        System.out.println("Change: " + register.giveChange());
```

```
        System.out.println("Expected: 0.13");
    }
}
```

- \*\* **E4.22.** Realizzate la classe `IceCreamCone` (cono gelato) con i metodi `getSurfaceArea()` e `getVolume()`. Nel costruttore specificate l'altezza e il raggio del cono. Fate attenzione nell'utilizzare la formula che calcola l'area superficiale: dovreste considerare solamente l'area laterale del cono, che è aperto in cima per contenere il gelato.
- \*\* **E4.23.** Realizzate la classe `SodaCan` (*lattina di bibita*) il cui costruttore riceve l'altezza e il diametro della lattina. Progettate i metodi `getVolume` e `getSurfaceArea`, nonché la classe `SodaCanTester` per il suo collaudo.
- \*\*\* **E4.24.** Realizzate la classe `Balloon` che rappresenti il modello di un pallone di forma sferica riempito d'aria, il cui costruttore costruisce un pallone vuoto. Dotatela di questi metodi:
- `void addAir(double amount)` aggiunge la quantità d'aria specificata
  - `double getVolume()` restituisce il volume attuale
  - `double getSurfaceArea()` restituisce l'area superficiale attuale
  - `double getRadius()` restituisce il raggio attuale

Progettate, poi, la classe di collaudo `BalloonTester` che costruisca un pallone, vi aggiunga 100 cm<sup>3</sup> d'aria, collaudi i tre metodi di accesso, aggiunga altri 100 cm<sup>3</sup> di aria e collaudi nuovamente i tre metodi di accesso.

**Sul sito web dedicato al libro si trova una raccolta di progetti di programmazione più complessi.**



# 5

## Decisioni



### Obiettivi del capitolo

- Saper realizzare decisioni usando enunciati if
- Confrontare numeri interi, numeri in virgola mobile e stringhe
- Scrivere enunciati usando il tipo di dato booleano
- Progettare strategie per il collaudo dei programmi
- Determinare se i dati ricevuti in ingresso sono validi

Una delle caratteristiche essenziali dei programmi per calcolatore è la loro capacità di prendere decisioni. Come un treno è in grado di cambiare binario in base al posizionamento degli scambi, così un programma può intraprendere azioni diverse in relazione ai valori ricevuti in ingresso e ad altre circostanze.

In questo capitolo imparerete a scrivere programmi che prendono decisioni, semplici o complesse, applicando quanto appreso al problema della verifica dei valori introdotti dall'utente.

## 5.1 L'enunciato if

L'enunciato if consente a un programma di compiere azioni diverse in relazione alla natura dei dati che vengono elaborati.

Per prendere una decisione all'interno di un programma si usa l'enunciato if: quando una determinata condizione è verificata, viene eseguito un certo insieme di enunciati, altrimenti ne viene eseguito un altro insieme.

Vediamo un esempio di utilizzo dell'enunciato if. In molti Paesi il numero 13 è considerato sfortunato, per cui, per rispetto nei confronti delle persone superstiziose, spesso i progettisti di edifici fanno in modo che il tredicesimo piano non sia presente: il dodicesimo piano è seguito dal quattordicesimo. Ovviamente il tredicesimo piano non viene lasciato vuoto: banalmente, viene chiamato "quattordicesimo piano". Di conseguenza, i computer che controllano gli ascensori devono tener conto di questa fobia, modificando in modo coerente i numeri di tutti i piani superiori al 13.

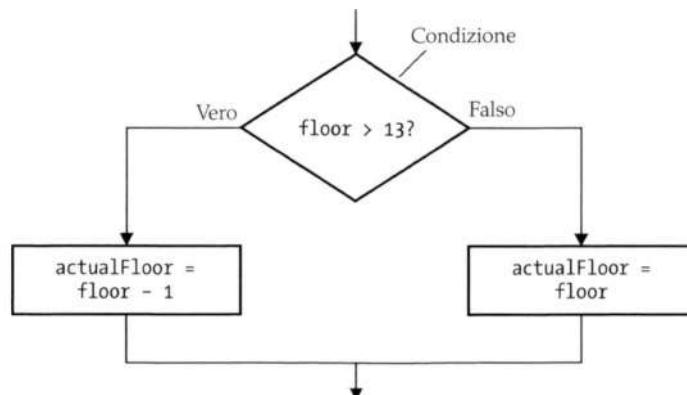
Cerchiamo di simulare questa procedura in Java. Chiederemo all'utente di digitare il numero del piano desiderato e, poi, calcoleremo il numero effettivo corrispondente: quando il dato in ingresso è superiore a 13, per ottenere il numero di piano effettivo dobbiamo decrementare di un'unità il valore acquisito. Se, ad esempio, l'utente digita il numero 20, il programma calcola che il numero di piano effettivo è 19, mentre, se il numero digitato è inferiore a 13, il programma visualizza semplicemente tale numero.

```
int actualFloor = 0;

if (floor > 13)
{
    actualFloor = floor - 1;
}
else
{
    actualFloor = floor;
}
```

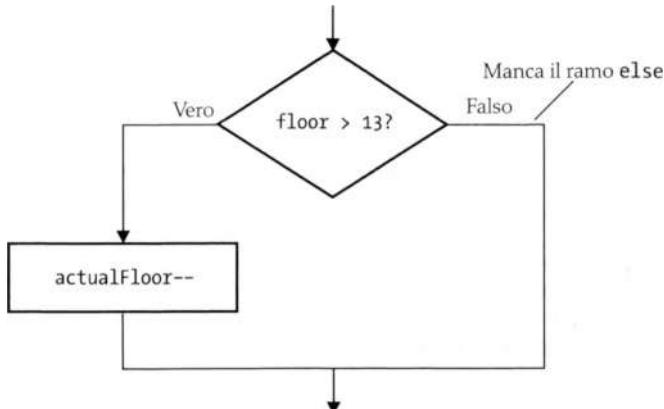
Il diagramma di flusso della Figura 1 illustra il comportamento della diramazione.

**Figura 1**  
Diagramma di flusso per l'enunciato if



**Figura 2**

Diagramma di flusso per l'enunciato if senza il ramo else



Nel nostro esempio entrambi i rami dell'enunciato if contengono un unico enunciato, ma in ciascun ramo si possono inserire anche più enunciati. A volte non c'è alcuna azione da compiere nel ramo else (che significa "altrimenti") e lo si omette completamente, come in questo esempio:

```

int actualFloor = floor;

if (floor > 13)
{
    actualFloor--;
}
// else non serve
  
```

Il diagramma di flusso corrispondente a questo caso è riportato nella Figura 2.

Il programma seguente usa enunciati if: chiede all'utente qual è il numero del piano desiderato e visualizza il corrispondente numero effettivo del piano.

### File ElevatorSimulation.java

```

1 import java.util.Scanner
2
3 /**
4      Simulazione del pannello di controllo di un ascensore per superstiziosi.
5 */
6 public class ElevatorSimulation
7 {
8     public static void main(String[] args)
9     {
10         Scanner in = new Scanner(System.in);
11         System.out.print("Floor: ");
12         int floor = in.nextInt();
13
14         // correge il piano se necessario
15
16         int actualFloor;
17         if (floor > 13)
18         {
  
```

```

19         actualFloor = floor - 1;
20     }
21     else
22     {
23         actualFloor = floor;
24     }
25
26     System.out.println("The elevator will travel to the actual floor "
27         + actualFloor);
28 }
29

```

### Esecuzione del programma:

Floor: 20  
The elevator will travel to the actual floor 19



### Auto-valutazione

- In alcuni Paesi asiatici è il numero 14 a essere considerato sfortunato. In alcuni edifici i proprietari, per stare tranquilli, evitano che siano presenti *entrambi* i piani potenzialmente sfortunati, cioè il tredicesimo e il quattordicesimo. Come si potrebbe modificare il programma visto come esempio in modo che gestisca l'ascensore di un tale edificio?
- Considerate il seguente enunciato if, che calcola un prezzo scontato:

## Sintassi di Java

### 5.1 L'enunciato if

#### Sintassi

```

if (condizione)
{
    enunciati
}
else { enunciati1 }
      { enunciati2 }

```

#### Esempio

Una condizione, che può essere vera o falsa. Spesso, come vedremo, si usano operatori relazionali: == != < <= > >=.

Le parentesi graffe non sono necessarie se il corpo contiene un solo enunciato.

Se non c'è niente da fare in alternativa, non specificate la diramazione else.

```

if (floor > 13)
{
    actualFloor = floor - 1;
}
else
{
    actualFloor = floor;
}

```

Qui non mettete il punto e virgola!

Se la condizione è vera, gli enunciati di questa diramazione vengono eseguiti in sequenza; se la condizione è falsa, vengono ignorati.

È bene incolonnare le parentesi graffe.

Se la condizione è falsa, gli enunciati di questa diramazione vengono eseguiti in sequenza; se la condizione è vera, vengono ignorati.

```
if (originalPrice > 100)
{
    discountedPrice = originalPrice - 20;
}
else
{
    discountedPrice = originalPrice - 10;
}
```

Qual è il prezzo scontato se il prezzo originario è 95? E se è 100? E se è 105?

3. Confrontate questo enunciato if con quello della domanda precedente:

```
if (originalPrice < 100)
{
    discountedPrice = originalPrice - 10;
}
else
{
    discountedPrice = originalPrice - 20;
}
```

I due enunciati calcolano lo stesso prezzo scontato in qualunque caso? In caso di risposta negativa, quando i due valori calcolati differiscono?

4. Considerate i seguenti enunciati, che calcolano un prezzo scontato:

```
discountedPrice = originalPrice;
if (originalPrice > 100)
{
    discountedPrice = originalPrice - 10;
}
```

Qual è il prezzo scontato se il prezzo originario è 95? E se è 100? E se è 105?

5. Le variabili fuelAmount e fuelCapacity contengono, rispettivamente, la quantità attuale di carburante presente nel serbatoio di un veicolo e la capacità del serbatoio stesso. Se nel serbatoio è rimasta una quantità di carburante inferiore al 10 per cento della sua capacità si accende una spia di colore rosso, altrimenti la spia ha il colore verde. Simulate questo processo visualizzando, rispettivamente, la stringa "red" oppure la stringa "green".

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R5.5, R5.6 e E5.10, al termine del capitolo.

## Suggerimenti per la programmazione 5.1

---

### Disposizione delle parentesi graffe

Il compilatore non fa caso a come disponete le parentesi graffe, ma noi raccomandiamo vivamente di seguire questa semplice regola: incolonnate le parentesi graffe aperte e chiuse.

```
if (floor > 13)
{
    floor--;
}
```

Questo schema permette di individuare con facilità la corrispondenza tra le parentesi. Altri programmatori collocano la parentesi graffa aperta nella stessa riga della parola if:

```
if (floor > 13) {
    floor--;
}
```

È una soluzione che fa risparmiare una riga di codice, ma è più difficile individuare le parentesi graffe corrispondenti. Ci sono difensori appassionati di entrambi gli stili.

È importante scegliere uno schema di disposizione delle parentesi e poi attenervisi coerentemente: dipende dai vostri gusti personali oppure dalle regole per lo stile di codifica che dovete seguire.



## Suggerimenti per la programmazione 5.2

### Usare sempre le parentesi graffe

Quando il corpo di un enunciato if è costituito da un solo enunciato, le parentesi graffe non servono. Ad esempio, questo enunciato è perfettamente corretto:

```
if (floor > 13)
    floor--;
```

Ciò nonostante, molti programmatori ritengono che usare sempre le parentesi graffe sia una buona idea:

```
if (floor > 13)
{
    floor--;
}
```

Le parentesi graffe agevolano la lettura del codice e facilitano la manutenzione del codice stesso, perché quando aggiungete enunciati all'interno di un enunciato if non vi dovete preoccupare di inserire le parentesi graffe.



## Errori comuni 5.1

### Un punto e virgola dopo la condizione dell'enunciato if

Questo frammento di codice ha un errore davvero subdolo:

```
if (floor > 13) ; // ERRORE
{
    floor--;
}
```

Dopo la condizione dell'enunciato if non ci dovrebbe essere il punto e virgola, Il compilatore interpreta quell'enunciato in questo modo: se floor è maggiore di 13, esegui l'enunciato rappresentato dal solo punto e virgola, cioè l'enunciato “nullo”, che non fa niente. L'enunciato fra le parentesi graffe che segue il punto e virgola non fa più parte dell'enunciato if e viene, quindi, sempre eseguito. In altre parole, il valore di floor viene sempre decrementato, anche quando floor non è maggiore di 13.



## Suggerimenti per la programmazione 5.3

### Tabulazioni

Quando scrivete programmi Java, per indicare i livelli di annidamento nel codice si usano i “rientri”, cioè gli spostamenti verso destra:

```
public class ElevatorSimulation
{
    public static void main(String[] args)
    {
        int floor;
        ...
        if (floor > 13)
        {
            floor--;
        }
        ...
    }
}
```

0 1 2 3  
Livelli di rientro

Come si sposta il cursore dell’editor dalla colonna più a sinistra, in cui viene posizionato quando si va a capo, alla colonna corrispondente al livello di rientro adeguato? Si può ovviamente premere la barra spaziatrice per il numero di volte sufficiente, ma in molti editor è possibile usare il tasto di tabulazione (Tab), che sposta il cursore al successivo livello di rientro. Alcuni editor hanno anche un’opzione che consente di aggiungere automaticamente i caratteri di tabulazione che servono.

Se il tasto di tabulazione aiuta nella digitazione del codice, alcuni editor usano i caratteri di tabulazione anche per incolonnare bene il testo: una soluzione non particolarmente efficiente, dal momento che non esiste uno standard per l’ampiezza di un carattere di tabulazione e, addirittura, alcuni software li ignorano completamente. Questo diventa un problema soprattutto quando inviate un file che contiene tabulazioni a un’altra persona o a una stampante, per cui conviene salvare i propri file inserendo spazi al posto delle tabulazioni. La maggior parte degli editor può convertire le tabulazioni in spazi in modo automatico: trovate tale opzione nella documentazione del vostro ambiente di sviluppo e attivatela.



## Argomenti avanzati 5.1

### L’operatore condizionale

Java ha un *operatore condizionale*, nella forma:

*condizione ? valore1 : valore2*

Il valore di questa espressione è *valore1* se la *condizione* è vera, *valore2* se la *condizione* è falsa. Nell’esempio dell’ascensore, possiamo calcolare il valore effettivo del piano in questo modo:

```
actualFloor = floor > 13 ? floor - 1 : floor;
```

L'espressione precedente è equivalente a questo frammento di codice:

```
if (floor > 13) { actualFloor = floor - 1; } else { actualFloor = floor; }
```

L'operatore condizionale può essere usato in qualsiasi punto del codice in cui è prevista la presenza di un valore, ad esempio:

```
System.out.println("Actual floor: " + (floor > 13 ? floor - 1 : floor));
```

In questo libro non useremo l'operatore condizionale, sebbene si tratti di un costrutto utile e ammesso, che troverete in molti programmi Java.



## Suggerimenti per la programmazione 5.4

### Evitare duplicazioni nelle diramazioni

Controllate sempre se le due diramazioni di un enunciato if contengono, come nell'esempio seguente, *codice duplicato*. In tal caso, spostatelo al di fuori dell'enunciato if.

```
if (floor > 13)
{
    actualFloor = floor - 1;
    System.out.println("Actual floor: " + actualFloor);
}
else
{
    actualFloor = floor;
    System.out.println("Actual floor: " + actualFloor);
}
```

L'enunciato di visualizzazione è esattamente il medesimo in entrambe le diramazioni. Non si tratta di un errore: il programma viene eseguito correttamente. Tuttavia, si può semplificare il codice spostando l'enunciato duplicato, in questo modo:

```
if (floor > 13)
{
    actualFloor = floor - 1;
}
else
{
    actualFloor = floor;
}
System.out.println("Actual floor: " + actualFloor);
```

L'eliminazione delle duplicazioni è particolarmente importante quando i programmi devono essere manutenuti per lungo tempo: quando due insiemi di enunciati devono produrre il medesimo effetto, può facilmente accadere che un programmatore modifichi un insieme senza modificare l'altro.

## 5.2 Confrontare valori

Ogni enunciato if contiene una condizione che, in molti casi, richiede un confronto tra due valori. Nei paragrafi che seguono vedrete come realizzare confronti in Java.

### 5.2.1 Operatori relazionali

Per confrontare numeri usiamo gli operatori relazionali:  
`<, <=, >, >=, ==, !=.`

Un **operatore relazionale** (*relational operator*) verifica la relazione esistente tra due valori, come l'operatore `>` che abbiamo già usato: `floor > 13`. In Java esistono sei operatori relazionali, elencati nella Tabella 1.

Come potete vedere, soltanto due operatori relazionali (`<` e `>`) appaiono nella consueta notazione matematica, esattamente come li avreste potuti immaginare. Le tastiere dei computer non hanno i simboli  $\geq$ ,  $\leq$  o  $\neq$ , tuttavia gli operatori `>=`, `<=` e `!=` si ricordano facilmente, perché sono molto simili ai corrispondenti simboli matematici, mentre l'operatore `==` è spesso fonte di confusione per chi si avvicina al linguaggio Java.

**Tabella 1**  
Operatori relazionali

| Operatore Java     | Notazione matematica           | Descrizione       |
|--------------------|--------------------------------|-------------------|
| <code>&gt;</code>  | <code>&gt;</code>              | Maggiore          |
| <code>&gt;=</code> | <code><math>\geq</math></code> | Maggiore o uguale |
| <code>&lt;</code>  | <code>&lt;</code>              | Minore            |
| <code>&lt;=</code> | <code><math>\leq</math></code> | Minore o uguale   |
| <code>==</code>    | <code>=</code>                 | Uguale            |
| <code>!=</code>    | <code><math>\neq</math></code> | Diverso           |

Gli operatori relazionali confrontano valori; in particolare, l'operatore `==` verifica un'uguaglianza.

In Java, l'operatore `=` ha già un significato: effettua l'assegnazione. L'operatore `==` rappresenta, invece, la verifica di uguaglianza.

```
floor = 13; // assegna 13 a floor
```

```
if (floor == 13) ... // verifica se floor è uguale a 13
```

Quindi, dovete ricordarvi di usare `==` all'interno di una verifica e `=` per le assegnazioni.

Gli operatori relazionali presentati nella Tabella 1 hanno una priorità inferiore rispetto agli operatori aritmetici: ciò significa che si possono scrivere espressioni aritmetiche a sinistra e a destra di un operatore relazionale senza dover usare parentesi. Ad esempio, nell'espressione

```
floor - 1 < 13
```

vengono valutati i due operandi ai lati dell'operatore `<`, cioè `floor - 1` e `13`, per poi confrontare i risultati ottenuti. L'Appendice B elenca gli operatori del linguaggio Java e la loro priorità.

## 5.2.2 Confrontare numeri in virgola mobile

Quando si confrontano numeri in virgola mobile bisogna fare attenzione e considerare l'eventualità che avvengano errori di arrotondamento. Per esempio, il codice seguente moltiplica per se stessa la radice quadrata di 2 e sottrae 2 al risultato:

```
double r = Math.sqrt(2);
double d = r * r - 2;
if (d == 0)
    System.out.println("sqrt(2) squared minus 2 is 0");
else
    System.out.println("sqrt(2) squared minus 2 is not 0 but " + d);
```

Sebbene le leggi della matematica dicano che il risultato è uguale a 0, questo frammento di programma scriverà:

```
sqrt(2) squared minus 2 is not 0 but 4.440892098500626E-16
```

Purtroppo questi errori di arrotondamento sono inevitabili. Il più delle volte non ha senso confrontare esattamente numeri in virgola mobile: dobbiamo, invece, verificare se questi numeri siano *sufficientemente prossimi*.

Per verificare se un numero  $x$  è prossimo a zero, potete verificare se il suo valore assoluto  $|x|$ , cioè il numero privato del segno, sia minore di un piccolo valore di soglia, che viene solitamente chiamato  $\epsilon$  (la lettera greca epsilon). Per confrontare numeri `double` di solito si usa un valore di  $\epsilon$  uguale a  $10^{-14}$ .

Analogamente, potete verificare se due numeri sono prossimi l'uno all'altro controllando se la loro differenza è prossima a 0.

$$|x - y| \leq \epsilon$$

In Java, possiamo scrivere la verifica in questo modo

```
final double EPSILON = 1E-14;
if (Math.abs(x - y) <= EPSILON)
{
    // x è quasi uguale a y
}
```

## 5.2.3 Confrontare stringhe

Per verificare se due stringhe sono uguali dovete usare il metodo `equals`:

```
if (string1.equals(string2)) . . .
```

Per confrontare stringhe non usate l'operatore `==`, ma il metodo `equals`.

Non usate l'operatore `==` per confrontare stringhe. L'espressione seguente ha un significato che non è correlato all'uguaglianza tra stringhe:

```
if (string1 == string2) // inutile
```

perché, in realtà, questo enunciato verifica se le due stringhe si trovano nella stessa posizione all'interno della memoria. Dal momento che possono esistere stringhe con

contenuto identico conservate in posizioni diverse, questa verifica non ha mai senso nella programmazione reale: consultate la sezione Errori comuni 5.2.

Se due stringhe non sono identiche, potrete comunque voler conoscere la loro relazione reciproca: il metodo `compareTo` confronta le stringhe secondo il criterio di **ordinamento lessicografico**, che è molto simile all'ordinamento alfabetico con cui sono disposte le parole in un dizionario. Se questa condizione è vera

```
string1.compareTo(string2) < 0
```

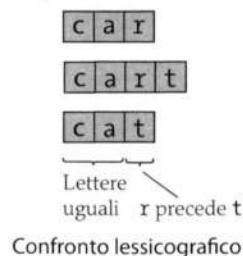
significa che `string1` precede la stringa `string2` nel dizionario. Ciò succede, ad esempio, se `string1` contiene "Harry" e `string2` contiene "Hello". Se, invece, si verifica che

```
string1.compareTo(string2) > 0
```

allora `string1` segue la stringa `string2` nel dizionario. Infine, `string1` e `string2` sono uguali se

```
string1.compareTo(string2) == 0
```

Il metodo `compareTo` confronta  
due stringhe secondo il criterio  
di ordinamento lessicografico.



In realtà, l'ordinamento lessicografico utilizzato da Java è leggermente diverso da quello di un normale dizionario. In Java:

- tutte le lettere maiuscole precedono tutte le lettere minuscole (ad esempio, "Z" precede "a");
- il carattere "spazio" precede tutti i caratteri visualizzabili;
- i numeri precedono le lettere;
- per l'ordinamento dei segni di punteggiatura, si veda l'Appendice A.

Per effettuare il confronto tra due stringhe si parte confrontando la prima lettera di ciascuna stringa, poi si procede confrontando le seconde lettere, e così via, finché una delle stringhe termina oppure si trova una coppia di lettere tra loro diverse.

Se una delle stringhe termina, la stringa più lunga è considerata la "maggiori" delle due. Ad esempio, confrontando "car" con "cart", si osserva che le prime tre lettere coincidono e, così, si raggiunge la fine della prima stringa, per cui, secondo l'ordinamento lessicografico, "car" precede "cart".

Quando, invece, si trova una coppia di lettere diverse in posizioni corrispondenti, la stringa contenente il carattere "maggiori" è considerata la "maggiori" delle due stringhe. Confrontiamo, ad esempio, "cat" con "cart": le prime due lettere della prima stringa coincidono con la lettera corrispondente della seconda stringa, ma nella terza posizione la lettera t è "maggiori" della lettera r, per cui, secondo l'ordinamento lessicografico, "cat" segue "cart".

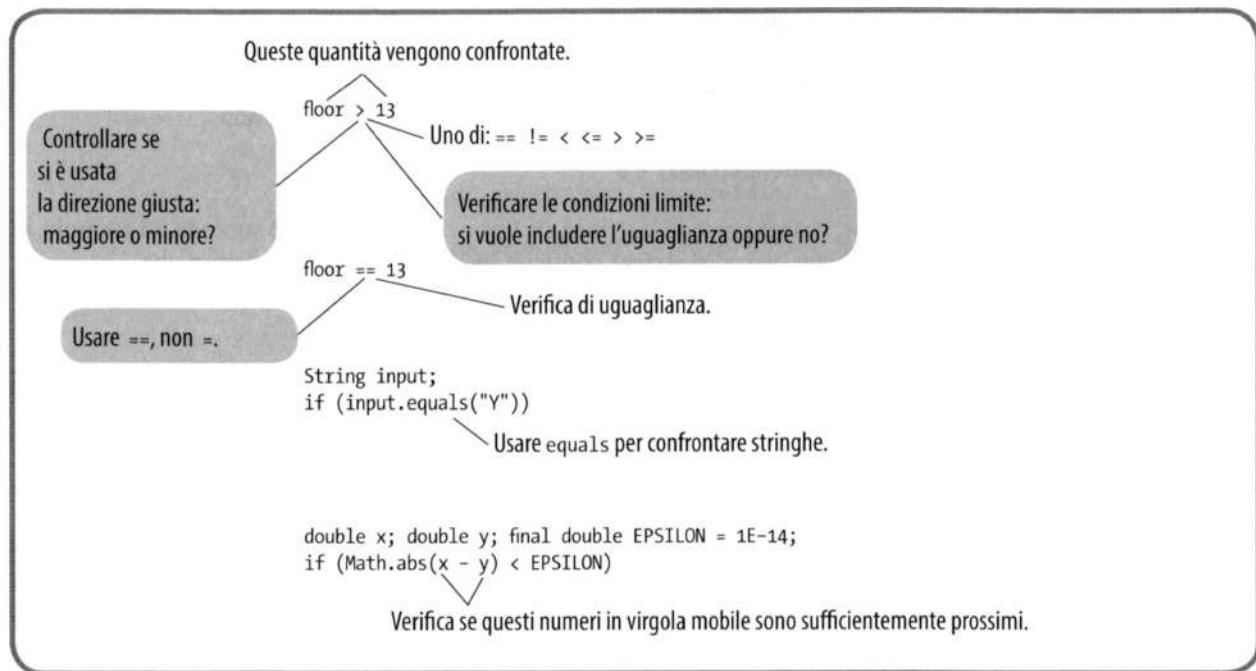
## 5.2.4 Confrontare oggetti

Confrontando due riferimenti a oggetti mediante l'operatore `==`, si controlla se i riferimenti puntano allo stesso oggetto. Analizziamo questo esempio:

```
Rectangle box1 = new Rectangle(5, 10, 20, 30);
Rectangle box2 = box1;
Rectangle box3 = new Rectangle(5, 10, 20, 30);
```

## Sintassi di Java

### 5.2 Confronti



Questo confronto risulta vero:

```
box1 == box2
```

Infatti, le due variabili si riferiscono allo stesso oggetto. Per contro, questo è falso:

```
box1 == box3
```

perché in questo caso le due variabili si riferiscono a oggetti *distinti* (osservate la Figura 3) e non importa che abbiano contenuti identici.

Per verificare se due rettangoli hanno lo stesso *contenuto* (cioè se hanno la stessa altezza, la stessa larghezza e il vertice superiore sinistro nella stessa posizione) potete usare il metodo `equals`. Per esempio, questa condizione è vera:

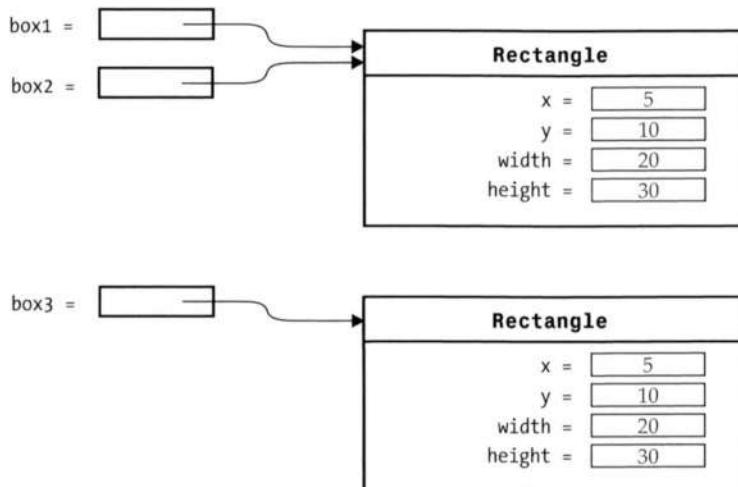
```
box1.equals(box3)
```

Il metodo `equals` va, però, usato con attenzione: funziona correttamente soltanto se chi ha realizzato la classe l'ha definito. La classe `Rectangle` ha un metodo `equals` che è adatto al confronto di rettangoli.

Nelle vostre classi dovete progettare un metodo `equals` appropriato: imparerete come farlo nel Capitolo 9. Fino a quel momento, non potete usare il metodo `equals` per confrontare oggetti delle vostre classi.

L'operatore == verifica se due riferimenti puntano allo stesso oggetto. Per confrontare, invece, i contenuti di oggetti, si deve usare il metodo equals.

**Figura 3**  
Confronto tra riferimenti



### 5.2.5 Confrontare con null

Il riferimento `null` non fa riferimento ad alcun oggetto.

Un riferimento può avere il valore speciale `null` se non si riferisce ad alcun oggetto, condizione spesso utilizzata per indicare che a una variabile non è ancora stato assegnato un valore valido, come in questo esempio:

```

String middleInitial = null; // valore non ancora assegnato
if (. . .)
{
    middleInitial = middleName.substring(0, 1);
}

```

Per verificare se un riferimento ha il valore `null` si usa l'operatore `==` (e non `equals`):

```

if (middleInitial == null)
{
    System.out.println(firstName + " " + lastName);
}
else
{
    System.out.println(firstName + " " + middleInitial + ". " + lastName);
}

```

Si noti che un riferimento `null` è diverso dalla stringa vuota, `""`. La stringa vuota è una stringa valida di lunghezza 0, mentre il valore `null` indica che una variabile di tipo stringa non si riferisce ad alcuna stringa.

La Tabella 2 riassume le possibilità di confronto tra valori in Java.



### Auto-valutazione

6. Quali delle condizioni seguenti sono vere, se `a` vale 3 e `b` vale 4?

- a.
- b.
- c.

$a + 1 \leq b$   
 $a + 1 \geq b$   
 $a + 1 \neq b$

**Tabella 2** Esempi con operatori relazionali

| Espressione                                                                                       | Valore | Commento                                                                                                                                                                             |
|---------------------------------------------------------------------------------------------------|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $3 \leq 4$                                                                                        | true   | 3 è minore di 4; $\leq$ fa la verifica “minore o uguale”.                                                                                                                            |
|  $3 \leqslant 4$ | Errore | L’operatore “minore o uguale” è $\leq$ , non $\leqslant$ .                                                                                                                           |
| $3 > 4$                                                                                           | false  | $>$ è l’opposto di $\leq$ .                                                                                                                                                          |
| $4 < 4$                                                                                           | false  | L’operando sinistro deve essere strettamente minore dell’operando destro.                                                                                                            |
| $4 \leq 4$                                                                                        | true   | I due operandi sono uguali; $\leq$ fa la verifica “minore o uguale”.                                                                                                                 |
| $3 == 5 - 2$                                                                                      | true   | $==$ fa la verifica di uguaglianza.                                                                                                                                                  |
| $3 != 5 - 1$                                                                                      | true   | $!=$ fa la verifica di diversità ed è vero che 3 è diverso da $5 - 1$ .                                                                                                              |
|  $3 = 6 / 2$     | Errore | Per la verifica di uguaglianza si usa $==$ .                                                                                                                                         |
| $1.0 / 3.0 == 0.3333333333$                                                                       | false  | Anche se i valori sono assai prossimi tra loro, non sono esattamente uguali.                                                                                                         |
|  "10" > 5        | Errore | Non si può confrontare una stringa con un numero.                                                                                                                                    |
| "Tomato".substring(0, 3).equals("Tom")                                                            | true   | Per verificare se due stringhe hanno il medesimo contenuto usate sempre il metodo <code>equals</code> .                                                                              |
| "Tomato".substring(0, 3) == ("Tom")                                                               | false  | Non usare mai <code>==</code> per confrontare stringhe, perché tale operatore verifica soltanto se le due stringhe si trovano nella stessa posizione in memoria (Errori comuni 5.2). |

7. Descrivete la condizione opposta a questa:

`floor > 13`

8. Cosa c’è di sbagliato in questo enunciato?

```
if (scoreA = scoreB)
{
    System.out.println("Tie");
}
```

9. In questo enunciato if, esprimete una condizione che verifichi se l’utente ha scritto la lettera Y:

```
System.out.println("Enter Y to quit.");
String input = in.next();
if (...)
{
    System.out.println("Goodbye.");
}
```

10. Scrivete due modi diversi per verificare se la stringa str è la stringa vuota.

11. Qual è il valore di `s.length()` se s è:

- a. la stringa vuota ""?
- b. la stringa " ", contenente soltanto uno spazio?
- c. null?

12. Date le seguenti inizializzazioni, quali dei confronti che seguono contengono errori di sintassi? Quali sono sintatticamente corretti, ma di dubbia utilità dal punto di vista logico?

```
String a = "1";
String b = "one";
double x = 1;
double y = 3 * (1.0 / 3);
```

- a. `a == "1"`
- b. `a == null`
- c. `a.equals("")`
- d. `a == b`
- e. `a == x`
- f. `x == y`
- g. `x - y == null`
- h. `x.equals(y)`

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R5.4, R5.7 e E5.14, al termine del capitolo.



## Errori comuni 5.2

### Utilizzare == per confrontare stringhe

Se scrivete

```
if (nickname == "Rob")
```

la condizione sarà vera solo se la variabile `nickname` si riferisce esattamente alla stessa posizione in memoria occupata dalla stringa letterale "Rob".

Se la variabile stringa era stata inizializzata con la stessa stringa letterale, allora la verifica avrà esito positivo:

```
String nickname = "Rob";
...
if (nickname == "Rob") // la condizione è vera
```

Tuttavia, se la stringa contiene le lettere R o b ed è stata costruita in qualche altro modo, allora la stessa verifica avrà esito negativo:

```
String name = "Robert";
String nickname = name.substring(0, 3);
...
if (nickname == "Rob") // la condizione è falsa
```

In questo secondo caso, il metodo `substring` genera una stringa che, in memoria, si trova in una posizione diversa: anche se le due stringhe hanno lo stesso contenuto, la verifica di uguaglianza fallisce.

Dovete ricordarvi di non usare mai l'operatore `==` per confrontare stringhe: utilizzate sempre il metodo `equals`.



## Consigli pratici 5.1

### Progettare un enunciato if

Questi Consigli pratici vi guideranno nel processo di progettazione di un enunciato `if`, usando, come sempre, un problema concreto.

**Problema.** Una libreria universitaria offre sconti particolari il 24 ottobre di ogni anno: il Giorno del Kilobyte. Lo sconto è pari all'otto per cento su tutti gli accessori per computer che costano meno di \$128, salendo al 16 per cento quando il prezzo è superiore. Scrivete un programma che chiede al cassiere il prezzo originario e visualizza il prezzo scontato.

**Fase 1** Decidere quale sia la condizione che controlla la diramazione

In questo problema la scelta più ovvia per la condizione è:

$$\text{prezzo originale} < 128 ?$$

È proprio la condizione giusta e la useremo nella soluzione che proporremo, ma potreste giungere a una soluzione altrettanto corretta usando la condizione opposta. Il prezzo originale è almeno uguale a \$128? Probabilmente scegliereste questa seconda condizione se immaginatevi di trovarvi nei panni di un cliente che vuole sapere quando si applica lo sconto maggiore.

**Fase 2** Scrivere lo pseudocodice per l'azione da compiere quando la condizione è vera

In questa fase dovete elencare le azioni da svolgere nel ramo “affermativo”. I dettagli, naturalmente, dipendono dalla natura del problema: può darsi che vogliate visualizzare un messaggio, oppure calcolare alcuni valori, oppure, ancora, terminare il programma.

In questo caso dobbiamo applicare uno sconto dell'8%:

$$\text{prezzo scontato} = 0.92 \cdot \text{prezzo originale}$$

**Fase 3** Scrivere lo pseudocodice per l'azione da compiere quando la condizione è *falsa*

Cosa volete fare quando la condizione delineata nella Fase 1 non è verificata? A volte non volete fare proprio nulla: in tal caso, usate un enunciato *if* privo della clausola *else*.

Nel nostro esempio la condizione controlla se il prezzo originale è inferiore a \$128. Se tale condizione *non* è vera, allora il prezzo è almeno uguale a \$128, per cui va applicato lo sconto maggiore, pari al 16%:

$$\text{prezzo scontato} = 0.84 \cdot \text{prezzo originale}$$

**Fase 4** Verificare con cura gli operatori relazionali

Per prima cosa bisogna controllare che le verifiche avvengano nella *direzione* corretta: è molto frequente scambiare, per errore, gli operatori *<* e *>*. Successivamente occorre decidere se usare, ad esempio, l'operatore *<* oppure il suo “parente stretto”, l'operatore *<=*.

Cosa ci si aspetta che accada quando il prezzo originale è esattamente uguale a \$128? Leggendo con attenzione la descrizione del problema, scopriamo che lo sconto minore si applica se il prezzo originale è *minore di* \$128, mentre lo sconto maggiore viene applicato quando il prezzo è *almeno uguale a* \$128. Quindi, un prezzo di \$128 *non* soddisfa la condizione che abbiamo espresso per applicare lo sconto minore, per cui bisogna usare l'operatore *<*, non *<=*.

**Fase 5** Eliminare le duplicazioni

Bisogna individuare eventuali azioni comuni alle due diramazioni e spostarle all'esterno dell'enunciato `if`, come visto nella sezione Suggerimenti per la programmazione 5.4.

Nel nostro esempio troviamo due enunciati che hanno la stessa forma:

$$\text{prezzo scontato} = \underline{\quad} \cdot \text{prezzo originale}$$

L'unica differenza è la percentuale di sconto. È preferibile assegnare un valore a tale percentuale all'interno delle due diramazioni, per calcolare il prezzo scontato in un secondo momento:

```
if prezzo originale < 128
    percentuale di sconto = 0.92
else
    percentuale di sconto = 0.84
prezzo scontato = percentuale di sconto · prezzo originale
```

**Fase 6** Collaudare entrambe le diramazioni

Individuate due casi da collaudare, uno che soddisfi la condizione che controlla l'esecuzione della prima diramazione dell'enunciato `if` e l'altro che non la soddisfi, chiedendovi cosa debba succedere nei due casi, poi seguite passo dopo passo lo pseudocodice che avete scritto ed eseguite le azioni indicate.

Nel nostro esempio, consideriamo i casi in cui il prezzo originale è uguale a \$100 e a \$200. Ci aspettiamo, naturalmente, che il primo prezzo scontato sia \$92 e che il secondo sia \$168.

Quando il prezzo originale è uguale a \$100, la condizione  $100 < 128$  è vera, per cui vengono eseguite le azioni seguenti:

$$\begin{aligned}\text{percentuale di sconto} &= 0.92 \\ \text{prezzo scontato} &= 0.92 \cdot 100 = 92\end{aligned}$$

Quando il prezzo originale è uguale a \$200, la condizione  $200 < 128$  è falsa, per cui vengono eseguite le azioni seguenti:

$$\begin{aligned}\text{percentuale di sconto} &= 0.84 \\ \text{prezzo scontato} &= 0.84 \cdot 200 = 168\end{aligned}$$

In entrambi i casi si ottiene la risposta prevista.

**Fase 7** Scrivere l'enunciato `if` in Java

Scrivete lo schema iniziale dell'enunciato `if`:

```
if ()
{
}
else
```

{

}

e riempitelo, come indicato nella sezione Sintassi di Java 5.1, cancellando la clausola `else` se non viene utilizzata.

Nel nostro esempio, l'enunciato completo diventa:

```
if (originalPrice < 128)
{
    discountRate = 0.92;
}
else
{
    discountRate = 0.84;
}
discountedPrice = discountRate * originalPrice;
```



## Esempi completi 5.1

### Estrazione della stringa centrale

**Problema.** Abbiamo come obiettivo la creazione di una stringa contenente il carattere centrale estratto da una stringa assegnata, `str`. Ad esempio, se la stringa è "crate", il risultato sarà "a". Nel caso in cui la stringa abbia un numero pari di caratteri, estraiamo i due caratteri centrali: se la stringa è "crates", il risultato sarà "at".

**Fase 1** Decidere quale sia la condizione che controlla la diramazione

Dobbiamo compiere azioni diverse per stringhe di lunghezza dispari e pari, per cui la condizione è:

La lunghezza della stringa è dispari?

In Java, per verificare se un valore è pari o dispari, si valuta il resto della sua divisione intera per due. Quindi, la condizione diventa:

`str.length() % 2 == 1?`

**Fase 2** Scrivere lo pseudocodice per l'azione da compiere quando la condizione è vera

Dobbiamo trovare la posizione del carattere centrale. Se la stringa ha lunghezza 5, la posizione è 2.

|   |   |   |   |   |
|---|---|---|---|---|
| c | r | a | t | e |
| 0 | 1 | 2 | 3 | 4 |

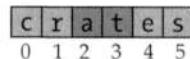
In generale:

```
posizione = str.length() / 2 (ignorando il resto)
risultato = str.substring(posizione, posizione + 1)
```

**Fase 3** Scrivere lo pseudocodice per l'azione da compiere quando la condizione è *falsa*

In questo caso dobbiamo trovare le posizioni dei caratteri centrali, che sono due. Se la stringa ha lunghezza 6, la posizione iniziale è 2 e quella finale è 3. Ricordando che il secondo parametro del metodo `substring` è la posizione del primo carattere che non vogliamo estrarre, costruiamo questa invocazione:

```
risultato = str.substring(2, 4);
```



In generale:

```
posizione = str.length() / 2 - 1
risultato = str.substring(posizione, posizione + 2)
```

**Fase 4** Verificare con cura gli operatori relazionali

Vogliamo veramente verificare la condizione `str.length() % 2 == 1`? Quando, ad esempio, la lunghezza è 5, il resto di 5 diviso 2 è 1. In generale, dividendo un numero dispari per 2 si ottiene resto 1 (in realtà, dividendo per 2 un numero dispari negativo si ottiene resto -1, ma la lunghezza di una stringa non è mai negativa). La nostra condizione è, quindi, corretta.

**Fase 5** Eliminare le duplicazioni

Siamo giunti a questo enunciato:

```
if str.length() % 2 == 1
    posizione = str.length() / 2 (ignorando il resto)
    risultato = str.substring(posizione, posizione + 1)
else
    posizione = str.length() / 2 - 1
    risultato = str.substring(posizione, posizione + 2)
```

Il secondo enunciato presente nelle due diramazioni è pressoché lo stesso: soltanto la lunghezza della sottostringa è diversa. Impostiamo, quindi, tale lunghezza in ciascuna diramazione:

```
if str.length() % 2 == 1
    posizione = str.length() / 2 (ignorando il resto)
    lunghezza = 1
else
    posizione = str.length() / 2 - 1
    lunghezza = 2
risultato = str.substring(posizione, posizione + lunghezza)
```

### Fase 6 Collaudare entrambe le diramazioni

Per il collaudo usiamo un insieme di stringhe diverso da quello usato durante la progettazione. Come stringa di lunghezza dispari consideriamo "monitor". Seguendo lo pseudocodice, otteniamo:

```
posizione = str.length() / 2 = 7 / 2 = 3 (ignorando il resto)
lunghezza = 1
risultato = str.substring(3, 4) = "i"
```

Con la stringa di lunghezza pari "monitors", otteniamo:

```
posizione = str.length() / 2 - 1 = 8 / 2 - 1 = 3
lunghezza = 2
risultato = str.substring(3, 5) = "it"
```

### Fase 7 Scrivere l'enunciato if in Java

Ecco il frammento di codice completo.

```
if (str.length() % 2 == 1)
{
    position = str.length() / 2;
    length = 1;
}
else
{
    position = str.length() / 2 - 1;
    length = 2;
}
String result = str.substring(position, position + length);
```

Nel pacchetto dei file scaricabili per questo libro, la cartella `worked_example_1` del Capitolo 5 contiene il codice sorgente completo della classe.

## 5.3 Alternative multiple

Per prendere decisioni articolate si possono combinare più enunciati if.

Nel Paragrafo 5.1 avete visto come realizzare una diramazione a due vie mediante un enunciato `if`, ma molte situazioni richiedono più di una singola decisione: in questo paragrafo vedrete come prendere una decisione che presenta più di due alternative.

Consideriamo come esempio un programma che visualizzi una descrizione degli effetti di un terremoto, secondo la scala Richter, che è un sistema di misurazione dell'energia di un terremoto. Ogni passaggio da un valore della scala al successivo, per esempio quello fra 6.0 e 7.0, indica una moltiplicazione per dieci dell'energia del sisma.

In questo caso abbiamo cinque diramazioni: una per ciascuna delle quattro descrizioni dei dati e una per l'assenza di dati. La Figura 4 mostra il diagramma di flusso corrispondente a questo enunciato con diramazioni multiple.



## Computer e società 5.1

### Il sistema di gestione dei bagagli di Denver

Prendere decisioni è un'attività essenziale in qualsiasi programma per calcolatore e questo è davvero evidente in un sistema automatico che coordina la gestione dei bagagli in un aeroporto. Dopo aver letto i codici identificativi dei bagagli, il sistema li ordina e li in-strada verso diversi nastri trasportatori, infine operatori umani li sistemanano su veicoli a motore. Quando nella città di Denver venne costruito un grande aeroporto per sostituire il precedente, obsoleto e congestionato, la società a cui venne affidata la realizzazione del sistema di gestione dei bagagli si spinse ancora oltre e progettò un sistema innovativo, sostituendo gli operatori umani con carrelli robotizzati. Il sistema semplicemente non funzionò: era viziato da problemi meccanici, come la caduta di bagagli sui binari, con conseguente malfunzionamento dei carrelli, e altrettanto frustranti si

rivelarono gli errori nel software di controllo. I carrelli tendevano ad accumularsi in alcuni punti, mentre erano necessari altrove.

L'apertura dell'aeroporto era prevista per il 1993, ma, in mancanza di un sistema di gestione dei bagagli funzionante, venne ritardata per oltre un anno, mentre i responsabili cercavano di rimediare ai problemi, senza, però, avere successo: alla fine venne reso operativo un sistema manuale. Il costo dei ritardi, per la città e per le linee aeree, ammontò quasi a un miliardo di dollari e la società in questione, una delle principali nel settore delle vendite di sistemi di gestione dei bagagli negli Stati Uniti, andò in bancarotta.

È evidente come sia molto rischioso costruire un sistema che agisce su grande scala basandosi su una tecnologia che non è mai stata collaudata su piccola scala. Nel 2013, la presentazione dell'assistenza sanitaria uni-

versale negli Stati Uniti venne messa a rischio dal mancato funzionamento del sito web che doveva consentire la scelta di un piano assicurativo. Il sistema prometteva di poter condurre un'esperienza di acquisto di un piano assicurativo simile alla prenotazione di un volo aereo, ma il sito HealthCare.gov non fu nemmeno in grado di proporre l'elenco dei piani assicurativi disponibili. Avrebbe dovuto anche verificare il livello di reddito di ciascun richiedente, usando tale informazione per proporre il livello di sussidio più adeguato: un compito che si rivelò decisamente più complesso della semplice verifica del credito disponibile al momento di pagare un biglietto aereo. L'amministrazione Obama avrebbe dovuto essere adeguatamente consigliata, in modo da progettare una procedura di iscrizione che non fosse basata su programmi non collaudati.

**Tabella 3**  
Scala Richter

|               | Valore | Effetto                                                         |
|---------------|--------|-----------------------------------------------------------------|
| Scala Richter | 8      | La maggior parte delle strutture cade                           |
|               | 7      | Molti edifici distrutti                                         |
|               | 6      | Molti edifici significativamente danneggiati, alcuni collassano |
|               | 4,5    | Danni a edifici costruiti in modo non adeguato                  |

Per realizzare alternative multiple si possono usare più enunciati if, in questo modo:

```
if (richter >= 8.0)
{
    description = "Most structures fall";
}
else if (richter >= 7.0)
{
    description = "Many buildings destroyed";
}
else if (richter >= 6.0)
{
    description = "Many buildings considerably damaged, some collapse";
}
else if (richter >= 4.5)
```

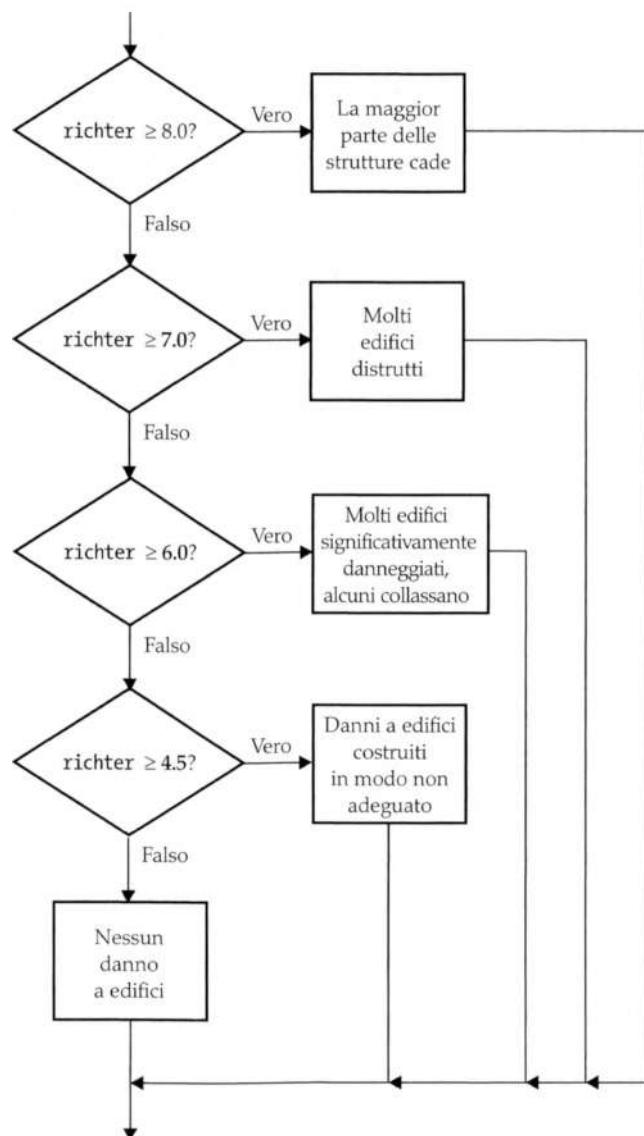
```

{
    description = "Damage to poorly constructed buildings";
}
else
{
    description = "No destruction of buildings";
}

```

Non appena una delle quattro verifiche ha successo, viene visualizzato il corrispondente effetto del terremoto e non viene controllata nessuna ulteriore condizione. Se nessuna delle condizioni è verificata, entra in gioco la clausola else conclusiva e viene visualizzato il messaggio previsto.

**Figura 4**  
Alternative multiple



Con questa struttura di programmazione è necessario ordinare in modo corretto le condizioni e controllare prima il valore maggiore. Supponiamo, infatti, di invertire l'ordine delle condizioni:

```
if (richter >= 4.5) // verifiche in ordine sbagliato
{
    description = "Damage to poorly constructed buildings";
}
else if (richter >= 6.0)
{
    description = "Many buildings considerably damaged, some collapse";
}
else if (richter >= 7.0)
{
    description = "Many buildings destroyed";
}
else if (richter >= 8.0)
{
    description = "Most structures fall";
}
```

Questo codice non funziona. Supponiamo, infatti, che il valore di `richter` sia 7.1: dato che non è minore di 4.5, la prima verifica ha successo e le altre condizioni non vengono nemmeno controllate.

La soluzione a questo problema consiste nel verificare dapprima le condizioni più specifiche. In questo caso, la condizione `richter >= 8.0` è più specifica della condizione `richter >= 7.0` e la condizione `richter >= 4.5` è più generica di entrambe le prime due (cioè è soddisfatta da un maggior numero di valori).

In questo esempio è anche importante che sia stata usata la sequenza `if/else if/else` e non un semplice elenco di enunciati `if` indipendenti. Analizziamo, infatti, questa sequenza di verifiche tra loro indipendenti:

```
if (richter >= 8.0) // non usa le clausole else
{
    description = "Most structures fall";
}
if (richter >= 7.0)
{
    description = "Many buildings destroyed";
}
if (richter >= 6.0)
{
    description = "Many buildings considerably damaged, some collapse";
}
if (richter >= 4.5)
{
    description = "Damage to poorly constructed buildings";
}
```

Ora le alternative non sono più mutuamente esclusive. Se il valore di `richter` è 7.1, le ultime *tre* condizioni sono verificate: alla variabile `description` vengono assegnate tre diverse stringhe, terminando con una sbagliata, che sarà quella definitiva.

**Usando enunciati `if` multipli, occorre verificare prima le condizioni più specifiche, poi quelle più generiche.**



## Auto-valutazione

13. Nel programma di un gioco al calcolatore, i punteggi dei giocatori A e B vengono memorizzati, rispettivamente, nelle variabili `scoreA` e `scoreB`. Ipotizzando che il giocatore vincente sia quello con il punteggio maggiore, scrivete una sequenza di `if/else if/else` che visualizzi "A won" (cioè "ha vinto A"), "B won" ("ha vinto B") oppure "Game tied" ("Pareggio").
14. Scrivete un enunciato condizionale con tre diramazioni che assegna a `s` il valore 1 se `x` è positivo, -1 se `x` è negativo o 0 se `x` è uguale a zero.
15. Come si può ottenere lo stesso risultato richiesto dalla domanda precedente usando due sole diramazioni?
16. A volte i principianti scrivono enunciati come questo:

```
if (price > 100)
{
    discountedPrice = price - 20;
}
else if (price <= 100)
{
    discountedPrice = price - 10;
}
```

Spiegate come si può migliorare questo codice.

17. Cosa viene visualizzato dal programma che descrive i terremoti se l'utente digita il valore -1?
18. Supponiamo di voler fare in modo che il programma che descrive i terremoti verifichi che l'utente abbia digitato un valore positivo. Quale ramo aggiungereste all'enunciato `if`, e in quale punto?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R5.23, E5.11 e E5.24, al termine del capitolo.



## Argomenti avanzati 5.2

### L'enunciato switch

Per confrontare *un singolo valore* rispetto a diversi valori alternativi, invece di una serie di enunciati `if/else if/else`, si può utilizzare un enunciato `switch`. Ecco un esempio:

```
int digit = . . .;
switch (digit)
{
    case 1: digitName = "one"; break;
    case 2: digitName = "two"; break;
    case 3: digitName = "three"; break;
    case 4: digitName = "four"; break;
    case 5: digitName = "five"; break;
    case 6: digitName = "six"; break;
    case 7: digitName = "seven"; break;
```

```

    case 8: digitName = "eight"; break;
    case 9: digitName = "nine"; break;
    default: digitName = ""; break;
}

```

Si tratta di una forma abbreviata per il codice seguente:

```

int digit = . . . ;
if (digit == 1) { digitName = "one"; }
else if (digit == 2) { digitName = "two"; }
else if (digit == 3) { digitName = "three"; }
else if (digit == 4) { digitName = "four"; }
else if (digit == 5) { digitName = "five"; }
else if (digit == 6) { digitName = "six"; }
else if (digit == 7) { digitName = "seven"; }
else if (digit == 8) { digitName = "eight"; }
else if (digit == 9) { digitName = "nine"; }
else { digitName = ""; }

```

È soltanto una scorciatoia, ma ha un vantaggio: appare ben evidente che tutte le diramazioni confrontano *lo stesso valore*, in questo caso `digit`.

L'impiego dell'enunciato `switch` è abbastanza limitato: i valori presenti nelle verifiche devono essere costanti e possono essere numeri interi o caratteri (o anche stringhe, a partire da Java 7). Non si può usare `switch` per realizzare diramazioni basate su numeri in virgola mobile.

Ciascuna diramazione dell'enunciato `switch` dovrebbe essere terminata da un'istruzione `break`. Se `break` è assente, l'esecuzione *procede alla diramazione successiva*, anche ripetutamente, finché raggiunge `break` o la fine dell'enunciato `switch`. È abbastanza raro, nella pratica, che questo meccanismo (denominato *fall through*) sia utile: più spesso è fonte di errori. Se per caso dimenticate un'istruzione `break`, il programma viene compilato correttamente, ma si troverà, in alcuni casi, a eseguire codice indesiderato. Per questo motivo molti programmati ritengono che l'enunciato `switch` sia pericoloso e preferiscono usare enunciati `if` multipli.

Lasciamo a voi la scelta di usare o meno l'enunciato `switch` nei vostri programmi. Ad ogni modo, dovete essere in grado di analizzare e capire un enunciato `switch`, perché potrete incontrarlo nel codice di altri programmati.

## 5.4 Diramazioni annidate

Quando un enunciato di decisione è contenuto nel ramo di un altro enunciato di decisione, si dice che i due enunciati sono **annidati**.

Spesso è necessario inserire un enunciato `if` all'interno di un altro enunciato `if`: si parla, in tal caso, di enunciati **annidati** (*nested*).

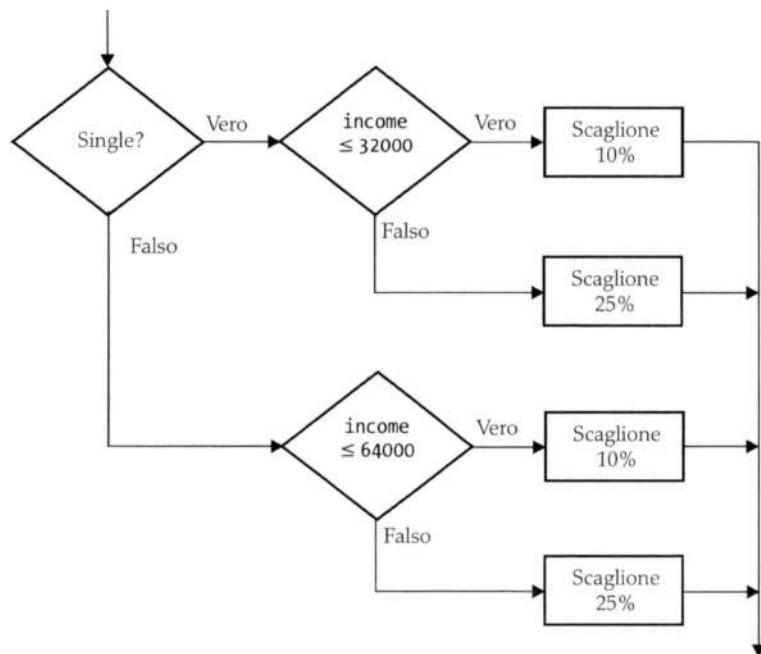
Ecco un esempio tipico. Negli Stati Uniti si applicano aliquote d'imposta diverse a seconda del reddito e dello stato civile del contribuente. Esistono due schemi principali, uno per contribuenti non coniugati e un altro per contribuenti coniugati, che fanno la “dichiarazione dei redditi coniuga”, cumulando i rispettivi redditi e pagando le tasse sul totale. La Tabella 4 riporta i calcoli dell'aliquota d'imposta per ciascuna categoria, usando una versione semplificata dei valori relativi alla dichiarazione federale dei redditi del 2008. Ad ogni “scaglione” di reddito si applica una diversa aliquota d'imposta e, in questo schema, al primo scaglione viene applicata un'aliquota

del 10%, mentre al secondo scaglione si applica un'aliquota del 25%: l'importo che delimita ciascuno scaglione di reddito, però, dipende dallo stato civile del contribuente (coniugato oppure no).

**Tabella 4**  
Schema per il calcolo delle tasse federali

| Se il vostro stato civile è "non coniugato" |          | Se il vostro stato civile è "coniugato" |          |
|---------------------------------------------|----------|-----------------------------------------|----------|
| Scaglione fiscale                           | Aliquota | Scaglione fiscale                       | Aliquota |
| \$ 0 ... \$ 32 000                          | 10%      | \$ 0 ... \$ 64 000                      | 10%      |
| Quota superiore a \$ 32 000                 | 25%      | Quota superiore a \$ 64 000             | 25%      |

**Figura 5**  
Calcolo dell'imposta sul reddito



Le decisioni annidate servono per risolvere problemi che richiedono decisioni su più livelli.

Ora calcoliamo le tasse dovute in base a un determinato stato civile e all'importo del relativo reddito, osservando un aspetto chiave: il risultato finale richiede *due livelli* di decisione. Per prima cosa dobbiamo prendere una decisione relativa allo stato civile; poi, per ciascuno stato civile, dobbiamo prendere un'altra decisione sulla base dello scaglione di reddito.

Il processo decisionale a due livelli si traduce, nel programma che termina il paragrafo, mediante due livelli di enunciati *if* (secondo il diagramma di flusso riportato nella Figura 5). In teoria, il processo di annidamento delle decisioni può coinvolgere anche più di due livelli: un processo di decisione a tre livelli (ad esempio, stato di residenza, stato civile e reddito) richiede un annidamento a tre livelli.

**File TaxReturn.java**

```
1  /**
2   * Tasse sul reddito di un contribuente nel 2008.
3  */
4  public class TaxReturn
5  {
6      public static final int SINGLE = 1;
7      public static final int MARRIED = 2;
8
9      private static final double RATE1 = 0.10;
10     private static final double RATE2 = 0.25;
11     private static final double RATE1_SINGLE_LIMIT = 32000;
12     private static final double RATE1_MARRIED_LIMIT = 64000;
13
14     private double income;
15     private int status;
16
17     /**
18      Costruisce un oggetto di tipo TaxReturn dato il
19      reddito e lo stato civile.
20      @param anIncome il reddito del contribuente
21      @param aStatus SINGLE oppure MARRIED
22     */
23     public TaxReturn(double anIncome, int aStatus)
24     {
25         income = anIncome;
26         status = aStatus;
27     }
28
29     public double getTax()
30     {
31         double tax1 = 0;
32         double tax2 = 0;
33
34         if (status == SINGLE)
35         {
36             if (income <= RATE1_SINGLE_LIMIT)
37             {
38                 tax1 = RATE1 * income;
39             }
40             else
41             {
42                 tax1 = RATE1 * RATE1_SINGLE_LIMIT;
43                 tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
44             }
45         }
46         else
47         {
48             if (income <= RATE1_MARRIED_LIMIT)
49             {
50                 tax1 = RATE1 * income;
51             }
52             else
53             {
```

```

54         tax1 = RATE1 * RATE1_MARRIED_LIMIT;
55         tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
56     }
57 }
58
59     return tax1 + tax2;
60 }
61 }
```

### File TaxCalculator.java

```

1 import java.util.Scanner;
2
3 /**
4     Programma che calcola le tasse per un contribuente.
5 */
6 public class TaxCalculator
7 {
8     public static void main(String[] args)
9     {
10         Scanner in = new Scanner(System.in);
11
12         System.out.print("Please enter your income: ");
13         double income = in.nextDouble();
14
15         System.out.print("Are you married? (Y/N) ");
16         String input = in.next();
17         int status;
18         if (input.equals("Y"))
19         {
20             status = TaxReturn.MARRIED;
21         }
22         else
23         {
24             status = TaxReturn.SINGLE;
25         }
26
27         TaxReturn aTaxReturn = new TaxReturn(income, status);
28         System.out.println("Tax: " + aTaxReturn.getTax());
29     }
30 }
```

### Esecuzione del programma

```

Please enter your income: 80000
Are you married? (Y/N) Y
Tax: 10400.0
```



### Auto-valutazione

19. Qual è l'ammontare delle tasse che deve pagare una persona non coniugata che abbia un reddito pari a \$32000?
20. Immaginate che il primo enunciato if annidato venga trasformato da:

```

if (income <= RATE1_SINGLE_LIMIT)
a:
    if (income < RATE1_SINGLE_LIMIT)

```

Che conseguenze avrebbe questa modifica sulla risposta fornita alla domanda precedente?

21. Se Harry e Sally hanno, ciascuno, un reddito annuo pari a \$40000 e si sposano, pagano meno tasse?
22. Come modifichereste il programma `TaxCalculator.java` per fare in modo che verifichi che l'utente abbia fornito un valore corretto per descrivere il suo stato civile (cioè Y o N)?
23. Alcuni cittadini sono contrari all'uso di aliquote crescenti per gli scaglioni di reddito superiori, affermando che, in questo modo, può succedere che, ottenendo un aumento di reddito conseguente a un lavoro più impegnativo, si abbia un reddito inferiore, al netto delle tasse. Cosa c'è di sbagliato in questo ragionamento?

### Per far pratica

È ora possibile risolvere gli esercizi R5.10, R5.22, E5.15 e E5.18, al termine del capitolo.



## Suggerimenti per la programmazione 5.5

### Eseguire a mano e tenere traccia su carta

Una tecnica molto utile per capire se un programma funziona correttamente consiste nell'eseguirlo a mano, tenendo traccia dei risultati su carta (*hand-tracing*): una tecnica utilizzabile sia con lo pseudocodice sia con il codice Java.

Prendete un foglio di carta e tracciate una colonna per ciascuna variabile. Tenete il codice del programma a portata di mano e usate un contrassegno (come una graffetta) per evidenziare l'enunciato in esecuzione. Eseguite mentalmente un enunciato dopo l'altro: ogni volta che una variabile viene modificata, scrivete il suo nuovo valore al di sotto di quello vecchio, sul quale tirate una riga (in modo che rimanga comunque visibile, per disporre di una traccia di ciò che è successo).

Come esempio, seguiamo passo dopo passo l'esecuzione del metodo `getTax`, usando i dati con cui abbiamo eseguito il programma in precedenza. Quando viene costruito l'oggetto di tipo `TaxReturn`, la sua variabile di esemplare `income` assume il valore 80000 e la sua variabile `status` diventa uguale a `MARRIED`. Viene poi invocato il metodo `getTax`: nelle righe 31 e 32 del file `TaxReturn.java`, azzera le variabili `tax1` e `tax2`.

```

29 public double getTax()
30 {
31     double tax1 = 0;
32     double tax2 = 0;

```

| income | status  | tax1 | tax2 |
|--------|---------|------|------|
| 80 000 | MARRIED | 0    | 0    |

Dato che `status` non è uguale a `SINGLE`, passiamo alla diramazione `else` dell'enunciato `if` più esterno (riga 46).

```

34     if (status == SINGLE)
35     {
36         if (income <= RATE1_SINGLE_LIMIT)

```

```

37     {
38         tax1 = RATE1 * income;
39     }
40     else
41     {
42         tax1 = RATE1 * RATE1_SINGLE_LIMIT;
43         tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
44     }
45 }
46 else
47 {

```

Ora, dato che `income` non è minore o uguale a 64000, passiamo alla diramazione `else` dell'enunciato `if` più interno (riga 52).

```

48     if (income <= RATE1_MARRIED_LIMIT)
49     {
50         tax1 = RATE1 * income;
51     }
52     else
53     {
54         tax1 = RATE1 * RATE1_MARRIED_LIMIT;
55         tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
56     }

```

I valori di `tax1` e `tax2` vengono aggiornati.

```

53     {
54         tax1 = RATE1 * RATE1_MARRIED_LIMIT;
55         tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
56     }
57 }

```

| income | status  | tax1 | tax2 |
|--------|---------|------|------|
| 80 000 | MARRIED | 0    | 0    |

Viene restituita la loro somma e l'esecuzione del metodo termina.

```

58
59     return tax1 + tax2;
60 }

```

| income | status  | tax1 | tax2 | valore restituito |
|--------|---------|------|------|-------------------|
| 80 000 | MARRIED | 0    | 0    | 10 400            |

Dato che la traccia del programma mostra che il metodo restituisce il valore previsto (\$10400), dimostra in modo efficace che questo collaudo funziona correttamente.



## Errori comuni 5.3

### Il problema dell'*else sospeso*

Quando si annida un enunciato *if* all'interno di un altro enunciato *if*, può verificarsi l'errore seguente:

```
double shippingCharge = 5.00; // $5 spedizione negli Stati Uniti continentali
if (country.equals("USA"))
    if (state.equals("HI"))
        shippingCharge = 10.00; // più costoso verso le Hawaii
    else // trappola!
        shippingCharge = 20.00; // spedizione all'estero
```

Il livello dei rientri suggerisce che la diramazione *else* sia associata alla verifica *country.equals("USA")*: sfortunatamente non è così. Il compilatore ignora i rientri e segue una regola ben precisa: un *else* appartiene sempre all'*if* che lo precede più da vicino. Ciò significa che il codice, in realtà, è il seguente:

```
double shippingCharge = 5.00;
if (country.equals("USA"))
    if (state.equals("HI"))
        shippingCharge = 10.00;
    else // trappola!
        shippingCharge = 20.00;
```

Non è ciò che volevamo, perché la nostra intenzione era quella di associare la diramazione *else* al primo *if*.

L'*else* ambiguo viene detto *else sospeso* ("dangling *else*"). Questo tipo di trappola si può evitare completamente se si usano sempre le parentesi graffe, come raccomandato nella sezione Suggerimenti per la programmazione 5.2:

```
double shippingCharge = 5.00; // $5 spedizione negli Stati Uniti continentali
if (country.equals("USA"))
{
    if (state.equals("HI"))
    {
        shippingCharge = 10.00; // più costoso verso le Hawaii
    }
}
else
{
    shippingCharge = 20.00; // spedizione all'estero
}
```



## Argomenti avanzati 5.3

### Ambito di visibilità limitato a un blocco

Un **blocco** (*block*) è una sequenza di enunciati racchiusa tra parentesi graffe. Ad esempio, nel frammento di codice seguente la porzione in grassetto costituisce un blocco:

```

if (status == TAXABLE)
{
    double tax = price * TAX_RATE;
    price = price + tax;
}

```

All'interno di un blocco è anche possibile dichiarare una variabile, come la variabile `tax` in questo esempio. Tale variabile è visibile soltanto all'interno del blocco stesso.

```

{
    double tax = price * TAX_RATE; // variabile dichiarata in un blocco
    price = price + tax;
}
// qui la variabile tax non è più accessibile

```

In effetti, la variabile viene creata soltanto nel momento in cui il flusso d'esecuzione del programma entra nel blocco, dopodiché viene eliminata non appena il programma esce dal blocco stesso: si parla di *ambito di visibilità relativo a un blocco* o, in breve, di “visibilità di blocco”. Più in generale, l'**ambito di visibilità** (*scope*) di una variabile è la porzione del programma da cui è possibile accedere alla variabile stessa; quindi, una variabile con visibilità di blocco è accessibile solamente all'interno di un blocco.

Si ritiene, in generale, che rendere minima l'estensione dell'ambito di visibilità di una variabile sia una buona consuetudine di progettazione, perché questo riduce la possibilità di modifiche accidentali e di conflitti tra nomi. In questo esempio, dal momento che non c'è bisogno di accedere alla variabile `tax` al di fuori del blocco in cui le viene assegnato un valore, è bene dichiararla all'interno del blocco stesso. Se, invece, la variabile serve anche al di fuori del blocco, bisogna definirla all'esterno. Ad esempio:

```

double tax = 0;
if (status == TAXABLE)
{
    tax = price * TAX_RATE;
}
price = price + tax;

```

In questo caso la variabile `tax` viene usata anche al di fuori del blocco che costituisce il corpo dell'enunciato `if`, quindi è necessario dichiararla all'esterno del blocco stesso.

In Java, l'ambito di visibilità di una variabile locale non può mai contenere la dichiarazione di un'altra variabile locale avente lo stesso nome. Ad esempio, il frammento di codice che segue contiene un errore:

```

double tax = 0;
if (status == TAXABLE)
{
    double tax = price * TAX_RATE;
    // ERRORE: non si può dichiarare un'altra variabile di nome tax
    price = price + tax;
}

```

È però possibile avere variabili locali con nomi identici se i loro ambiti di visibilità non hanno parti comuni, come in questo esempio:

```
if (Math.random() > 0.5)
{
    Rectangle r = new Rectangle(5, 10, 20, 30);
    ...
} // l'ambito di visibilità di r finisce qui
else
{
    int r = 5;
    // VA BENE: qui si può dichiarare un'altra variabile di nome r
    ...
}
```

Queste due variabili sono indipendenti l'una dall'altra. Si possono dichiarare variabili locali aventi lo stesso nome, purché i loro ambiti di visibilità non si sovrappongano.



## Argomenti avanzati 5.4

### Tipi enumerativi

In molti programmi si usano variabili che possono assumere soltanto un limitato numero di valori. Nell'esempio sul calcolo delle tasse abbiamo visto che la variabile di esemplare `status` poteva assumere soltanto il valore `SINGLE` oppure `MARRIED`, per cui abbiamo scelto di definire, in modo del tutto arbitrario, la costante `SINGLE` uguale a 1 e la costante `MARRIED` uguale a 2. Se, per qualche errore di programmazione, la variabile `status` assumesse qualsiasi altro valore intero, come -1, 0 o 3, la logica del programma potrebbe dare luogo a risultati non validi.

In un programma semplice come il nostro questo non è un vero problema, ma, dal momento che i programmi aumentano di dimensione nel tempo e vengono ad aggiungersi ulteriori casi (come potrebbe essere la categoria "sposato con dichiarazione dei redditi disgiunta"), potrebbero presentarsi nuovi errori. Java presenta una soluzione per questi problemi: i **tipi enumerativi** (*enumeration type*). Una variabile di un tipo enumerativo può assumere soltanto un valore che appartenga a un insieme definito in questo modo:

```
public enum FilingStatus { SINGLE, MARRIED }
```

Potete definire un numero di valori qualsiasi, ma li dovete elencare tutti nella dichiarazione `enum`.

Dopo aver dichiarato il tipo enumerativo si possono dichiarare variabili:

```
FilingStatus status = FilingStatus.SINGLE;
```

Se cercate di assegnare a `status` un valore che non appartenga a `FilingStatus`, come 2 o "S", il compilatore segnala un errore.

Per confrontare valori di un tipo enumerativo si usa l'operatore `==`, come in questo esempio:

```
if (status == FilingStatus.SINGLE) . . .
```

Soltanamente le dichiarazioni di tipo `enum` vengono inserite all'interno di una classe:

```
public class TaxReturn
```

```
{
    public enum FilingStatus { SINGLE, MARRIED, MARRIED_FILING_SEPARATELY }
    ...
}
```

## 5.5 Problem Solving: diagrammi di flusso

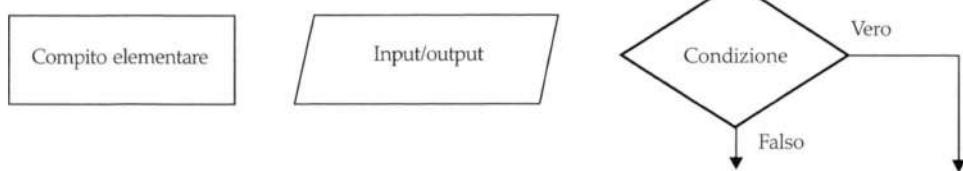
I diagrammi di flusso (*flowchart*) sono costituiti da elementi che rappresentano compiti da svolgere, acquisizione e visualizzazione di dati (input/output) e decisioni da prendere.

Nei paragrafi precedenti di questo capitolo avete visto esempi di diagrammi di flusso (*flowchart*). Un diagramma di flusso descrive graficamente le strutture di decisione e i compiti che vanno portati a termine per risolvere un problema. Quando dovete risolvere un problema complesso, è spesso molto utile disegnare un diagramma di flusso che visualizzi, appunto, il flusso delle operazioni da svolgere e delle attività di controllo.

I componenti elementari dei diagrammi di flusso sono rappresentati nella Figura 6.

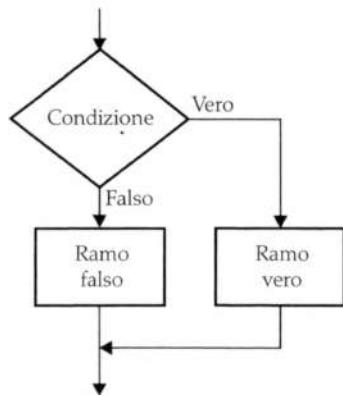
**Figura 6**

Elementi di un diagramma di flusso



**Figura 7**

Diagramma di flusso con due alternative



L'idea su cui si basano i diagrammi di flusso è abbastanza semplice: collegare i blocchi che rappresentano compiti da svolgere e fasi di input/output secondo la sequenza in cui vanno eseguiti, disegnando un rombo con due uscite ogni volta che c'è bisogno di prendere una decisione (si veda la Figura 7 per un esempio).

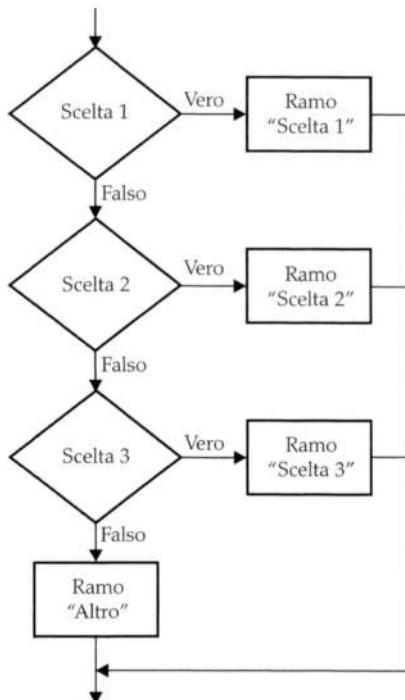
Ogni diramazione può contenere una sequenza di compiti da svolgere e ulteriori decisioni da prendere. Nel caso siano presenti alternative multiple controllate da un unico valore, è preferibile disporle graficamente come nella Figura 8.

Quando si disegnano diagrammi di flusso occorre tener presente un potenziale problema: la fusione di più flussi di esecuzione può portare al cosiddetto “codice a spaghetti” (*spaghetti code*), una rete confusa di possibili percorsi di flusso nel programma.

Ogni ramo di una decisione può contenere compiti da svolgere e altre decisioni da prendere.

**Figura 8**

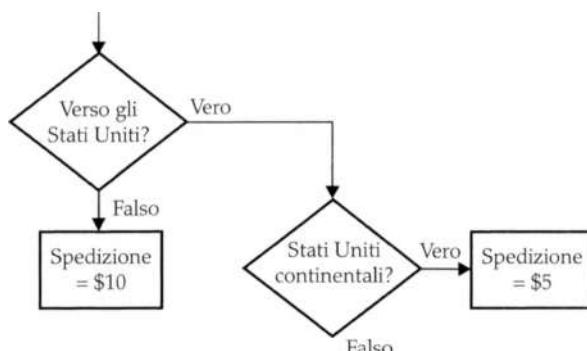
Diagramma di flusso con più alternative



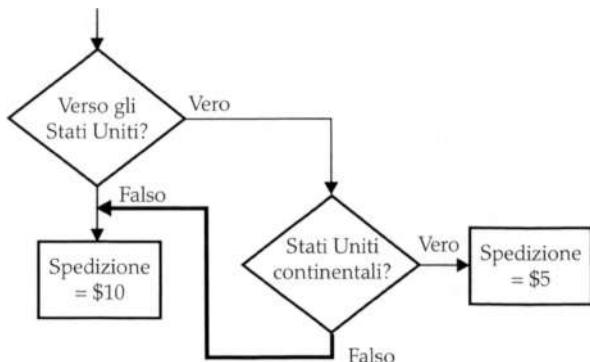
Non far mai entrare una freccia all'interno di una diversa diramazione.

Per evitare il codice a spaghetti è sufficiente rispettare questa semplice regola: non far mai entrare una freccia *all'interno* di una diversa diramazione.

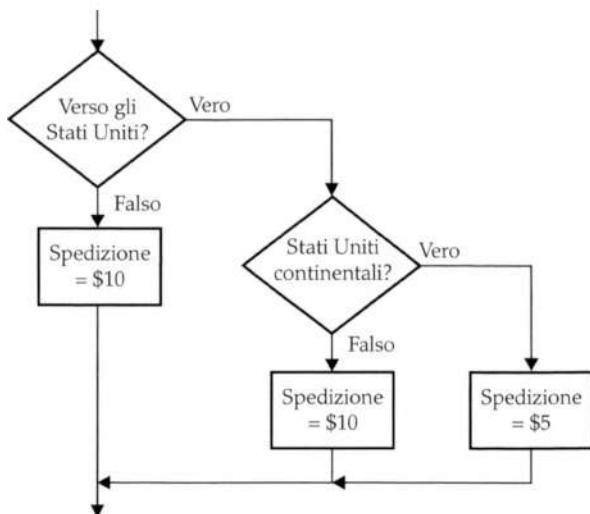
Per comprendere tale regola, analizziamo questo esempio: i costi di spedizione sono \$5 all'interno degli Stati Uniti, tranne nel caso di Hawaii e Alaska, che richiedono un costo di \$10, esattamente come le spedizioni internazionali. Si potrebbe partire da un diagramma di flusso come questo:



Per gli Stati Uniti *non* continentali potreste cedere alla tentazione di riutilizzare il compito denominato "Spedizione = \$10", in questo modo:



Non fatelo! La freccia evidenziata (di spessore maggiore) punta all'interno di un ramo diverso da quello da cui parte. Piuttosto, aggiungete un ulteriore blocco che imposta il costo di spedizione a \$10, in questo modo:



Così facendo, non solo si evita di creare codice a spaghetti, ma si ottiene anche un progetto migliore: in futuro potrebbe accadere che i costi di spedizione per destinazioni internazionali diventino diversi dal costo di spedizione relativo ad Alaska e Hawaii.

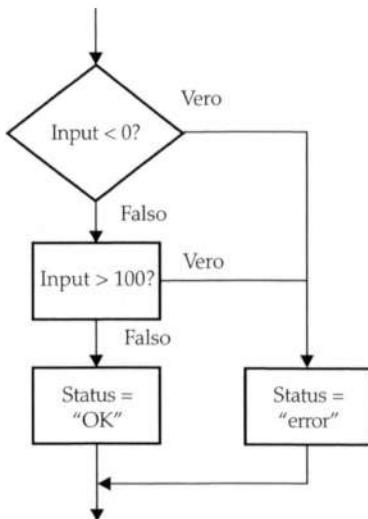
I diagrammi di flusso possono essere decisamente utili per comprendere in modo intuitivo il flusso di esecuzione di un programma, ma tendono a diventare rapidamente enormi quando si scende a un livello di dettaglio maggiore. A quel punto, conviene passare dal diagramma di flusso allo pseudocodice.



### Auto-valutazione

24. Disegnate un diagramma di flusso per un programma che legga un valore di temperatura (`temp`) e visualizzi il messaggio "Frozen" se è inferiore a zero.

25. Cosa c'è di sbagliato in questo diagramma di flusso?



26. Come si può correggere il diagramma di flusso della domanda precedente?  
 27. Disegnate un diagramma di flusso per un programma che legga un valore,  $x$ . Se tale valore è minore di zero il programma visualizza “Error”, altrimenti ne visualizza la radice quadrata.  
 28. Disegnate un diagramma di flusso per un programma che legga un valore di temperatura,  $\text{temp}$ . Se tale valore è inferiore a zero il programma visualizza il messaggio “Ice”; se, invece, è maggiore di 100, visualizza “Steam”; altrimenti, visualizza “Liquid”.

### Per far pratica

È ora possibile risolvere gli esercizi R5.13, R5.14 e R5.15, al termine del capitolo.

## 5.6 Problem Solving: preparare casi di prova

Il collaudo a scatola chiusa (*black-box testing*) descrive una metodologia di collaudo che non prende in considerazione la struttura dell’implementazione.

Il collaudo trasparente (*white-box testing*) usa informazioni sulla struttura del programma.

La copertura (*coverage*) di un collaudo misura quante parti di un programma sono state collaudate.

Quando si collaudano le funzionalità di un programma senza tenere conto della sua struttura interna si parla di *black-box testing*, ovvero di “collaudo a scatola chiusa”. È una strategia di collaudo importante: in fin dei conti, gli utenti di un programma non conoscono la sua struttura interna. Se un programma funziona perfettamente con tutti i dati di ingresso, significa che fa bene il suo lavoro.

Tuttavia, utilizzando un numero limitato di casi di prova, è impossibile avere la certezza assoluta che un programma funzioni correttamente con tutti i valori di ingresso. Come osservò il famoso ricercatore informatico Edsger Dijkstra, il collaudo può evidenziare solamente la presenza di errori, non la loro assenza.

Per poter giungere a una maggiore confidenza in merito alla correttezza di un programma, è utile tenere conto della sua struttura interna. Le tecniche di collaudo che guardano all'interno di un programma sono chiamate *white-box testing* (“collaudo trasparente”) ed eseguire collaudi di unità su ciascun metodo fa parte di questa tecnica.

Nel white-box testing ci si vuole accertare che ogni porzione del programma venga collaudata da almeno uno dei casi di prova. Questa informazione viene rappresentata dalla **copertura del codice** da parte del collaudo (*code coverage*): se qualche porzione di

codice non viene mai eseguita dai casi di prova, non avrete modo di sapere se tale porzione di codice funzionerà correttamente nel caso venga attivata dai dati forniti in ingresso dall'utente. Per raggiungere questo scopo, dovete esaminare ogni diramazione `if/else`, per controllare che ciascuna venga eseguita da qualche caso di prova. Molte diramazioni sono inserite nel codice solamente per gestire valori strani o anomali, ma eseguono comunque qualche operazione: accade di frequente che finiscano per fare qualcosa di scorretto, ma questi errori non vengono mai scoperti durante il collaudo, semplicemente perché nessuno ha fornito valori strani o anomali. Il problema si risolve garantendo che ciascuna parte di codice venga coperta da qualche caso di prova.

Per esempio, per collaudare il metodo `getTax` della classe `TaxReturn` occorre garantire che ciascun enunciato `if` venga eseguito da almeno un caso di prova, usando entrambi i tipi di contribuenti, "coniugati" e "non coniugati", con redditi appartenenti a ciascuno degli scaglioni fiscali.

Quando selezionate i casi di prova, dovreste prendere l'abitudine di considerare anche **casi limite** (*boundary test case*): valori validi dei dati di ingresso che, però, si trovano al limite dei valori accettabili.

Ecco un piano che consente di ottenere un insieme di casi di prova completo per il programma che calcola le tasse dovute da un contribuente:

- ci sono due possibili valori di stato civile e due scaglioni di tassazione per ciascuno di essi, dando luogo a quattro casi;
- bisogna collaudare un certo numero di casi limite, ad esempio un reddito che si trova al confine tra due scaglioni, oppure un reddito uguale a zero;
- se avete il compito di verificare che non ci siano errori (*error checking*, di cui si parlerà nel Paragrafo 5.8), fate un collaudo anche con valori non validi, come un reddito negativo.

Fate un elenco dei casi di prova e dei risultati previsti:

| Caso di prova | Sposato | Previsto | Commento             |
|---------------|---------|----------|----------------------|
| 30 000        | No      | 3000     | Scaglione 10%        |
| 72 000        | No      | 13 200   | 3200 + 25% di 40 000 |
| 50 000        | Sì      | 5000     | Scaglione 10%        |
| 104 000       | Sì      | 16 400   | 6400 + 25% di 40 000 |
| 32 000        | No      | 3200     | Caso limite          |
| 0             |         | 0        | Caso limite          |

Quando si sviluppa un insieme di casi di prova, è utile consultare il diagramma di flusso del programma (Paragrafo 5.5): verificate che ogni diramazione abbia un caso di prova e inserite casi di prova per i casi limite di ciascuna decisione che viene presa dal programma. Ad esempio, se una decisione dipende dalla verifica che un dato acquisito in ingresso sia inferiore a 100, inserite un caso di prova con valore d'ingresso uguale a 100.

È sempre bene progettare i casi di prova *prima* di iniziare a scrivere il codice del programma. La progettazione dei casi di prova, infatti, vi consente di comprendere meglio l'algoritmo che state per implementare.

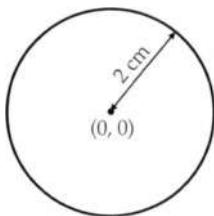
I casi limite sono casi di prova che usano valori limite dei dati in ingresso.

È bene progettare i casi di prova prima di scrivere il codice di un programma.



## Auto-valutazione

29. Usando come traccia la Figura 1 di questo capitolo, seguite il procedimento delineato in questo paragrafo per progettare un insieme di casi di prova per il programma `ElevatorSimulation.java` visto nel Paragrafo 5.1.
30. Qual è un possibile caso limite per l'algoritmo riportato nella sezione Consigli pratici 5.1? Qual è il risultato previsto?
31. Usando come traccia la Figura 4 di questo capitolo, seguite il procedimento delineato in questo paragrafo per progettare un insieme di casi di prova per il programma `Earthquake.java` visto nel Paragrafo 5.3.
32. Immaginate di dover progettare parte di un programma di controllo di un robot dotato di un sensore che restituisce le coordinate  $x$  e  $y$  della sua posizione (misurate in cm) e di dover verificare se tale posizione si trova all'interno (“inside”), all'esterno (“outside”) o sulla circonferenza (“on the boundary”) di un cerchio avente centro nel punto di coordinate  $(0, 0)$  e raggio di 2 cm (la posizione va ritenuta appartenente alla circonferenza se la sua distanza da essa è inferiore a 1 mm). Fornite un insieme di casi di prova.



### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R5.16 e R5.17, al termine del capitolo.



## Suggerimenti per la programmazione 5.6

### Preparare un piano e riservarsi tempo per problemi imprevisti

È risaputo che il software commerciale viene spesso reso disponibile oltre la data prevista. Ad esempio, Microsoft inizialmente assicurò che il suo sistema operativo Windows Vista sarebbe stato disponibile alla fine del 2003, poi nel 2005, poi nel marzo 2006: venne finalmente distribuito nel gennaio 2007. Alcune delle prime promesse non potevano essere realistiche, ma rientra negli interessi di Microsoft tenere i potenziali acquirenti in continua attesa di una disponibilità imminente del prodotto, in modo che, nel frattempo, non decidano di passare a un prodotto concorrente. Innegabilmente, però, Microsoft non aveva fatto trasparire tutta la complessità dei problemi che si era proposta di risolvere.

Microsoft può ritardare il lancio dei suoi prodotti, ma probabilmente voi non potrete farlo: in qualità di studenti o di programmati, ci si aspetta che organizziate saggiamente il vostro tempo e che terminiate ciò che vi è stato assegnato secondo le scadenze previste. Durante la notte che precede la data di consegna potrete probabilmente fare esercizi di programmazione semplici, ma un compito che sembra difficile il doppio può facilmente richiedere quattro volte il tempo previsto, perché possono verificarsi molti imprevisti. Pertanto, quando iniziate un progetto di programmazione, dovreste fare un piano di lavoro.

Per prima cosa, stimate realisticamente quanto tempo vi occorrerà per:

- progettare la logica del programma
- sviluppare i casi di prova
- digitare il programma e correggere gli errori di sintassi
- collaudare il programma e correggere gli errori logici

Ad esempio, per il programma che calcola l'imposta sul reddito potremmo stimare: un'ora per il progetto; 30 minuti per sviluppare casi di prova; un'ora per scrivere il codice al calcolatore e correggere gli errori di sintassi; infine, un'ora per il collaudo e per correggere gli errori logici. Il totale è di tre ore e mezza: lavorando a questo progetto due ore al giorno, occorreranno almeno due giorni.

Bisogna poi prevedere le cose che possono andare storte: il computer potrebbe guastarsi oppure un problema con il sistema operativo potrebbe disorientarvi (quest'ultimo caso è particolarmente importante per i principianti: è *molto* frequente perdere una giornata intera per un problema banale, solo perché occorre tempo per rintracciare una persona che conosca il comando "magico" per risolverlo). Come regola empirica, *raddoppiate* il tempo che avete stimato. In questo caso significa che dovete iniziare quattro giorni prima della scadenza, anziché due. Se tutto va bene avrete il programma pronto due giorni prima; per contro, se si verificano problemi inevitabili avrete una riserva di tempo che vi proteggerà dall'imbarazzo e dal fallimento.



## Argomenti avanzati 5.5

### Tracciamento (*logging*)

A volte capita di eseguire un programma e di non capire dove stia perdendo tempo. Per avere un prospetto del flusso di esecuzione del programma potete inserirvi **messaggi di tracciamento** (*trace message*), come questo:

```
if (status == SINGLE)
{
    System.out.println("status is SINGLE");
    ...
}
```

La visualizzazione dei messaggi di tracciamento con `System.out.println` pone, però, un problema: quando avete terminato il collaudo del programma dovete eliminare tutti gli enunciati che visualizzano messaggi di tracciamento. Se, poi, trovate un ulteriore errore, dovete inserirli nuovamente.

Per risolvere questo problema dovreste usare la classe `Logger`, che consente di zittire i messaggi di tracciamento senza eliminare dal programma gli enunciati che li generano.

Invece di inviare dati direttamente al flusso `System.out`, usate l'oggetto di tracciamento globale che viene restituito dall'invocazione `Logger.getGlobal()` (nelle versioni di Java precedenti alla versione 7 questo oggetto era restituito dall'invocazione `Logger.getLogger("global")`) e invocatene il metodo `info`:

```
Logger.getGlobal().info("status is SINGLE");
```

I messaggi di tracciamento possono essere disattivati dopo aver completato il collaudo.

Per impostazione predefinita il messaggio viene visualizzato sul flusso di uscita standard. Se, però, all'inizio del metodo `main` del vostro programma invocate:

```
Logger.getGlobal().setLevel(Level.OFF);
```

la visualizzazione di tutti i messaggi di tracciamento viene soppressa. Riportando, invece, il livello di attenzione al valore `Level.INFO` il tracciamento riprende. In questo modo potete sopprimere i messaggi di tracciamento quando il programma funziona correttamente e riabilitarli se trovate un nuovo errore. In altre parole, usare `Logger.getGlobal().info` è come usare `System.out.println`, con la differenza che è possibile attivare e disattivare il tracciamento in modo molto semplice. La classe `Logger` ha molte altre opzioni, che consentono un'attività di tracciamento adatta ad ambienti di programmazione professionale: per avere un maggiore controllo sui messaggi di tracciamento potete controllare la documentazione API della classe

## 5.7 Variabili booleane e operatori

Il tipo booleano, `boolean`, ha due valori: `false` e `true`.

A volte c'è bisogno di valutare una condizione logica in un punto del programma, per poi utilizzarla altrove. Per memorizzare una condizione, che può essere vera o falsa, si usa una **variabile booleana**, che prende il nome dal matematico George Boole (1815–1864), un pioniere nello studio della logica.

In Java, il tipo di dato `boolean` ha esattamente due valori, denominati `false` e `true`. Questi due valori non sono stringhe, né numeri interi: sono valori speciali, validi soltanto per le variabili booleane. Ecco l'inizializzazione di una variabile che viene impostata al valore `true`:

```
boolean failed = true;
```

Il suo valore può essere utilizzato più avanti nel programma per prendere una decisione:

```
if (failed) // ramo eseguito solo se failed ha il valore true
{
    ...
}
```

Java ha due operatori booleani che combinano condizioni: `&& (and)` e `|| (or)`.

Quando si devono prendere decisioni complesse, spesso si ha bisogno di combinare valori booleani: un operatore che combina valori booleani è detto **operatore booleano**. In Java, l'operatore `&&` (chiamato *and*) restituisce il valore `true` quando entrambe le condizioni sono vere, mentre l'operatore `||` (chiamato *or*) restituisce `true` se almeno una delle condizioni è vera.

**Figura 9**

Tabelle della verità degli operatori booleani

| A     | B     | A && B | A     | B     | A    B | A     | !A    |
|-------|-------|--------|-------|-------|--------|-------|-------|
| true  | true  | true   | true  | true  | true   | true  | false |
| true  | false | false  | true  | false | true   | false | true  |
| false | true  | false  | false | true  | true   |       |       |
| false | false | false  | false | false | false  |       |       |

Immaginiamo di voler scrivere un programma che elabori valori di temperatura e di voler verificare se una data temperatura corrisponde ad acqua liquida (ricordando che, al livello del mare, l'acqua ghiaccia a 0 gradi e bolle a 100 gradi, secondo la scala Celsius). L'acqua è liquida se la temperatura è maggiore di zero e minore di 100:

```
if (temp > 0 && temp < 100) { System.out.println("Liquid"); }
```

La condizione che viene verificata è costituita da due parti, unite dall'operatore `&&`. Ciascuna parte è un valore booleano, che può essere, quindi, vero o falso. L'espressione composta è vera se entrambe le singole parti componenti sono vere, mentre, se almeno una delle due è falsa, il risultato sarà falso (si veda la Figura 9).

Gli operatori booleani `&&` e `||` hanno una precedenza inferiore a quella degli operatori relazionali `e`, per questo motivo, si possono scrivere espressioni relazionali ai lati degli operatori booleani senza bisogno di parentesi. Per esempio, nell'espressione:

```
temp > 0 && temp < 100
```

le espressioni `temp > 0` e `temp < 100` vengono valutate per prime, poi l'operatore `&&` combina i risultati (l'Appendice B riporta una tabella con tutti gli operatori di Java e le loro precedenze).

Viceversa, proviamo a verificare che l'acqua *non* sia liquida a una data temperatura, cosa che avviene quando la temperatura è al massimo 0 o come minimo 100. Usiamo, quindi, l'operatore `||` (*or*) per combinare le relative espressioni:

```
if (temp <= 0 || temp >= 100) { System.out.println("Not liquid"); }
```

La Figura 10 mostra i diagrammi di flusso per questi esempi.

Per invertire una condizione  
si usa l'operatore `!` (*not*).

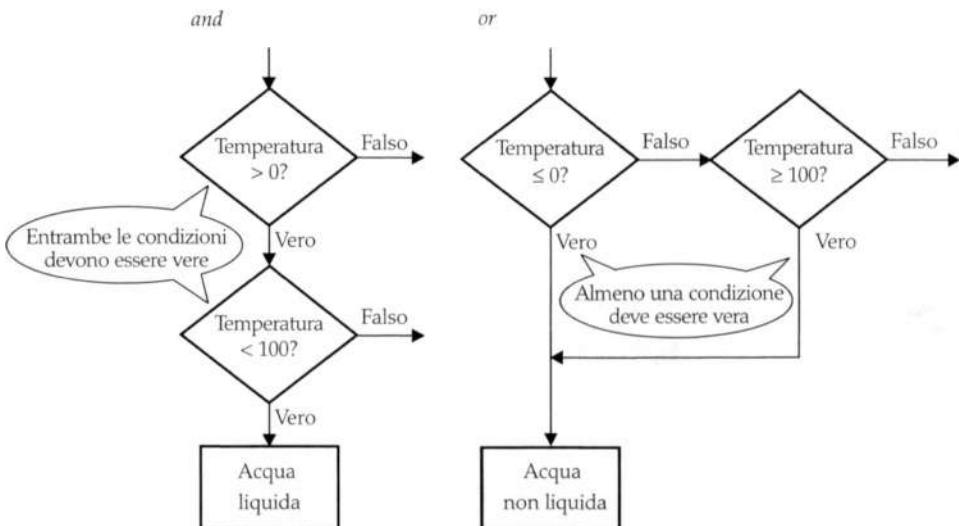
A volte c'è bisogno di *invertire* una condizione: si usa l'operatore booleano `!` (*not*), che analizza una singola condizione e calcola il valore `true` se tale condizione è falsa, `false` se è vera. In questo esempio, la visualizzazione del messaggio avviene se la variabile `frozen` ha il valore `false`:

```
if (!frozen) { System.out.println("Not frozen"); }
```

La Tabella 5 mostra ulteriori esempi di valutazione di operatori booleani.

**Figura 10**

Diagrammi di flusso per combinazioni mediante gli operatori `&&` e `||`

**Tabella 5** Esempi con operatori booleani

| Espressione                                            | Valore                                                                     | Commento                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|--------------------------------------------------------|----------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>0 &lt; 200 &amp;&amp; 200 &lt; 100</code>        | false                                                                      | Soltanto la prima condizione è vera.                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <code>0 &lt; 200    200 &lt; 100</code>                | true                                                                       | La prima condizione è vera.                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <code>0 &lt; 200    100 &lt; 200</code>                | true                                                                       | L'operatore <code>  </code> non è una verifica di tipo "o uno o l'altro": se entrambe le condizioni sono vere, il risultato è vero.                                                                                                                                                                                                                                                                                                                                       |
| <code>0 &lt; x &lt; 100</code>                         | Errore di sintassi                                                         | <b>Errore:</b> Questa espressione non verifica se <code>x</code> è compreso tra 0 e 100. L'espressione <code>0 &lt; x</code> ha un valore booleano, che non può poi essere confrontato con il numero 100.                                                                                                                                                                                                                                                                 |
| <code>x &amp;&amp; y &gt; 0</code>                     | Errore di sintassi                                                         | <b>Errore:</b> Questa espressione non verifica se <code>x</code> e <code>y</code> hanno valori positivi. A sinistra dell'operatore <code>&amp;&amp;</code> c'è un valore numerico intero, <code>x</code> , mentre a destra c'è un valore booleano, il risultato del confronto <code>y &gt; 0</code> : non si può usare <code>&amp;&amp;</code> con un operando numerico. L'operatore <code>&amp;&amp;</code> ha una priorità più elevata dell'operatore <code>  </code> . |
| <code>0 &lt; x &amp;&amp; x &lt; 100    x == -1</code> | ( <code>0 &lt; x &amp;&amp; x &lt; 100</code> )<br><code>   x == -1</code> | <code>0 &lt; 200</code> vale true, quindi il suo inverso è false.                                                                                                                                                                                                                                                                                                                                                                                                         |
| <code>!(0 &lt; 200)</code>                             | false                                                                      | <code>0 &lt; 200</code> vale true, quindi il suo inverso è false.                                                                                                                                                                                                                                                                                                                                                                                                         |
| <code>frozen == true</code>                            | frozen                                                                     | Non c'è alcun bisogno di confrontare una variabile booleana con il valore <code>true</code> .                                                                                                                                                                                                                                                                                                                                                                             |
| <code>frozen == false</code>                           | <code>!frozen</code>                                                       | Piuttosto che confrontare con il valore <code>false</code> , l'uso dell'operatore <code>!</code> è più esplicito.                                                                                                                                                                                                                                                                                                                                                         |



### Auto-valutazione

33. Se `x` e `y` sono numeri interi, come si controlla se sono entrambi uguali a zero?
34. Se `x` e `y` sono numeri interi, come si controlla se almeno uno dei due è uguale a zero?
35. Se `x` e `y` sono numeri interi, come si controlla se *esattamente uno* dei due è uguale a zero?

36. Qual è il valore di `!!frozen`?
37. Che vantaggio c'è a usare il tipo `boolean` invece delle stringhe "false" e "true" o dei numeri interi 0 e 1?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R5.30, E5.23 e E5.24, al termine del capitolo.



## Errori comuni 5.4

### Espressioni con più operatori relazionali

Esaminate l'espressione seguente:

```
if (0 <= temp <= 100) . . . // Errore
```

Benché l'enunciato assomigli all'espressione matematica " $0 \leq \text{temp} \leq 100$ ", purtroppo in Java costituisce un errore di sintassi.

Proviamo ad analizzare la condizione. La prima parte, `0 <= temp`, è una verifica che ha come risultato `true` o `false`, in funzione del valore di `temp`. L'esito di tale verifica (`true` o `false`) viene poi confrontato con `100`: un confronto privo di senso. Il valore `true` è più grande di `100` oppure no? Si possono confrontare valori di verità e numeri? In Java non si può e il compilatore rifiuta questo enunciato.

Usiamo, invece, l'operatore `&&` per combinare due verifiche distinte:

```
if (0 <= temp && temp <= 100) . . .
```

Un altro errore comune, dello stesso genere, è quello di scrivere un enunciato come il seguente, che vorrebbe verificare se `input` sia uguale a 1 o a 2:

```
if (input == 1 || 2) . . . // Errore
```

Ancora una volta il compilatore Java segnalerà che questo costrutto è errato, perché non si può applicare l'operatore `||` a valori numerici. In questo caso dovete scrivere due espressioni booleane e unirle con l'operatore `||`:

```
if (input == 1 || input == 2) . . .
```



## Errori comuni 5.5

### Confondere gli operatori `&&` e `||`

Confondere gli operatori *and* e *or* è un errore sorprendentemente comune. Un valore si colloca fra 0 e 100 se è almeno uguale a zero *e*, al massimo, uguale a 100. Per contro, cade al di fuori di questo intervallo se è minore di zero *o* maggiore di 100. Non esiste una "regola d'oro", quindi dovete solo riflettere con attenzione.

Spesso gli operatori *and* e *or* sono enunciati chiaramente e l'espressione che li contiene si può implementare senza troppa difficoltà, ma altre volte la formulazione non è così

esplicita. Accade di frequente che singole condizioni vengano accuratamente separate in un elenco per punti, ma con scarse indicazioni sul modo di combinarle.

Considerate queste istruzioni per la denuncia dei redditi, che affermano che potete dichiarare lo stato di "non coniugato" se è vera una qualsiasi delle condizioni seguenti:

- Non vi siete mai sposati.
- Eravate legalmente separati o divorziati l'ultimo giorno dell'anno.
- Eravate vedovi e non vi siete risposati.

Dal momento che la verifica ha esito positivo se *una qualsiasi* delle condizioni è vera, dovete combinare tutte le condizioni mediante l'operatore *or*.

D'altra parte, le stesse istruzioni affermano che potete avvalervi del più vantaggioso stato di coniugato, con dichiarazione congiunta, se sono vere tutte e cinque le condizioni seguenti:

- Il coniuge è deceduto meno di due anni fa e non vi siete risposati.
- Avete un figlio che potete dichiarare a vostro carico.
- Il figlio ha abitato presso di voi per tutto l'anno.
- Avete pagato oltre la metà del costo di gestione della casa per vostro figlio.
- Avete presentato una dichiarazione congiunta con il coniuge nell'anno in cui è deceduto.

Poiché *tutte* le condizioni devono essere vere per il buon esito della verifica, dovete combinarle mediante l'operatore *and*.



## Argomenti avanzati 5.6

### Valutazione "in cortocircuito" degli operatori booleani

Gli operatori `&&` e `||` vengono valutati mediante *cortocircuito*: non appena viene determinato il risultato, non viene valutata alcuna ulteriore condizione.

Gli operatori `&&` e `||` sono calcolati, in Java, usando la valutazione *in cortocircuito*. In altre parole, le espressioni logiche vengono esaminate da sinistra a destra e l'analisi cessa appena viene determinato il valore finale. Quando viene valutato un operatore *and* e la prima condizione è falsa, la seconda non viene esaminata: non importa che cosa esprime, perché comunque la condizione combinata è falsa. Ecco un esempio:

```
quantity > 0 && price / quantity < 10
```

Se il valore di `quantity` è zero, allora la prima condizione, `quantity > 0`, è falsa e, quindi, la seconda verifica non viene valutata: una cosa utile, perché non sarebbe lecito eseguire una divisione per zero.

Analogamente, quando la prima condizione di un operatore `||` è vera, la seconda non viene esaminata, perché il risultato sarà comunque vero.

Questa procedura si chiama **valutazione in cortocircuito**.



## Argomenti avanzati 5.7

### La legge di De Morgan

Gli esseri umani generalmente fanno fatica a comprendere le condizioni logiche che applicano operatori *not* a espressioni di tipo *and/or*. Per semplificare queste espressioni booleane si può usare la **legge di De Morgan**, dal nome del logico Augustus De Morgan (1806-1871).

Immaginiamo di voler applicare costi di spedizione più elevati nel caso in cui la destinazione non si trovi negli Stati Uniti continentali.

```
if (!(country.equals("USA") && !state.equals("AK") && !state.equals("HI")))
{
    shippingCharge = 20.0;
}
```

Questa verifica è un po' complicata e dovete riflettere attentamente sulla sua logica: quando *non* è vero che la nazione è USA e lo stato non è Alaska e lo stato non è Hawaii, allora il costo di spedizione è \$20.00. Cosa? Non è vero che alcune persone non saranno confuse da questo codice!

Al computer non importa, ma gli esseri umani devono fare molta attenzione quando scrivono o modificano questo codice. È, quindi, decisamente utile sapere come si possa semplificare una tale condizione.

La legge di De Morgan si usa in due forme, una per la negazione di un'espressione *and* e una per la negazione di un'espressione *or*:

$$\begin{array}{lll} !(A \&\& B) & \text{è uguale a} & !A \mid\mid !B \\ !(A \mid\mid B) & \text{è uguale a} & !A \&\& !B \end{array}$$

La legge di De Morgan indica come semplificare espressioni in cui l'operatore *not* sia applicato a espressioni *and* o *or*.

Fate particolare attenzione al fatto che gli operatori *and* e *or* vengono *scambiati* quando si sposta l'operatore *not* all'interno delle parentesi. Ad esempio, la negazione di "lo stato è Alaska o è Hawaii", cioè

`!(state.equals("AK") || state.equals("HI"))`

diventa "lo stato non è Alaska e non è Hawaii", cioè

`!state.equals("AK") && !state.equals("HI")`

Proviamo, ora, ad applicare questa legge al nostro calcolo delle spese di spedizione:

`!(country.equals("USA") && !state.equals("AK") && !state.equals("HI"))`

equivale a:

`!country.equals("USA") \mid\mid !!state.equals("AK") \mid\mid !!state.equals("HI")`

Dato che due negazioni si cancellano, l'espressione risultante è ancora più semplice:

`!country.equals("USA") \mid\mid state.equals("AK") \mid\mid state.equals("HI")`

In altre parole, le spese di spedizione più elevate si applicano quando la destinazione è al di fuori degli Stati Uniti oppure è l'Alaska o le Hawaii.

Riassumendo, spesso è utile usare la legge di De Morgan per semplificare condizioni espresse mediante la negazione di operatori *and* o *or*, spostando la negazione stessa a un livello più interno.

## 5.8 Applicazione: validità dei dati in ingresso

Il controllo dei dati in ingresso (*input validation*) è un'importante applicazione dell'enunciato *if*. Ogni volta che un programma acquisisce dati, bisogna accertarsi che i valori forniti dall'utente siano validi prima di poterli usare nell'elaborazione.

Torniamo ad analizzare il programma di simulazione di un ascensore. Nell'ipotesi che il pannello di controllo dell'ascensore abbia i pulsanti etichettati con i numeri che vanno da 1 a 20 (con l'esclusione del numero 13), i valori qui elencati non sono validi:

- il numero 13
- zero o un numero negativo
- un numero maggiore di 20
- un dato in ingresso che non sia una sequenza di cifre, come *five*

In ognuno di questi casi vogliamo che il programma visualizzi un messaggio d'errore e termini la propria esecuzione.

Proteggersi dall'inserimento del numero 13 è semplice:

```
if (floor == 13)
{
    System.out.println("Error: There is no thirteenth floor.");
}
```

Ecco, invece, come assicurarsi che l'utente non abbia inserito un numero che non appartiene all'intervallo dei valori validi:

```
if (floor <= 0 || floor > 20)
{
    System.out.println("Error: The floor must be between 1 and 20.");
}
```

Gestire un dato in ingresso che non sia un numero intero è decisamente più problematico. Quando viene eseguito l'enunciato seguente:

```
floor = in.nextInt();
```

e l'utente digita una sequenza di caratteri che non costituisce un numero intero (come, ad esempio, *five*), alla variabile *floor* non viene assegnato alcun valore, perché si verifica una **eccezione** durante l'esecuzione del programma (*run-time exception*), che termina. Per evitare questo problema, bisogna invocare prima il metodo *hasNextInt*, che verifica se il prossimo dato disponibile è un numero intero: se tale metodo restituisce il valore *true*, allora si può invocare *nextInt* senza problemi, altrimenti è possibile, ad esempio, visualizzare un messaggio d'errore e porre fine all'esecuzione del programma.

Per controllare se il dato successivo  
in ingresso è un numero,  
si può invocare *hasNextInt*  
o *hasNextDouble*.

```

if (in.hasNextInt())
{
    int floor = in.nextInt();
    . . . // elabora il dato acquisito
}
else
{
    System.out.println("Error: Not an integer.");
}

```

Infine, vediamo una nuova versione del programma di controllo dell'ascensore, nel quale è stata aggiunta la verifica della validità del dato ricevuto in ingresso.

### File ElevatorSimulation2.java

```

1 import java.util.Scanner;
2
3 /**
4     Simulazione del pannello di controllo di un ascensore per superstiziosi,
5     con verifica della validità del dato fornito in ingresso.
6 */
7 public class ElevatorSimulation2
8 {
9     public static void main(String[] args)
10    {
11        Scanner in = new Scanner(System.in);
12        System.out.print("Floor: ");
13        if (in.hasNextInt())
14        {
15            // ora sappiamo che l'utente ha fornito un numero intero
16
17            int floor = in.nextInt();
18
19            if (floor == 13)
20            {
21                System.out.println("Error: There is no thirteenth floor.");
22            }
23            else if (floor <= 0 || floor > 20)
24            {
25                System.out.println("Error: The floor must be between 1 and 20.");
26            }
27            else
28            {
29                // ora sappiamo che il dato acquisito è valido
30
31                int actualFloor = floor;
32                if (floor > 13)
33                {
34                    actualFloor = floor - 1;
35                }
36
37                System.out.println("The elevator will travel to the actual floor "
38                               + actualFloor);
39            }
40        }
41    }

```

```

42     {
43         System.out.println("Error: Not an integer.");
44     }
45 }
46 }
```

## Esecuzione del programma

Floor: 13  
Error: There is no thirteenth floor.



## Auto-valutazione

38. Cosa visualizza il programma `ElevatorSimulation2.java` quando il dato in ingresso è il seguente?
- 100
  - 1
  - 20
  - thirteen
39. Dovete riscrivere le righe 19–26 del programma `ElevatorSimulation2.java` in modo che venga usato un unico enunciato `if` con una condizione più articolata. Qual è la condizione che serve?
- ```

if ( . . . )
{
    System.out.println("Error: Invalid floor number");
}
```
40. Nel racconto di Sherlock Holmes intitolato “The Adventure of the Sussex Vampire”, l'inimitabile investigatore afferma: “Matilda Briggs non era il nome di una giovane donna, Watson, ... Era una nave che aveva a che fare con il ratto gigante di Sumatra, una storia che il mondo non è ancora preparato ad affrontare”. Oltre un centinaio di anni più tardi, alcuni ricercatori trovarono effettivamente dei ratti giganti nella Nuova Guinea Occidentale, una regione dell'Indonesia, anche se diversa da Sumatra.

Se dovreste scrivere un programma che elabora il peso dei ratti e contiene questi enunciati:

```
System.out.print("Enter weight in kg: ");
double weight = in.nextDouble();
```

quali verifiche dovreste condurre sui dati acquisiti dal programma?

41. Eseguite il seguente programma di collaudo e, quando vi vengono chiesti, fornite come dati in ingresso `2` e `three`. Cosa succede? Perché?

```

import java.util.Scanner;

public class Test
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter an integer: ");
```

```

    int m = in.nextInt();
    System.out.print("Enter another integer: ");
    int n = in.nextInt();
    System.out.println(m + " " + n);
}
}

```

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R5.3, R5.33 e E5.13, al termine del capitolo.



## Computer e società 5.2

### Intelligenza artificiale

Quando si usa un programma informatico sofisticato, come un pacchetto software per la dichiarazione dei redditi, si è inclini ad attribuire al computer una qualche forma di intelligenza. Il computer pone quesiti sensati ed esegue calcoli che per noi sarebbero una sfida. Dopo tutto, se fosse facile preparare una dichiarazione dei redditi, non avremmo bisogno di un computer.

In qualità di programmatore, tuttavia, sappiamo che questa apparente intelligenza è un'illusione. I programmatore umani hanno "addestrato" accuratamente il software perché sappia far fronte a tutte le eventualità possibili, riproducendo banalmente le azioni e le decisioni che sono state programmate al suo interno.

È possibile scrivere programmi informatici che abbiano una qualche forma di vera intelligenza? Fin dagli albori dell'informatica si ebbe la percezione che il cervello umano non fosse null'altro che una sorta di enorme computer e che, pertanto, sarebbe stato possibile programmare i computer affinché imitassero alcuni processi del pensiero umano. A metà degli Anni 50 iniziarono serie ricerche nel campo dell'intelligenza artificiale e i primi vent'anni portarono alcuni incoraggianti successi. I programmi che giocavano a scacchi, sicuramente un'attività che sembra richiedere no-

tevoli abilità intellettuali, divennero talmente capaci che oggi sono in grado di battere abitualmente quasi tutti i migliori giocatori umani. Nel 1975, un programma appartenente alla categoria dei *sistemi esperti*, chiamato Mycin, divenne famoso perché nella diagnosi della meningite era mediamente migliore dei medici.

Fin dai primi esordi, uno degli obiettivi dichiarati della comunità dell'*AI* (Artificial Intelligence) fu quello di produrre software che potesse tradurre testo da una lingua all'altra, per esempio dall'inglese al russo, ma l'impresa si dimostrò estremamente complicata. Il linguaggio umano sembra essere molto più sofisticato e più intrecciato con l'esperienza umana di quanto si pensasse inizialmente. Sistemi come Siri di Apple sono in grado di rispondere a domande ricorrenti, in merito a previsioni meteorologiche, appuntamenti e traffico, ma, al di là di tali limitati ambiti, sono più divertenti che utili.

In alcuni settori, però, abbiamo assistito a rilevanti passi avanti nelle tecnologie guidate dall'intelligenza artificiale. Uno degli esempi più eclatanti riguarda il successo di una serie di "grandi sfide" (*Grand Challenge*) per veicoli autonomi lanciate negli Stati Uniti da DARPA (Defense Advanced Research Projects Agency). I concorrenti devono presentare un veicolo,

controllato da un calcolatore, che possa portare a termine un percorso a ostacoli senza un pilota umano e senza un controllo a distanza. Il primo evento, nel 2004, fu davvero deludente: nessuno dei concorrenti portò a termine il percorso. Nel 2005, cinque veicoli completarono un estenuante percorso di 212 km nel deserto del Mojave: il veicolo Stanley di Stanford University vinse, con una velocità media di 30 km/h. Nel 2007, DARPA spostò la gara in una zona "urbana", un aeroporto militare abbandonato: i veicoli dovevano essere in grado di interagire tra loro, rispettando il codice della strada della California. Oggi i veicoli a guida autonoma sono in fase di collaudo su strade pubbliche in molti stati e si prevede che diventeranno commercialmente disponibili nell'arco di un decennio.

Quando un sistema dotato di intelligenza artificiale sostituisce una persona umana nello svolgimento di un'attività come la diagnosi medica o la guida di veicoli, sorge spontanea una domanda: chi è responsabile in caso di errori? Siamo abituati ad accettare l'idea che medici o guidatori umani compiano, di tanto in tanto, errori con conseguenze mortali, ma siamo pronti a fare la stessa cosa con sistemi esperti che guidano l'automobile o fanno diagnosi mediche?

## Riepilogo degli obiettivi di apprendimento

### L'enunciato if per prendere decisioni

- L'enunciato if consente a un programma di compiere azioni diverse in relazione alla natura dei dati che vengono elaborati.

### Confronti tra numeri e tra oggetti

- Per confrontare numeri usiamo gli operatori relazionali: <, <=, >, >=, ==, !=.
- Gli operatori relazionali confrontano valori; in particolare, l'operatore == verifica un'uguaglianza.
- Confrontando numeri in virgola mobile, non fate verifiche di uguaglianza, ma controllate se i valori sono *sufficientemente prossimi*.
- Per confrontare stringhe non usate l'operatore ==, ma il metodo equals.
- Il metodo compareTo confronta due stringhe secondo il criterio di ordinamento lessicografico.
- L'operatore == verifica se due riferimenti puntano allo stesso oggetto. Per confrontare, invece, i contenuti di oggetti, si deve usare il metodo equals.
- Il riferimento null non fa riferimento ad alcun oggetto.

### Per decisioni complesse servono enunciati if multipli

- Per prendere decisioni articolate si possono combinare più enunciati if.
- Usando enunciati if multipli, occorre verificare prima le condizioni più specifiche, poi quelle più generiche.

### I rami di alcune decisioni possono richiedere ulteriori decisioni

- Quando un enunciato di decisione è contenuto nel ramo di un altro enunciato di decisione, si dice che i due enunciati sono *annidati*.
- Le decisioni annidate servono per risolvere problemi che richiedono decisioni *su più livelli*.

### Con i diagrammi di flusso si visualizza il flusso d'esecuzione

- I diagrammi di flusso (*flowchart*) sono costituiti da elementi che rappresentano compiti da svolgere, acquisizione e visualizzazione di dati (input/output) e decisioni da prendere.
- Ogni ramo di una decisione può contenere compiti da svolgere e altre decisioni da prendere.
- Non far mai entrare una freccia all'interno di una diversa diramazione.

### Per ogni programma bisogna preparare casi di prova

- Il collaudo a scatola chiusa (*black-box testing*) descrive una metodologia di collaudo che non prende in considerazione la struttura dell'implementazione.
- Il collaudo trasparente (*white-box testing*) usa informazioni sulla struttura del programma.
- La copertura (*coverage*) di un collaudo misura quante parti di un programma sono state collaudate.
- I casi limite sono casi di prova che usano valori limite dei dati in ingresso.
- È bene progettare i casi di prova prima di scrivere il codice di un programma.
- I messaggi di tracciamento possono essere disattivati dopo aver completato il collaudo.

### Il tipo di dato boolean memorizza il risultato di condizioni: vere o false

- Il tipo booleano, boolean, ha due valori: false e true.
- Java ha due operatori booleani che combinano condizioni: && (*and*) e || (*or*).
- Per invertire una condizione si usa l'operatore ! (*not*).
- Gli operatori && e || vengono valutati mediante *cortocircuito*: non appena viene determinato il risultato, non viene valutata alcuna ulteriore condizione.

- La legge di De Morgan indica come semplificare espressioni in cui l'operatore *not* sia applicato a espressioni *and* o *or*.

**Con enunciati if si può decidere se i dati acquisiti sono validi**

- Per controllare se il dato successivo in ingresso è un numero, si può invocare `hasNextInt` o `hasNextDouble`.

## Elementi di libreria presentati in questo capitolo

```
java.awt.Rectangle
    equals
java.lang.String
    equals
    compareTo
java.util.Scanner
    hasNextDouble
    hasNextInt
```

```
java.util.logging.Level
    INFO
    OFF
java.util.logging.Logger
    getGlobal
    info
    setLevel
```

## Esercizi di riepilogo e approfondimento

- ★ **R5.1.** Qual è il valore di ciascuna variabile dopo l'esecuzione dell'enunciato if?

- int n = 1; int k = 2; int r = n;  
if (k < n) { r = k; }
- int n = 1; int k = 2; int r;  
if (n < k) { r = k; }  
else { r = k + n; }
- int n = 1; int k = 2; int r = k;  
if (r < k) { n = r; }  
else { k = n; }
- int n = 1; int k = 2; int r = 3;  
if (r < n + k) { r = 2 \* n; }  
else { k = 2 \* r; }

- ★★ **R5.2.** Spiegate la differenza tra:

```
s = 0
if (x > 0) { s++; }
if (y > 0) { s++; }
```

e:

```
s = 0
if (x > 0) { s++; }
else if (y > 0) { s++; }
```

- ★★ **R5.3.** Trovate gli errori nei seguenti enunciati if:

- if x > 0 then System.out.print(x);
- if (1 + x > Math.pow(x, Math.sqrt(2))) { y = y + x; }
- if (x = 1) { y++; }
- x = in.nextInt();
 if (in.hasNextInt())

```
{  
    sum = sum + x;  
}  
else  
{  
    System.out.println("Bad input for x");  
}  
e. String letterGrade = "F";  
if (grade >= 90) { letterGrade = "A"; }  
if (grade >= 80) { letterGrade = "B"; }  
if (grade >= 70) { letterGrade = "C"; }  
if (grade >= 60) { letterGrade = "D"; }
```

\* **R5.4.** Cosa visualizzano questi frammenti di codice?

- a. int n = 1;  
int m = -1;  
if (n < -m) { System.out.print(n); }  
else { System.out.print(m); }
- b. int n = 1;  
int m = -1;  
if (-n >= m) { System.out.print(n); }  
else { System.out.print(m); }
- c. double x = 0;  
double y = 1;  
if (Math.abs(x - y) < 1) { System.out.print(x); }  
else { System.out.print(y); }
- d. double x = Math.sqrt(2);  
double y = 2;  
if (x \* x == y) { System.out.print(x); }  
else { System.out.print(y); }

\*\* **R5.5.** Nell'ipotesi che x e y siano variabili di tipo double, scrivete un frammento di codice che assegni a y il valore di x se questo è positivo, altrimenti 0.

\*\* **R5.6.** Nell'ipotesi che x e y siano variabili di tipo double, scrivete un frammento di codice che assegni a y il valore assoluto di x senza invocare la funzione Math.abs. Usate un enunciato if.

\*\* **R5.7.** Spiegate perché è più difficile confrontare numeri in virgola mobile piuttosto che numeri interi. Scrivete un frammento di codice che verifichi se una variabile intera n è uguale a 10 e un altro che verifichi se una variabile in virgola mobile x è circa uguale a 10.

\*\* **R5.8.** Dati due pixel sullo schermo aventi come coordinate  $(x_1, y_1)$  e  $(x_2, y_2)$  (tutti numeri interi), scrivete frammenti di codice che verifichino se:

- a. Si tratta dello stesso pixel.  
b. Sono due pixel molto vicini, con distanza minore di 5.

\* **R5.9.** È facile confondere gli operatori = e ==. Scrivete un programma di prova che contenga l'enunciato:

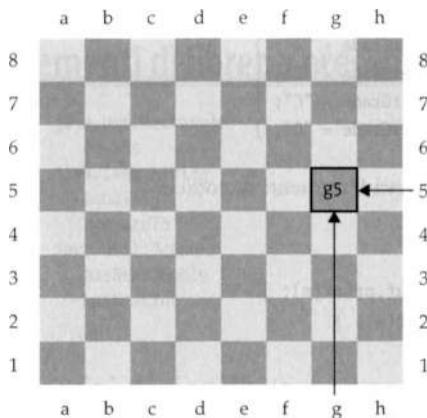
```
if (floor = 13)
```

Quale messaggio d'errore viene visualizzato? Scrivete un altro programma di prova che contenga l'enunciato:

```
count == 0;
```

Cosa fa il compilatore quando si compila il programma?

- \*\* R5.10. Ogni casella di una scacchiera può essere individuata da una lettera e un numero, come, in questo esempio, la casella g5:



Lo pseudocodice seguente descrive un algoritmo che determina se una casella è nera o bianca, dato il suo identificativo (lettera e numero).

Se la lettera è a, c, e oppure g

Se il numero è dispari

colore = "nero"

Altrimenti

colore = "bianco"

Altrimenti

Se il numero è pari

colore = "nero"

Altrimenti

colore = "bianco"

Usando la procedura delineata nella sezione Suggerimenti per la programmazione 5.5, seguite passo dopo passo l'esecuzione del codice nel caso g5.

- \*\* R5.11 (collaudo). Progettate un insieme di quattro casi di prova per l'algoritmo dell'esercizio precedente, in modo da coprire tutte le diramazioni.
- \*\* R5.12. In un programma di calendarizzazione (*scheduling*) di eventi si deve verificare se due appuntamenti si sovrappongono. Per semplicità, ipotizziamo che gli appuntamenti inizino sempre a un'ora esatta (senza minuti) e usiamo l'orario militare (cioè con le ore che vanno da 0 a 23). Lo pseudocodice seguente descrive un algoritmo che determina se l'appuntamento che inizia all'ora *start1* e termina all'ora *end1* si sovrappone all'appuntamento che inizia all'ora *start2* e termina all'ora *end2*.

Se *start1* > *start2*

*s* = *start1*

Altrimenti

*s* = *start2*

```
Se end1 < end2
    e = end1
Altrimenti
    e = end2
Se s < e
    Gli appuntamenti si sovrappongono.
Altrimenti
    Gli appuntamenti non si sovrappongono.
```

Seguite passo dopo passo l'esecuzione del codice con gli appuntamenti 10-12 e 11-13, poi con gli appuntamenti 10-11 e 12-13.

- \* **R5.13.** Disegnate il diagramma di flusso per l'algoritmo dell'esercizio precedente.
- \* **R5.14.** Disegnate il diagramma di flusso per l'algoritmo dell'Esercizio E5.14.
- \* **R5.15.** Disegnate il diagramma di flusso per l'algoritmo dell'Esercizio E5.15.
- \*\* **R5.16 (collaudo).** Progettate un insieme di casi di prova per l'algoritmo dell'Esercizio R5.12.
- \*\* **R5.17 (collaudo).** Progettate un insieme di casi di prova per l'algoritmo dell'Esercizio E5.15.
- \*\* **R5.18.** Scrivete lo pseudocodice per un programma che chiede all'utente un mese e un giorno e visualizza un messaggio opportuno se si tratta di una di queste festività statunitensi:
  - New Year's Day (1 gennaio)
  - Independence Day (4 luglio)
  - Veterans Day (11 novembre)
  - Christmas Day (25 dicembre)
- \*\* **R5.19.** Scrivete lo pseudocodice per un programma che assegna un voto in lettere al risultato di un questionario, secondo la tabella seguente:

Punteggio	Voto
90-100	A
80-89	B
70-79	C
60-69	D
< 60	F

- \*\* **R5.20.** Illustrate le differenze tra l'ordinamento lessicografico delle stringhe in Java e l'ordinamento delle parole in un dizionario o in un elenco telefonico. Suggerimento: considerate stringhe come IBM, wiley.com, Century 21, While-U-Wait.
- \*\* **R5.21.** In ciascuna delle seguenti coppie, quale stringa precede l'altra nell'ordinamento lessicografico?
  - a. "Tom", "Jerry"
  - b. "Tom", "Tomato"
  - c. "church", "Churchill"
  - d. "car manufacturer", "carburetor"
  - e. "Harry", "hairy"
  - f. "Java", "Car"
  - g. "Tom", "Tom"
  - h. "Car", "Carl"
  - i. "car", "bar"

- \* **R5.22.** Illustrate le differenze esistenti tra enunciati `if` annidati e una sequenza `if/else if/else`, fornendo esempi di entrambi i casi.
- \*\* **R5.23.** Fornite un esempio di una sequenza `if/else if/else` in cui l'ordine delle verifiche è ininfluente, e un altro esempio in cui, invece, l'ordine è influente.
- \* **R5.24.** Riscrivete la condizione vista nel Paragrafo 5.3 in modo che usi operatori `<` al posto di operatori  `$\geq$` . Come va modificato l'ordine dei confronti?
- \*\* **R5.25 (collaudo).** Progettate un insieme di casi di prova per il programma del calcolo delle tasse presentato nell'Esercizio P5.2 e calcolate a mano i valori previsti.
- \* **R5.26.** Scrivete un frammento di codice Java che illustri il problema dell'`else` sospeso usando questa frase: uno studente con voto complessivo (nel sistema anglosassone *GPA*, *grade point average*) inferiore a 2 ma non inferiore a 1.5 è “rimandato”, uno con *GPA* inferiore a 1.5 è bocciato.
- \*\*\* **R5.27.** Completate la tabella della verità seguente, calcolando i valori assunti dalle espressioni booleane indicate per tutte le combinazioni dei valori di ingresso `p`, `q` e `r`.

<code>p</code>	<code>q</code>	<code>r</code>	<code>(p &amp;&amp; q)    !r</code>	<code>!(p &amp;&amp; (q    !r))</code>
falso	falso	falso		
falso	falso	vero		
falso	vero	falso		
...				
altre 5 combinazioni				
...				

- \*\*\* **R5.28.** È vero o è falso che  $A \&\& B$  assume lo stesso valore di  $B \&\& A$  per qualsiasi espressione booleana  $A$  e  $B$ ?
- \* **R5.29.** La funzione di “ricerca avanzata” di molti motori di ricerca vi consente di utilizzare operatori booleani per comporre interrogazioni complesse, come “(cats OR dogs) AND NOT pets”. Mettete a confronto questi operatori di ricerca con gli operatori booleani di Java.
- \*\* **R5.30.** Se il valore di `b` è `false` e il valore di `x` è 0, qual è il valore di ciascuna delle seguenti espressioni?
  - `b && x == 0`
  - `b || x == 0`
  - `!b && x == 0`
  - `!b || x == 0`
  - `b && x != 0`
  - `b || x != 0`
  - `!b && x != 0`
  - `!b || x != 0`
- \*\* **R5.31.** Semplificate le espressioni seguenti, dove `b` è una variabile di tipo `boolean`.
  - `b == true`
  - `b == false`
  - `b != true`
  - `b != false`

- \*\*\* R5.32. Semplificate gli enunciati seguenti, dove *b* è una variabile di tipo boolean e *n* è una variabile di tipo int.

a. if (*n* == 0) { *b* = true; } else { *b* = false; }

*Suggerimento:* qual è il valore di *n* == 0?

- b. if (*n* == 0) { *b* = false; } else { *b* = true; }  
 c. *b* = false; if (*n* > 1) { if (*n* < 2) { *b* = true; } }  
 d. if (*n* < 1) { *b* = true; } else { *b* = *n* > 2; }

- \* R5.33. Cosa c'è di sbagliato nel programma seguente?

```
System.out.print("Enter the number of quarters: ");
int quarters = in.nextInt();
if (in.hasNextInt())
{
    total = total + quarters * 0.25;
    System.out.println("Total: " + total);
}
else
{
    System.out.println("Input error.");
}
```

## Esercizi di programmazione

- \* E5.1. Scrivete un programma che acquisisca un numero intero e visualizzi un messaggio diverso nei casi in cui il numero acquisito sia negativo, uguale a zero o positivo.
- \*\* E5.2. Scrivete un programma che acquisisca un numero in virgola mobile e visualizzi il messaggio “zero” se è uguale a zero, altrimenti visualizzi il messaggio “positive” (se è positivo) o “negative” (se è negativo). Aggiungete il messaggio “small” se il valore assoluto del numero è inferiore a 1 oppure “large” se è superiore a un milione.
- \*\* E5.3. Scrivete un programma che legga un numero intero e visualizzi il numero delle sue cifre, verificando prima se il numero è  $\geq 10$ , poi se è  $\geq 100$  e così via (ipotizzando che tutti i possibili numeri interi siano inferiori a dieci miliardi). Se il numero è negativo, moltiplicatelo prima per -1.
- \*\* E5.4. Scrivete un programma che legga tre numeri e visualizzi il messaggio “all the same” se sono tutti uguali, “all different” se sono tutti diversi e “neither” se non sono tutti uguali e non sono tutti diversi.
- \*\* E5.5. Scrivete un programma che legga tre numeri e visualizzi il messaggio “increasing” se sono in ordine crescente, “decreasing” se sono in ordine decrescente e “neither” se non sono né in ordine crescente né in ordine decrescente. In questo esercizio *crescente* significa *strettamente crescente*, cioè ciascun valore deve essere maggiore del precedente (analogo significato ha il termine *decrescente*): la sequenza 3 4 4, quindi, non va considerata crescente.
- \*\* E5.6. Risolvete nuovamente l'esercizio precedente ma, prima di leggere i numeri, chiedete all'utente se l'ordine crescente/decrescente va verificato in senso “stretto” oppure no. Nel secondo caso, ovviamente, la sequenza 3 4 4 è considerata crescente, così come la sequenza 4 4 4, che è anche decrescente.

- \*\* E5.7. Scrivete un programma che legga tre numeri interi e visualizzi il messaggio “in order” se sono ordinati in senso crescente o decrescente e “not in order” in caso contrario. Ad esempio:

```
1 2 5    in order
1 5 2    not in order
5 2 1    in order
1 2 2    in order
```

- \*\* E5.8. Scrivete un programma che legga quattro numeri interi e visualizzi il messaggio “two pairs” se i dati acquisiti costituiscono due coppie (in qualsiasi ordine) e “not two pairs” in caso contrario. Ad esempio:

```
1 2 2 1  two pairs
1 2 2 3  not two pairs
2 2 2 2  two pairs
```

- \*\* E5.9. L’ago di una bussola indica il numero di gradi di deviazione rispetto al Nord, misurati in senso orario. Scrivete un programma che legga il valore dell’angolo e visualizzi la direzione più vicina, scelta tra una di quelle riportate solitamente nelle bussole: N, NE, E, SE, S, SW, W, NW (dove le lettere N, E, S e W indicano i punti cardinali in senso orario: nord, est, sud, ovest). In caso di equidistanza, va preferita una delle direzioni principali (N, E, S o W).

- \*\* E5.10 (economia). Scrivete un programma che legga il nome di un dipendente e la sua paga oraria, ad esempio \$9.25. Poi, deve chiedere il numero di ore lavorate dal dipendente nella settimana precedente, accettando anche un numero frazionario. Infine, deve calcolare il pagamento dovuto e visualizzare la busta paga del dipendente, tenendo presente che le ore di straordinario (cioè quelle che eccedono le 40 ore) hanno una paga oraria pari al 150 per cento di quella normale. Progettate la classe Paycheck (*busta paga*).

- \* E5.11. Scrivete un programma che legga un valore di temperatura seguito dalla lettera C per Celsius o F per Fahrenheit, poi visualizzi un messaggio che dica se, al livello del mare e a quella temperatura, l’acqua si trova allo stato solido, liquido o gassoso.

- \* E5.12. Il punto di ebollizione dell’acqua diminuisce di circa un grado Celsius ogni 300 metri (o 1000 piedi) di altitudine. Migliorate il programma dell’esercizio precedente, consentendo all’utente di specificare l’altitudine, in metri o in piedi.

- \* E5.13. Aggiungete all’esercizio precedente la gestione degli errori: se l’utente non digita un numero quando sarebbe previsto che lo facesse, oppure fornisce un’unità di misura non valida per l’altitudine, visualizzate un messaggio d’errore e terminate il programma.

- \*\* E5.14. Confrontando due istanti di tempo, ciascuno espresso come ora e minuti (usando la consuetudine militare, cioè con le ore che vanno da 0 a 23), il codice seguente determina quale istante preceda l’altro:

```
Se ora1 < ora2
  istante1 è il primo.
Altrimenti se ora1 è uguale a ora2
  Se minuto1 < minuto2
    istante1 è il primo.
  Altrimenti se minuto1 è uguale a minuto2
    istante1 e istante2 sono identici.
  Altrimenti
    istante2 è il primo.
Altrimenti
  istante2 è il primo.
```

Scrivete un programma che chieda all'utente di fornire due istanti di tempo e li visualizzi in ordine temporale, visualizzando cioè per primo quello che precede l'altro. Progettate la classe `Time` dotata del metodo

```
public int compareTo(Time other)
```

che restituisca -1 se l'istante di tempo rappresentato dal parametro implicito precede quello rappresentato dal parametro esplicito `other` e 1 nel caso opposto, restituendo 0 se i due istanti sono identici.

- \*\* E5.15. L'algoritmo seguente individua la stagione (Spring, Summer, Fall o Winter, cioè, rispettivamente, primavera, estate, autunno o inverno) a cui appartiene una data, fornita come mese e giorno, due numeri interi.

Se mese è 1, 2 o 3, stagione = "Winter"

Altrimenti se mese è 4, 5 o 6, stagione = "Spring"

Altrimenti se mese è 7, 8 o 9, stagione = "Summer"

Altrimenti se mese è 10, 11 o 12, stagione = "Fall"

Se mese è divisibile per 3 e giorno >= 21

    Se stagione è "Winter", stagione = "Spring"

    Altrimenti se stagione è "Spring", stagione = "Summer"

    Altrimenti se stagione è "Summer", stagione = "Fall"

    Altrimenti stagione = "Winter"

Scrivete un programma che chieda all'utente un mese e un giorno e, poi, visualizzi la stagione determinata da questo algoritmo. Progettate la classe `Date` dotata del metodo `getSeason`.

- \*\* E5.16. Scrivete un programma che traduca un voto espresso in lettere nel corrispondente voto numerico. I voti in lettere sono A, B, C, D e F, eventualmente seguiti da un segno + o -. I loro valori numerici sono, nell'ordine, 4, 3, 2, 1 e 0. I voti F+ e F- non esistono. Un segno + aumenta il voto numerico di 0.3, mentre un segno - lo diminuisce della stessa quantità. Il voto A+ è comunque uguale a 4.0.

Enter a letter grade: B-

The numeric value is 2.7.

Progettate la classe `Grade` dotata del metodo `getNumericGrade`.

- \*\* E5.17. Scrivete un programma che traduca un numero compreso tra 0 e 4 nel voto letterale più vicino, usando le regole definite nell'esercizio precedente. Ad esempio, il numero 2.8 (che potrebbe essere la media di più voti) deve essere convertito in B-. Risolvete i casi di parità in favore del voto migliore: ad esempio, 2.85 deve essere convertito in B. Progettate la classe `Grade` dotata del metodo `getLetterGrade`.

- \*\* E5.18. Nel 1913 lo schema di tassazione statunitense era abbastanza semplice. L'importo da pagare era:

- 1% sulla quota di reddito inferiore a \$50000
- 2% sulla quota di reddito compresa tra \$50000 e \$75000
- 3% sulla quota di reddito compresa tra \$75000 e \$100000
- 4% sulla quota di reddito compresa tra \$100000 e \$250000
- 5% sulla quota di reddito compresa tra \$250000 e \$500000
- 6% sulla quota di reddito superiore a \$500000

Non esistevano regimi di tassazione distinti per persone coniugate e per persone non coniugate. Scrivete un programma che calcoli l'importo della tassazione sul reddito secondo questo schema.

- \*\* E5.19. Scrivete un programma che acquisisca dall'utente la descrizione di una carta da gioco, usando la seguente notazione abbreviata:

A	Ace ( <i>asso</i> )
2 . . . 10	Valore della carta
J	Jack ( <i>fante</i> )
Q	Queen ( <i>donna</i> )
K	King ( <i>re</i> )
D	Diamonds ( <i>quadrì</i> )
H	Hearts ( <i>cuori</i> )
S	Spades ( <i>picche</i> )
C	Clubs ( <i>fiori</i> )

Poi, il programma deve visualizzare la descrizione completa della carta. Ad esempio:

```
Enter the card notation: QS
Queen of Spades
```

Progettate la classe `Card` il cui costruttore riceva come parametro la stringa che descrive la carta e il cui metodo `getDescription` restituisca una stringa che descrive la carta, come specificato. Se la stringa fornita come parametro di costruzione non rispetta il formato richiesto, il metodo `getDescription` deve restituire la stringa "Unknown".

- \*\* E5.20. Scrivete un programma che legga tre numeri in virgola mobile e visualizzi il maggiore di essi. Ad esempio:

```
Please enter three numbers: 4 9 2.5
The largest number is 9.
```

- \*\* E5.21. Scrivete un programma che legga tre stringhe e le visualizzi in ordine lessicografico. Ad esempio:

```
Enter three strings: Charlie Able Baker
Able
Baker
Charlie
```

- \*\* E5.22. Scrivete un programma che legga due numeri in virgola mobile e verifichi se sono uguali quando vengono arrotondati alla seconda cifra decimale. Ecco due esempi di esecuzione:

```
Enter two floating-point numbers: 2.0 1.99998
They are the same up to two decimal places.
Enter two floating-point numbers: 2.0 1.98999
They are different.
```

- \* E5.23. Scrivete un programma che chieda all'utente di fornire una singola lettera dell'alfabeto, visualizzando poi il messaggio "Vowel" se si tratta di una vocale e "Consonant" se si tratta di una consonante. Se l'utente non digita una lettera (cioè un carattere compreso tra "a" e "z" oppure tra "A" e "Z") oppure digita una stringa di lunghezza maggiore di uno, visualizzate un messaggio d'errore.

- \*\* E5.24. Scrivete un programma che chieda all'utente di digitare il numero identificativo di un mese (1 per gennaio, 2 per febbraio, e così via), per poi visualizzare il numero di giorni da cui è composto il mese selezionato. Nel caso di febbraio, visualizzate il messaggio "28 days".

```
Enter a month: 5  
30 days
```

Progettate la classe Month dotata del metodo:

```
public int getLength()
```

Non usate un diverso ramo if/else per ciascun mese, ma componete le condizioni usando gli opportuni operatori booleani.

- \* **E5.25 (economia).** Un supermercato premia i propri clienti con buoni spesa il cui importo dipende dalla quantità di denaro speso in prodotti alimentari (*groceries*). Ad esempio, spendendo 50 dollari si ottiene un buono spesa di importo pari all'otto per cento di quella somma. La tabella seguente mostra la percentuale usata per calcolare il buono spesa relativo a somme diverse. Scrivete un programma che calcoli e visualizzi il valore del buono da consegnare al cliente, sulla base della somma di denaro che ha speso nell'acquisto di prodotti alimentari.

Denaro speso	Percentuale del buono
Meno di \$10	Nessun buono
Da \$10 a \$60	8%
Da più di \$60 a \$150	10%
Da più di \$150 a \$210	12%
Più di \$210	14%

Ecco un esempio di esecuzione del programma:

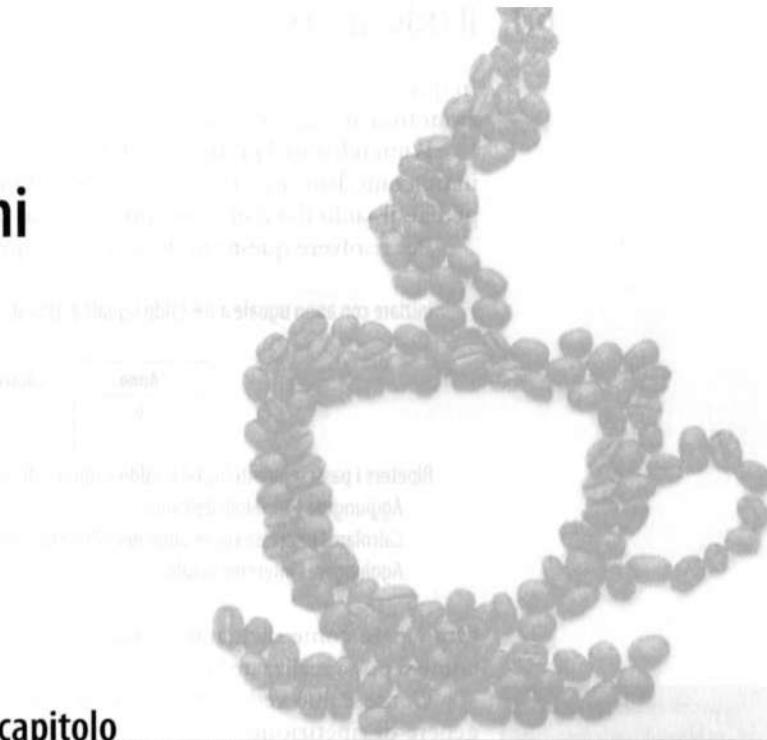
```
Please enter the cost of your groceries: 14  
You win a discount coupon of $1.12 (8% of your purchase).
```

Sul sito web dedicato al libro si trova una raccolta di progetti di programmazione più complessi.



# 6

## Iterazioni



### Obiettivi del capitolo

- Saper programmare cicli con gli enunciati `while`, `for` e `do`
- Riprodurre su carta l'esecuzione di un programma
- Apprendere gli algoritmi più diffusi che richiedono cicli
- Comprendere il funzionamento dei cicli annidati
- Realizzare programmi che acquisiscono ed elaborano insiemi di dati
- Usare un calcolatore per effettuare simulazioni
- Conoscere il debugger

In un ciclo, una porzione di un programma viene eseguita ripetutamente, fino al raggiungimento di uno specifico obiettivo. I cicli sono assai importanti per realizzare quelle elaborazioni che richiedono passi ripetuti e per l'elaborazione di dati in ingresso costituiti da molti valori. In questo capitolo imparerete gli enunciati di ciclo presenti in Java, oltre alle tecniche usate per scrivere programmi che ricevono dati in ingresso e simulano attività del mondo reale.

## 6.1 Il ciclo while

In questo paragrafo vedrete come usare *enunciati di ciclo* per eseguire ripetutamente istruzioni fino al raggiungimento di un obiettivo.

Riprendiamo il problema dell'investimento visto nel Capitolo 1. Si depositano \$10000 in un conto bancario, remunerato con un interesse annuo del 5%: quanti anni occorrono perché il saldo del conto diventi uguale al doppio dell'investimento iniziale?

Per risolvere questo problema, nel Capitolo 1 abbiamo progettato questo algoritmo:

Iniziare con anno uguale a 0 e saldo uguale a 10000.

Anno	Interesse	Saldo
0		10 000

Ripetere i passi seguenti finché il saldo è minore di 20000

Aggiungere 1 al valore dell'anno.

Calcolare l'interesse come saldo per 0.05 (cioè il 5%).

Aggiungere l'interesse al saldo.

Ora sapete come dichiarare e aggiornare variabili in Java. Ciò che ancora non sapete è come si possa realizzare la frase “ripetere i passi seguenti finché il saldo è minore di 20000”.

Come si può vedere nella sezione Sintassi di Java 6.1, l'enunciato `while` realizza questo genere di ripetizione:

Un ciclo esegue ripetutamente  
un blocco di codice fintantoché  
una specifica condizione risulta vera.

```
while (condizione)
{
    enunciati
}
```

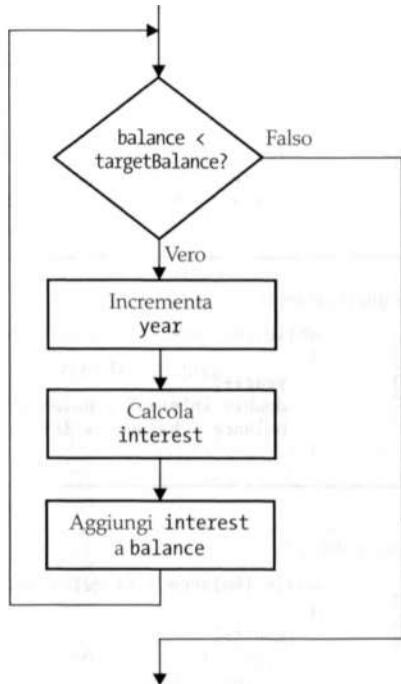
Finché la *condizione* è vera, gli *enunciati* presenti all'interno dell'enunciato `while` vengono eseguiti ripetutamente: costituiscono il cosiddetto **corpo** (*body*) dell'enunciato `while`.

Nel nostro caso vogliamo incrementare il contatore degli anni e, conseguentemente, aggiungere interessi finché il saldo è minore dell'obiettivo da raggiungere, che è un saldo di \$20000:

```
while (balance < targetBalance)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

L'enunciato `while` è un esempio di **ciclo** (*loop*). Se disegnate un diagramma di flusso, vedrete che il controllo torna ciclicamente alla verifica iniziale della condizioni (come nella Figura 1).

**Figura 1**  
Diagramma di flusso  
di un ciclo while



## Sintassi di Java

### 6.1 L'enunciato while

#### Sintassi

```
while (condizione)
{
  enunciati
}
```

#### Esempio

Questa variabile è dichiarata all'esterno del ciclo  
e viene aggiornata al suo interno.

Se la condizione non  
diventa mai falsa, si  
ha un ciclo infinito.

Questa variabile viene creata  
a ogni iterazione del ciclo.

double balance = 0;

...

while (balance < targetBalance)

{  
 double interest = balance \* RATE / 100;  
 balance = balance + interest;  
}

Non mettere un punto  
e virgola qui!

È utile incolonnare  
le parentesi graffe.

Se il corpo contiene un solo enunciato,  
le parentesi graffe non sono necessarie,  
ma è bene inserirle ugualmente.

Questi enunciati vengono  
eseguiti finché la condizione  
è vera.

**Figura 2**

Esecuzione  
di un ciclo while

- ① Verifica della condizione del ciclo

balance =   
year =

```
while (balance < targetBalance)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

La condizione  
è vera

- ② Esecuzione degli enunciati interni al ciclo

balance =   
year =   
interest =

```
while (balance < targetBalance)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

- ③ Nuova verifica della condizione del ciclo

balance =   
year =

```
while (balance < targetBalance)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

La condizione  
è ancora vera

:

- ④ Dopo 15 iterazioni

balance =   
year =

```
while (balance < targetBalance)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

La condizione  
non è più vera

- ⑤ Esecuzione dell'enunciato che segue il ciclo

balance =   
year =

```
while (balance < targetBalance)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
System.out.println(year);
```

Quando una variabile viene dichiarata *all'interno* del corpo del ciclo, ad ogni nuova iterazione del ciclo la variabile viene creata all'inizio ed eliminata alla fine. Ad esempio, osservate la variabile `interest` in questo ciclo:

```
while (balance < targetBalance)
{
```

```
year++;
// ad ogni iterazione viene creata una nuova variabile interest
double interest = balance * RATE / 100;
balance = balance + interest;
}
// qui balance non esiste più
```

Al contrario, le variabili `balance` e `year` sono state dichiarate al di fuori del corpo del ciclo: in questo modo, per tutte le iterazioni del ciclo viene usata la stessa variabile.

Come esempio completo, ecco il programma che risolve il problema del raddoppio di un investimento, il cui funzionamento è illustrato nella Figura 2.

### File Investment.java

```
1 /**
2  * Classe che gestisce l'aumento di un investimento
3  * che accumula interessi annualmente con tasso fisso.
4 */
5 public class Investment
6 {
7     private double balance;
8     private double rate;
9     private int year;
10
11    /**
12     * Costruisce un oggetto che rappresenta un investimento
13     * con un dato saldo iniziale e un dato tasso di interesse.
14     * @param aBalance il saldo iniziale
15     * @param aRate il tasso di interesse in percentuale
16    */
17    public Investment(double aBalance, double aRate)
18    {
19        balance = aBalance;
20        rate = aRate;
21        year = 0;
22    }
23
24    /**
25     * Continua ad accumulare interessi fino al raggiungimento
26     * del saldo desiderato.
27     * @param targetBalance il saldo desiderato
28    */
29    public void waitForBalance(double targetBalance)
30    {
31        while (balance < targetBalance)
32        {
33            year++;
34            double interest = balance * rate / 100;
35            balance = balance + interest;
36        }
37    }
38
39    /**
40     * Ispeziona il saldo attuale dell'investimento.
41     * @return il saldo attuale
```

```

42     */
43     public double getBalance()
44     {
45         return balance;
46     }
47
48 /**
49     Ispeziona il numero di anni durante i quali l'investimento
50     ha accumulato interessi.
51     @return il numero di anni dall'inizio dell'investimento
52 */
53     public int getYears()
54     {
55         return year;
56     }
57 }
```

### File InvestmentRunner.java

```

1 /**
2     Questo programma calcola il tempo necessario
3     per il raddoppio di un investimento.
4 */
5 public class InvestmentRunner
6 {
7     public static void main(String[] args)
8     {
9         final double INITIAL_BALANCE = 10000;
10        final double RATE = 5;
11        Investment invest = new Investment(INITIAL_BALANCE, RATE);
12        invest.waitForBalance(2 * INITIAL_BALANCE);
13        int years = invest.getYears();
14        System.out.println("The investment doubled after "
15                           + years + " years");
16    }
17 }
```

### Esecuzione del programma

The investment doubled after 15 years.



### Auto-valutazione

- Quanti anni servono per triplicare l'investimento? Modificate il programma ed eseguitelo.
- Se il tasso di interesse fosse il 10%, quanti anni servirebbero per raddoppiare l'investimento? Modificate il programma ed eseguitelo.
- Modificate il programma in modo che il saldo venga visualizzato ogni anno. Come si fa?
- Immaginate di modificare il programma in modo che la condizione del ciclo `while` diventi questa:  
`while (balance <= targetBalance)`  
 Che effetto ha sul programma? Perché?
- Cosa visualizza questo ciclo?  
`int n = 1;`  
`while (n < 100)`

```
{
    n = 2 * n;
    System.out.print(n + " ");
}
```

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R6.4, R6.8 e E6.14, al termine del capitolo.

**Tabella 1**

Esempi con cicli while

Ciclo	Visualizza	Spiegazione
<pre>i = 0; sum = 0; while (sum &lt; 10) {     i++; sum = sum + i;     Stampa i e sum; }</pre>	1 1 2 3 3 6 4 10	Quando <code>sum</code> diventa uguale a 10, la condizione del ciclo diventa falsa e il ciclo termina.
<pre>i = 0; sum = 0; while (sum &lt; 10) {     i++; sum = sum - i;     Stampa i e sum; }</pre>	1 -1 2 -3 3 -6 4 -10 . . .	Dato che <code>sum</code> non raggiunge mai 10, si ha un ciclo infinito (Errori comuni 6.2).
<pre>i = 0; sum = 0; while (sum &lt; 0) {     i++; sum = sum - i;     Stampa i e sum; }</pre>	(Niente)	La condizione <code>sum &lt; 0</code> è già falsa alla prima verifica, per cui il ciclo non viene mai eseguito.
<pre>i = 0; sum = 0; while (sum &gt;= 10) {     i++; sum = sum + i;     Stampa i e sum; }</pre>	(Niente)	Probabilmente il programmatore voleva scrivere “fermati quando la somma vale almeno 10”, però la condizione presente nel ciclo dice quando il corpo va eseguito, non quando il ciclo deve terminare (Errori comuni 6.1).
<pre>i = 0; sum = 0; while (sum &lt; 10) ; {     i++; sum = sum + i;     Stampa i e sum; }</pre>	(Niente, il programma non termina)	Notate il punto e virgola prima della parentesi graffa aperta: questo ciclo ha un corpo vuoto e viene eseguito indefinitamente, continuando a controllare che <code>sum</code> sia minore di 10, senza fare nulla altro.

### Errori comuni 6.1

#### Non pensate “Abbiamo finito?”

Quando facciamo qualcosa di ripetitivo, spesso ci chiediamo se abbiamo finito. Nel caso del raddoppio dell’investimento, ad esempio, potremmo pensare “Voglio arrivare almeno a \$20000”, scrivendo la condizione del ciclo in questo modo:

```
balance >= balanceTarget
```

Ma il ciclo `while` ragiona in modo opposto: fino a quando posso proseguire? La condizione corretta per il ciclo è, quindi:

```
while (balance < balanceTarget)
```

In altre parole: "Continua finché il saldo è minore dell'obiettivo".



## Errori comuni 6.2

### Cicli infiniti

Nel progetto di cicli, un errore molto frequente è il *ciclo infinito*: un ciclo che viene eseguito indefinitamente e può essere arrestato soltanto terminando l'esecuzione del programma tramite il sistema operativo oppure, addirittura, riavviando il computer. Se il ciclo contiene enunciati di visualizzazione, vedremo molte righe di testo scorrere velocemente sullo schermo, altrimenti il programma continua silenziosamente la propria esecuzione e appare *bloccato*, apparentemente senza fare nulla. In alcuni sistemi operativi si può terminare un programma che si trova in questo stato premendo la combinazione di tasti Ctrl-C, in altri si può chiudere la finestra in cui si sta eseguendo il programma.

Molto spesso si innesca un ciclo infinito perché ci si dimentica di aggiornare la variabile che lo controlla:

```
int year = 1;
while (year <= 20)
{
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

Qui il programmatore ha dimenticato di aggiungere all'interno del ciclo l'istruzione `year++`. Di conseguenza, il valore dell'anno rimane sempre 1 e il ciclo non arriva mai a conclusione.

Altrettanto spesso si realizza un ciclo infinito perché, sbadatamente, si incrementa un contatore che dovrebbe essere decrementato (o viceversa), come in questo esempio:

```
int year = 20;
while (year > 0)
{
    double interest = balance * RATE / 100;
    balance = balance + interest;
    year++;
}
```

La variabile `year` avrebbe dovuto essere decrementata, non incrementata: questo errore è molto frequente perché capita più spesso di dover incrementare i contatori, piuttosto che decrementarli, per cui le nostre dita scrivono inconsapevolmente `++`. Di conseguenza, `year` è sempre maggiore di zero e il ciclo non termina mai (in realtà, prima o poi il valore di `year` supererà il massimo numero intero rappresentabile e diventerà un numero negativo,

per effetto della modalità di rappresentazione dei numeri interi, facendo così terminare il ciclo, anche se con un risultato completamente errato).



## Errori comuni 6.3

### Errori per scarto di uno

Considerate il nostro calcolo del numero di anni necessari per raddoppiare un investimento:

```
int year = 0;
while (balance < targetBalance)
{
    year++;
    balance = balance * (1 + RATE / 100);
}
System.out.println("The investment doubled after ",
+ year + "years.");
```

Il valore iniziale della variabile `year` deve essere 0 o 1? La condizione da verificare è `balance < targetBalance` oppure `balance <= targetBalance`? In queste espressioni è facile cadere in un errore *per scarto di uno*.

Alcuni programmatore tentano di risolvere gli errori per scarto di uno inserendo casualmente `+1` o `-1` fino a quando il programma sembra funzionare. Si tratta, ovviamente, di una pessima strategia, che può richiedere molto tempo per sperimentare tutte le diverse possibilità: dedicare al problema un piccolo ragionamento fa veramente risparmiare tempo.

Fortunatamente, gli errori per scarto di uno si possono facilmente evitare, semplicemente riflettendo con cura su un paio di casi di prova e utilizzando le informazioni ottenute per individuare la logica corretta da esprimere nella condizione del ciclo.

Il valore iniziale della variabile `year` deve essere 0 o 1? Ipotizzate uno scenario con valori semplici: un saldo iniziale di \$100 e un tasso di interesse del 50%. Dopo l'anno 1 il saldo sarà diventato \$150, e, dopo l'anno 2, sarà \$225, comunque oltre \$200, quindi dopo due anni l'investimento è raddoppiato. Il ciclo è stato eseguito due volte, incrementando `year` ogni volta, quindi deve partire da 0, non da 1.

year	balance
0	\$ 100
1	\$ 150
2	\$ 225

In altre parole, la variabile `balance` indica il saldo dopo la fine dell'anno: all'inizio, la variabile `balance` contiene il saldo dopo zero anni, non dopo un anno.

Procediamo con il ragionamento: nella condizione bisogna inserire l'operatore di confronto `<` oppure `<=`? Questo è più difficile da capire, perché è raro che il saldo sia esattamente il doppio dell'investimento iniziale. Naturalmente esiste un caso in cui ciò si verifica: quando l'interesse è pari al 100%. In questa situazione il ciclo viene eseguito una volta sola, dopodiché `year` diventa uguale a 1, mentre `balance` è esattamente uguale a `2 * INITIAL_BALANCE`. L'investimento è raddoppiato dopo un anno? Sì, quindi non bisogna eseguire nuovamente il ciclo. Se la condizione della verifica è `balance < targetBalance`, il

Gli errori per scarto di uno sono molto comuni nella programmazione dei cicli: per evitarli, ragionate con cura sui casi più semplici.

ciclo si ferma, come dovrebbe. Per contro, se la condizione fosse `balance <= targetBalance`, il ciclo verrebbe eseguito ancora una volta.

In altre parole, il ciclo deve continuare a sommare gli interessi finché il saldo *non è ancora raddoppiato*.

## 6.2 Problem Solving: esecuzione manuale

Per tenere traccia dell'esecuzione di un programma, lo si simula a mano, eseguendo le istruzioni passo dopo passo e annotando i valori delle variabili.

Nei Suggerimenti per la programmazione 5.5 avete visto come tenere traccia a mano dell'esecuzione di un programma, descritto mediante codice o pseudocodice: si scrivono su un foglio i nomi delle variabili, eseguendo mentalmente le istruzioni passo dopo passo e aggiornando i valori delle variabili.

Se possibile, è meglio scrivere su un foglio di carta anche il codice, in modo da poter contrassegnare la riga in esecuzione mediante un pennarello o una graffetta. Ogni volta che una variabile viene modificata, fate un segno di cancellazione sul vecchio valore (lasciandolo però visibile) e scrivete il nuovo valore un po' più in basso. Quando, invece, il programma visualizza qualche informazione, scrivetela in un'altra colonna.

Consideriamo questo esempio. Quale valore viene visualizzato?

```
int n = 1729;
int sum = 0;
while (n > 0)
{
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
System.out.println(sum);
```

Ci sono tre variabili: `n`, `sum` e `digit`.

<code>n</code>	<code>sum</code>	<code>digit</code>

Prima di entrare nel ciclo, le prime due variabili vengono inizializzate, rispettivamente, ai valori 1729 e 0.



```
int n = 1729;
int sum = 0;
while (n > 0)
{
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
System.out.println(sum);
```

<code>n</code>	<code>sum</code>	<code>digit</code>
1729	0	

Dato che `n` è maggiore di zero, si entra nel ciclo. Si assegna il valore 9 (cioè il resto della divisione di 1729 per 10) alla variabile `digit`, così come alla variabile `sum` (perché  $0 + 9 = 9$ ).

```

int n = 1729;
int sum = 0;
while (n > 0)
{
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
System.out.println(sum);

```

n	sum	digit
1729	0	
	9	9

Al termine di questa iterazione `n` diventa uguale a 172 (ricordando che il resto della divisione  $1729 / 10$  viene ignorato, perché entrambi gli operandi sono numeri interi).

Cancelliamo i vecchi valori (lasciandoli visibili) e scriviamo quelli nuovi sotto quelli vecchi.

```

int n = 1729;
int sum = 0;
while (n > 0)
{
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
System.out.println(sum);

```

n	sum	digit
1729	0	
172	9	9

A questo punto dobbiamo controllare di nuovo se la condizione è verificata.

```

int n = 1729;
int sum = 0;
while (n > 0)
{
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
System.out.println(sum);

```

Dato che `n` è ancora maggiore di zero, ripetiamo il ciclo. Ora si assegna il valore 2 a `digit`, il valore  $9 + 2 = 11$  a `sum` e il valore 17 a `n`.

n	sum	digit
1729	0	
172	9	9
17	11	2

Ripetiamo di nuovo il ciclo, assegnando a `digit` il valore 7, a `sum` il valore  $11 + 7 = 18$  e a `n` il valore 1.

n	sum	digit
1729	0	
172	9	9
17	11	2
1	18	7

Entriamo nel ciclo per l'ultima volta e assegniamo a `digit` il valore 1 e a `sum` il valore 19, mentre `n` diventa uguale a zero.

<code>n</code>	<code>sum</code>	<code>digit</code>
1729	0	
172	9	9
17	11	2
1	18	1
0	19	1

```
int n = 1729;
int sum = 0;
while (n > 0)
{
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
System.out.println(sum);
```

Dato che `n` è uguale a zero, questa condizione non è vera.

Ora la condizione `n > 0` è falsa, per cui si procede con l'enunciato che si trova dopo il ciclo.

```
int n = 1729;
int sum = 0;
while (n > 0)
{
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
System.out.println(sum);
```

<code>n</code>	<code>sum</code>	<code>digit</code>	<code>output</code>
1729	0		
172	9	9	
17	11	2	
1	18	1	
0	19	1	19

Questo enunciato visualizza il valore di `sum`, che è 19.

Naturalmente si può ottenere la stessa risposta anche eseguendo il programma! Tenere traccia dell'esecuzione passo dopo passo, a mano su carta, vi consente, però, di osservare *dettagli* che non potreste cogliere eseguendo semplicemente il codice. Analizziamo di nuovo cosa succede in ciascuna iterazione del ciclo:

- estraiamo l'ultima cifra di `n`;
- aggiungiamo tale cifra a `sum`;
- eliminiamo tale cifra da `n`.

L'esecuzione a mano, passo dopo passo, vi può aiutare a comprendere il funzionamento di un algoritmo che non conoscete.

In altre parole, il ciclo calcola la somma delle cifre di `n` e ora sapete quale sia l'elaborazione svolta da questo ciclo per qualsiasi valore di `n`, non solo per il valore usato come esempio (se vi state chiedendo perché mai qualcuno dovrebbe voler calcolare la somma delle cifre che compongono un numero, ricordate che, ad esempio, operazioni di questo tipo sono utili nella verifica della validità dei numeri di carte di credito e di altre forme di numeri identificativi).

L'esecuzione a mano, passo dopo passo, non vi aiuta soltanto a comprendere meglio il funzionamento di codice corretto, ma è una tecnica efficace anche per scoprire errori. Quando un programma si comporta in modo inatteso, prendere un foglio di carta e tenete traccia dei valori assunti dalle variabili mentre navigate mentalmente nel codice seguendo il flusso d'esecuzione.

L'esecuzione a mano, passo dopo passo, può evidenziare errori nel codice o nello pseudocodice.

Per fare questo non c'è bisogno di avere un programma funzionante: lo si può fare anche con lo pseudocodice e, in effetti, è consigliato farlo prima di affrontare il problema della sua traduzione in codice, per verificare che sia corretto.



## Auto-valutazione

6. Eseguite a mano, passo dopo passo, il codice seguente, mostrando i valori assunti da `n` e le informazioni visualizzate.

```
int n = 5;
while (n >= 0)
{
    n--;
    System.out.print(n);
}
```

7. Eseguite a mano, passo dopo passo, il codice seguente, mostrando i valori assunti da `n` e le informazioni visualizzate. Quale potenziale errore si può notare?

```
int n = 1;
while (n <= 3)
{
    System.out.print(n + ", ");
    n++;
}
```

8. Eseguite a mano, passo dopo passo, il codice seguente, nel caso in cui `a` vale 2 e `n` vale 4. Successivamente, spiegate quale elaborazione viene svolta dal codice per valori di `a` e `n` qualsiasi.

```
int r = 1;
int i = 1;
while (i <= n)
{
    r = r * a;
    i++;
}
```

9. Eseguite a mano, passo dopo passo, il codice seguente. Che errore riscontrate?

```
int n = 1;
while (n != 50)
{
    System.out.print(n);
    n = n + 10;
}
```

10. Lo pseudocodice che segue dovrebbe contare il numero di cifre presenti nel numero `n`:

```
count = 1
temp = n
while (temp > 10)
    Incrementa count.
    Dividi temp per 10.0.
```

Eseguitelo a mano, passo dopo passo, prima con  $n = 123$ , poi con  $n = 100$ . Che errore avete trovato?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R6.6 e R6.9, al termine del capitolo.



## Computer e società 6.1

### Pirateria digitale

Leggendo questo libro avete scritto un po' di programmi per computer e avete sperimentato in prima persona quanto impegno ci vuole per scrivere anche il più modesto dei programmi. Progettare e realizzare un vero prodotto software, come un'applicazione amministrativa o un gioco per computer, richiede una gran quantità di tempo e di denaro. Poche persone, e ancor meno aziende, sono disposte a investire tanto tempo e denaro se non hanno una ragionevole possibilità di ricavare da tale sforzo più denaro di quanto ne spendono (in realtà, alcune società regalano il loro software nella speranza che gli utenti passino, poi, a versioni a pagamento più sofisticate, oppure che visualizzino la pubblicità presente nel software; altre società regalano il software che consente agli utenti di leggere e utilizzare alcuni tipi di file, ma vendono il software che è necessario per creare quei file; infine, vi sono persone che regalano il proprio tempo, per puro entusiasmo, e producono programmi che si possono copiare liberamente).

Quando vende software, un'azienda deve poter contare sull'onestà dei propri clienti. È facile, per una persona priva di scrupoli, fare copie di programmi per computer senza

pagarli: nella maggior parte degli Stati questo è illegale e molti governi forniscono una protezione legale, sotto forma di leggi sul diritto d'autore e di brevetti, per incoraggiare lo sviluppo di nuovi prodotti. Gli Stati che tollerano forme diffuse di pirateria hanno scoperto di avere un'ampia disponibilità di software straniero a buon mercato, ma nessun produttore locale è tanto stupido da progettare del buon software per i cittadini di quei paesi, come per esempio un elaboratore di testi per la lingua locale o programmi di contabilità adatti alla normativa fiscale locale.

Quando cominciò a delinearsi un mercato di massa per il software, i fornitori erano furibondi per il denaro che perdevano a causa della pirateria e tentarono di combatterla con i mezzi più vari, per garantire che soltanto gli utenti legittimi potessero utilizzare il software. Alcuni produttori utilizzavano *chiavi hardware* (dette "dongle"), da collegare alla porta del calcolatore dedicata alla stampante per poter utilizzare il software. Gli utenti in regola odiavano queste misure: avevano pagato il loro software, ma dovevano rassegnarsi ad avere parecchie chiavi hardware che penzolavano dal retro dei loro computer.

Siccome è così facile e poco costoso fare copie pirata del software, e la probabilità di essere scoperti è

minima, dovete fare una scelta morale autonoma. Se un programma che vorreste tanto avere è troppo caro per i vostri mezzi, cosa fate: lo rubate o restate onesti e vi accontentate di un prodotto che potete permettervi?

Ovviamente, la pirateria non è limitata al software: lo stesso problema sorge anche per altri prodotti digitali. Può darsi che abbiate avuto l'occasione di copiare canzoni o filmati, senza pagarli, oppure vi può essere capitato di essere scontenti per un dispositivo, presente nel vostro riproduttore di musica, che protegge dalle copie illegali e che vi ha reso difficile ascoltare una canzone che avevate regolarmente pagato. Sinceramente, è difficile avere molta simpatia per un complesso musicale il cui editore faccia pagare un sacco di soldi per ciò che sembra aver richiesto un piccolo sforzo da parte loro, soprattutto quando lo si confronta con lo sforzo richiesto per progettare e realizzare un pacchetto software. Non di meno, sembra che sia giusto che artisti e autori vengano pagati in qualche modo per il loro lavoro.

Come pagare in modo corretto artisti, autori e programmatore, senza creare problemi ai clienti onesti, è un problema tuttora irrisolto e molti informatici sono coinvolti in ricerche in questo campo.

## 6.3 Il ciclo for

Si usa un ciclo for quando una variabile varia da un valore iniziale a un valore finale con un incremento o decremento costante.

Capita spesso di voler eseguire una sequenza di enunciati per un determinato numero di volte. Si può usare un ciclo while controllato da un contatore (*counter*), come in questo esempio:

```
int counter = 1; // inizializzazione del contatore
while (counter <= 10) // verifica del contatore
{
    System.out.println(counter); // oppure altre elaborazioni...
    counter++; // aggiornamento del contatore
}
```

Dal momento che questo ciclo è così comune, esiste una sintassi speciale che ne evidenzia la struttura:

```
for (int counter = 1; counter <= 10; counter++)
{
    System.out.println(counter);
}
```

Alcuni programmatore parlano, in casi come questo, di *ciclo controllato da un contatore*, per mettere in evidenza la differenza sostanziale che esiste rispetto ai cicli while visti nel Paragrafo 6.1, che sono *cicli controllati da un evento*, perché vengono eseguiti finché

## Sintassi di Java

### 6.2 L'enunciato for

#### Sintassi

```
for (inizializzazione; condizione; aggiornamento)
{
    enunciati
}
```

#### Esempio

Tra queste espressioni dovrebbe esistere una relazione.

Questa *inizializzazione* viene eseguita una sola volta, prima che il ciclo inizi.

Il ciclo viene eseguito finché questa *condizione* è vera.

Questo *aggiornamento* viene eseguito dopo ciascuna iterazione.

La variabile i è definita soltanto all'interno di questo ciclo for.

```
for (int i = 5; i <= 10; i++)
{
    sum = sum + i;
}
```

Questo ciclo viene eseguito 6 volte.

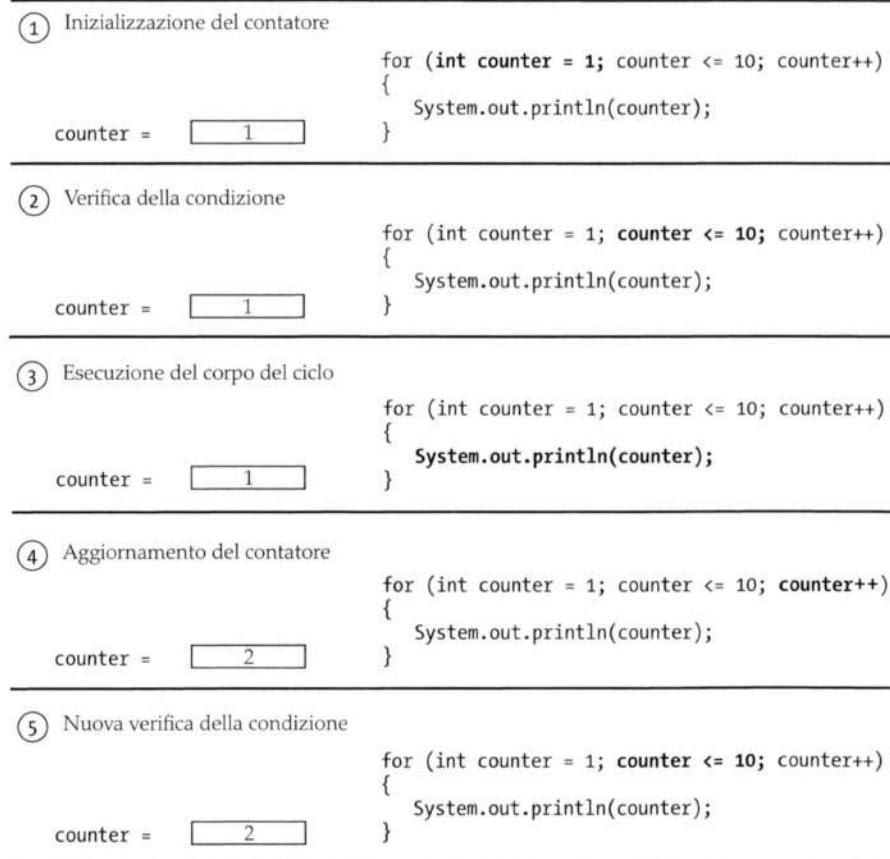
non si verifica un evento, cioè, nel nostro esempio di quel paragrafo, finché il saldo non raggiunge il suo valore finale previsto come obiettivo. Una definizione alternativa per i cicli controllati da contatore è quella di *ciclo definito*: si sa fin dall'inizio che il corpo di un tale ciclo verrà eseguito un determinato numero di volte. Al contrario, non si sa quante iterazioni saranno necessarie perché il saldo raggiunga il valore desiderato: in tal caso si parla di *ciclo indefinito*.

Il ciclo `for` mette ben in evidenza le espressioni usate per l'inizializzazione, la valutazione della condizione e l'aggiornamento, ma è molto importante comprendere che tali componenti non vengono eseguite insieme: a tale scopo può essere molto utile l'esame della Figura 3.

- L'*inizializzazione* viene eseguita una sola volta, prima di entrare nel ciclo (punto 1).
- La *condizione* viene verificata prima di ciascuna iterazione (punti 2 e 5).
- L'*aggiornamento* viene eseguito dopo ogni iterazione (punto 4).

**Figura 3**

Esecuzione di un ciclo `for`



Un ciclo `for` può anche usare un contatore che si decrementa, anziché incrementarsi:

```
for (int counter = 10; counter >= 0; counter--) . . .
```

e il valore dell'incremento o del decremento non deve necessariamente essere unitario:

```
for (int counter = 0; counter <= 10; counter = counter + 2) . . .
```

La Tabella 2 riporta altre possibili varianti.

Finora la variabile `counter` è stata sempre dichiarata nella sezione di inizializzazione del ciclo:

```
for (int counter = 1; count <= 10; counter++)
{
    . .
}
// qui la variabile counter non esiste più
```

Tale variabile esiste durante tutte le iterazioni del ciclo, ma non vi si può più accedere dopo la terminazione del ciclo stesso. Se la variabile `counter` viene, invece, dichiarata prima del ciclo, la si può utilizzare anche in seguito:

```
int counter;
for (counter = 1; count <= 10; counter++)
{
    . .
}
// qui la variabile counter esiste ancora
```

Il ciclo `for` viene spesso usato per esaminare tutti i caratteri di una stringa:

```
for (int i = 0; i < str.length(); i++)
{
    char ch = str.charAt(i);
    . . . // elabora ch
}
```

Osservate che la variabile `i`, che funge da contatore, parte da 0 e il ciclo termina quando `i` assume lo stesso valore della lunghezza della stringa. Ad esempio, se `str` ha lunghezza 5, all'interno del corpo del ciclo `i` assume i valori 0, 1, 2, 3 e 4: sono proprio le posizioni valide dei caratteri presenti nella stringa.

Vediamo un altro utilizzo tipico del ciclo `for`. Vogliamo calcolare l'aumento del saldo di un conto bancario di risparmio nell'arco di un certo numero di anni (`numberOfYears`), come in questa tabella:

Anno	Saldo
1	10500.00
2	11025.00
3	11576.25
4	12155.06
5	12762.82

Si può applicare lo schema che caratterizza il ciclo `for` perché la variabile `year` parte da 1 e aumenta di un'unità ad ogni passo, fino a raggiungere il valore finale previsto:

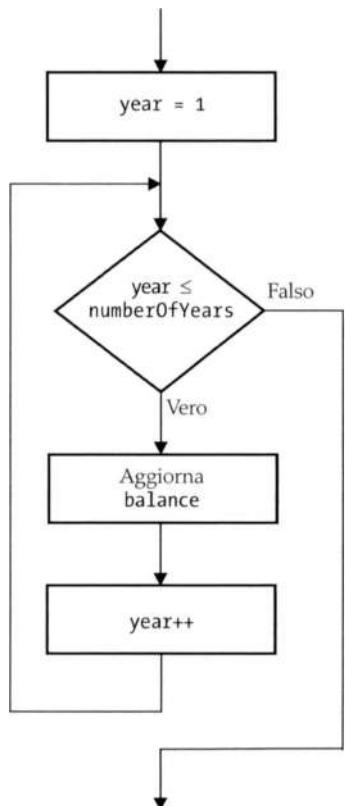
```

for (int year = 1; year <= numberOfYears; year++)
{
    . . . // aggiorna il saldo
}

```

La Figura 4 mostra il diagramma di flusso corrispondente, mentre nel seguito trovate il programma completo.

**Figura 4**  
Diagramma di flusso  
di un ciclo for



### File Investment.java

```

1 /**
2  * Classe che gestisce l'aumento di un investimento
3  * che accumula interessi annualmente con tasso fisso.
4 */
5 public class Investment
6 {
7     private double balance;
8     private double rate;
9     private int year;
10
11 /**
12  * Costruisce un oggetto che rappresenta un investimento
13  * con un dato saldo iniziale e un dato tasso di interesse.
14  * @param aBalance il saldo iniziale

```

```
15     @param aRate il tasso di interesse in percentuale
16 */
17 public Investment(double aBalance, double aRate)
18 {
19     balance = aBalance;
20     rate = aRate;
21     year = 0;
22 }
23
24 /**
25 Continua ad accumulare interessi fino al raggiungimento
26 del saldo desiderato.
27 @param targetBalance il saldo desiderato
28 */
29 public void waitForBalance(double targetBalance)
30 {
31     while (balance < targetBalance)
32     {
33         year++;
34         double interest = balance * rate / 100;
35         balance = balance + interest;
36     }
37 }
38
39 /**
40 Accumula interessi per un dato numero di anni.
41 @param numberOYears il numero di anni d'attesa
42 */
43 public void waitYears(int numberOYears)
44 {
45     for (int i = 1; i <= numberOYears; i++)
46     {
47         double interest = balance * rate / 100;
48         balance = balance + interest;
49     }
50     year = year + numberOYears;
51 }
52
53 /**
54 Ispeziona il saldo attuale dell'investimento.
55 @return il saldo attuale
56 */
57 public double getBalance()
58 {
59     return balance;
60 }
61
62 /**
63 Ispeziona il numero di anni durante i quali l'investimento
64 ha accumulato interessi.
65 @return il numero di anni dall'inizio dell'investimento
66 */
67 public int getYears()
68 {
69     return year;
70 }
71 }
```

### File InvestmentRunner.java

```

1  /**
2   * Questo programma calcola l'aumento di un
3   * investimento in un dato numero di anni.
4  */
5 public class InvestmentRunner
6 {
7     public static void main(String[] args)
8     {
9         final double INITIAL_BALANCE = 10000;
10        final double RATE = 5;
11        final int YEARS = 20;
12        Investment invest = new Investment(INITIAL_BALANCE, RATE);
13        invest.waitYears(YEARS);
14        double balance = invest.getBalance();
15        System.out.printf("The balance after %d years is %.2f\n",
16                          YEARS, balance);
17    }
18 }
```

### Esecuzione del programma

The balance after 20 years is 26532.98

**Tabella 2** Esempi con cicli for

Ciclo	Valori di i	Commento
for (i = 0; i <= 5; i++)	0 1 2 3 4 5	Il ciclo viene eseguito 6 volte (Suggerimenti per la programmazione 6.3).
for (i = 5; i >= 0; i--)	5 4 3 2 1 0	Per valori decrescenti si usa i--.
for (i = 0; i < 9; i = i + 2)	0 2 4 6 8	Per un incremento di 2 si usa i = i + 2.
for (i = 0; i != 9; i = i + 2)	0 2 4 6 8 10 12 14 ... (ciclo infinito)	Per evitare questo problema si può usare < oppure <=, invece di !=.
for (i = 0; i <= 20; i = i * 2)	1 2 4 8 16	Per modificare i si può specificare una regola qualsiasi, come questa che lo raddoppia ad ogni iterazione.
for (i = 0; i < str.length(); i++)	0 1 2 ... fino all'ultimo indice valido per la stringa str.	Nel corpo del ciclo, per ottenere il carattere in posizione i si usa l'espressione str.charAt(i).



### Auto-valutazione

11. Scrivete sotto forma di ciclo while il ciclo for presente nella classe Investment.
12. Quanti numeri vengono visualizzati da questo ciclo?
 

```
for (int n = 10; n >= 0; n--)
{
    System.out.println(n);
}
```
13. Scrivete un ciclo for che visualizzi tutti i numeri pari compresi tra 10 e 20 (estremi inclusi).

14. Scrivete un ciclo `for` che calcoli la somma dei numeri interi che vanno da 1 a `n`.
15. Come si può modificare il programma `InvestmentRunner.java` in modo che visualizzi i saldi dopo 20, 40, ..., 100 anni?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R6.7, R6.13, E6.9 e E6.13, al termine del capitolo.

## Suggerimenti per la programmazione 6.1

### Usare i cicli `for` solamente per lo scopo previsto

Un ciclo `for` è un'espressione *idiomatica* che rappresenta un ciclo avente una forma specifica: un contatore viene modificato con incrementi o decrementi costanti a partire da un valore iniziale per arrivare a un valore finale.

Il compilatore non verifica se le espressioni usate per le sezioni di inizializzazione, condizione e aggiornamento siano tra loro correlate e, ad esempio, il ciclo seguente è valido:

```
// pessimo stile: nell'intestazione ci sono espressioni non correlate
for (System.out.print("Inputs: "); in.hasNextDouble(); sum = sum + x)
{
    x = in.nextDouble();
}
```

Nonostante la correttezza sintattica, i programmatore che leggeranno questo ciclo saranno probabilmente confusi, perché non corrisponde a ciò che si aspettano da un ciclo `for`. Per le iterazioni che non si adattano alla struttura del ciclo `for`, utilizzate un ciclo `while`.

Bisogna anche fare attenzione a non modificare il valore del contatore all'interno del corpo di un ciclo `for`. Considerate l'esempio seguente:

```
for (int counter = 1; counter <= 100; counter++)
{
    if (counter % 10 == 0) // salta i valori divisibili per 10
    {
        counter++; // pessimo stile: non si aggiorna il contatore nel corpo
    }
    System.out.println(counter);
}
```

L'aggiornamento del contatore all'interno del corpo di un ciclo `for` crea confusione, perché tale contatore verrà aggiornato *di nuovo* al termine dell'iterazione del ciclo. In pratica, durante alcune iterazioni di questo ciclo la variabile `counter` viene incrementata due volte, in altre iterazioni una volta sola: si tratta di una caratteristica che contrasta con ciò che i programmatore pensano che accada in un ciclo `for`.

Se vi trovate in una situazione come questa, potete cambiare il ciclo e trasformarlo in un ciclo `while`, oppure implementare il "salto" di un valore in altro modo, ad esempio così:

```
for (int counter = 1; counter <= 100; counter++)
{
```

```

if (counter % 10 != 0) // esegue solo per valori non divisibili per 10
{
    System.out.println(counter);
}
}

```



## Suggerimenti per la programmazione 6.2

### Limiti simmetrici e asimmetrici

Immaginiamo di voler visualizzare i numeri interi che vanno da 1 a 10. Useremo, ovviamente, un ciclo come questo:

```
for (int i = 1; i <= 10; i++)
```

I valori di *i* sono delimitati dalla relazione  $1 \leq i \leq 10$ . Dal momento che su entrambi i lati della relazione ci sono operatori di confronto del tipo  $\leq$ , i limiti sono detti *simmetrici*.

Quando si esaminano tutti i caratteri presenti in una stringa, è più naturale usare limiti *asimmetrici*, come in questo esempio:

```
for (int i = 0; i < str.length(); i++)
```

Durante l'esecuzione di questo ciclo, *i* assume tutti i valori corrispondenti alle posizioni valide all'interno della stringa, e, nel corpo del ciclo, si può accedere al carattere in posizione *i* usando l'invocazione `str.charAt(i)`. I valori di *i* sono delimitati dalla relazione  $0 \leq i < str.length()$ , con l'operatore di confronto  $\leq$  sulla sinistra e l'operatore  $<$  sulla destra: è giusto, perché `str.length()` non è una posizione valida. Limiti di questo tipo di dicono *asimmetrici*.

In casi come questi non conviene forzare una simmetria in modo artificioso:

```
for (int i = 0; i <= str.length() - 1; i++) // meglio usare <
```

perché la forma asimmetrica è di più facile comprensione.



## Suggerimenti per la programmazione 6.3

### Contare le iterazioni

Individuare il limite inferiore e superiore di un ciclo può non essere banale. Bisogna iniziare da zero o da uno? Come condizione di terminazione, è meglio usare  $\leq b$  oppure  $< b$ ?

Per comprendere meglio il funzionamento di un ciclo è molto utile contare il numero delle sue iterazioni, operazione che risulta essere più agevole per i cicli con limiti asimmetrici. Questo ciclo viene eseguito  $b - a$  volte:

```
for (int i = a; i < b; i++)
```

Per esempio, questo ciclo, che esamina ordinatamente i caratteri di una stringa, viene eseguito `str.length()` volte:

```
for (int i = 0; i < str.length(); i++)
```

Ciò è perfettamente logico, perché in una stringa esistono `str.length()` caratteri. Questo ciclo, che ha limiti simmetrici:

```
for (int i = a; i <= b; i++)
```

viene eseguito  $b - a + 1$  volte. Questo “+1” aggiuntivo è fonte di molti errori di programmazione. Per esempio, questo ciclo viene ripetuto 11 volte:

```
for (int i = 0; i <= 10; i++)
```

Può darsi che questo sia l'effetto desiderato; se non lo è, iniziate da 1 oppure usate la condizione  $< 10$ .



## Argomenti avanzati 6.1

### Variabili dichiarate nell'intestazione di un ciclo for

Come detto, in Java è ammesso dichiarare una variabile nell'intestazione di un ciclo `for`. Ecco la forma più comune di questa sintassi:

```
for (int i = 1; i <= n; i++)
{
    ...
}
// qui la variabile i non è più definita
```

La visibilità della variabile si estende fino alla fine del ciclo `for` e, quando il ciclo termina, `i` non è più definita. Se vi occorre usare il valore della variabile anche dopo la fine del ciclo, dovete dichiararla all'esterno del ciclo. In questo esempio il valore di `i` non vi interessa, perché sapete già che, alla fine del ciclo, sarà uguale a  $n + 1$  (in realtà, non è del tutto vero, perché è possibile interrompere un ciclo prima della fine: consultate Argomenti avanzati 6.4). Se, invece, ci sono due o più condizioni di terminazione, potreste avere ancora bisogno della variabile. Per esempio, considerate questo ciclo:

```
for (i = 1; balance < targetBalance && i <= n; i++)
{
    ...
}
```

Volete che il saldo raggiunga un valore prestabilito, ma siete disposti ad aspettare al massimo un numero di anni prefissato. Se il saldo raggiunge in anticipo il valore desiderato, potrete voler conoscere il valore di `i`: in questo caso, non è appropriato dichiarare la variabile nell'intestazione del ciclo.

Nella coppia di cicli `for` riportata qui di seguito, notate come le due variabili `i` siano indipendenti:

```
for (int i = 1; i <= 10; i++)
{
    System.out.println(i * i);
}
```

```
for (int i = 1; i <= 10; i++) // dichiara una nuova variabile i
{
    System.out.println(i * i * i);
}
```

Nell'intestazione del ciclo potete dichiarare più variabili, purché siano dello stesso tipo, e potete anche scrivere più espressioni di aggiornamento, separandole mediante virgole:

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

Tuttavia, molte persone ritengono che un ciclo `for` che modifica più di una variabile sia poco chiaro e anche noi sconsigliamo l'utilizzo di questa forma (consultate Suggerimenti per la programmazione 6.1). Create, piuttosto, un ciclo `for` controllato da un unico contatore e aggiornate l'altra variabile esplicitamente:

```
int j = 10;
for (int i = 0; i <= 10; i++)
{
    ...
    j--;
}
```

## 6.4 Il ciclo do

Si usa un ciclo `do` quando il corpo del ciclo deve essere eseguito almeno una volta.

Ci sono casi in cui si vuole eseguire un ciclo almeno una volta, verificando la condizione di terminazione dopo avere eseguito gli enunciati del corpo. Il ciclo `do` serve a questo:

```
do
{
    enunciati
}
while (condizione);
```

In un ciclo `do`, prima viene eseguito il corpo, poi viene verificata la condizione.

A volte un ciclo di questo tipo viene chiamato *ciclo con verifica posticipata*, perché la condizione viene sottoposta a verifica dopo l'esecuzione del corpo del ciclo; viceversa, i cicli `while` e `for` sono *cicli a verifica anticipata*, dove la condizione è verificata prima di entrare nel corpo del ciclo.

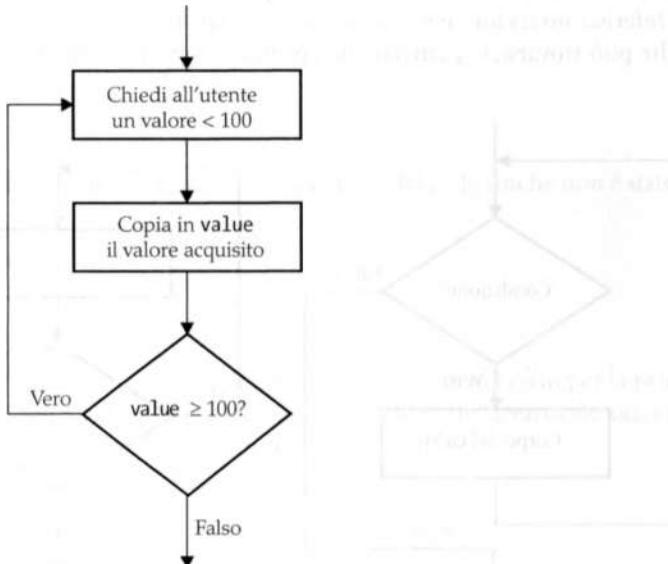
Un tipico campo di applicazione del ciclo `do` è la verifica della validità dei dati acquisiti in ingresso. Immaginiamo di chiedere all'utente di fornire un valore inferiore a 100: se non fa attenzione e digita un numero troppo elevato, continuiamo semplicemente a richiedere un dato valido, finché il valore non è corretto. Ovviamente non possiamo controllare la validità di un valore prima che l'utente lo inserisca: in questa situazione un ciclo `do` è perfetto, come si può vedere anche nella Figura 5:

```

int value;
do
{
    System.out.print("Enter an integer < 100: ");
    value = in.nextInt();
} while (value >= 100);

```

**Figura 5**  
Diagramma di flusso  
di un ciclo do



### Auto-valutazione

16. Immaginate di voler verificare che un valore acquisito in ingresso sia almeno 0 e al massimo 100. Modificate il ciclo do presentato in questo paragrafo in modo che verifichi questa condizione.
17. Riscrivete il ciclo do presentato in questo paragrafo usando un ciclo while. Che svantaggio c'è in questa nuova soluzione?
18. Se Java non avesse il ciclo do, sarebbe possibile riscrivere qualsiasi ciclo do usando un ciclo while?
19. Scrivete un ciclo do che legga numeri interi e ne calcoli la somma, terminando la procedura dopo aver letto il valore zero.
20. Scrivete un ciclo do che legga numeri interi e ne calcoli la somma, terminando la procedura dopo aver letto il valore zero oppure dopo aver letto due numeri consecutivi uguali tra loro. Ad esempio, se i valori in ingresso sono 1 2 3 4 4, la somma è 14 e il ciclo termina.

### Per far pratica

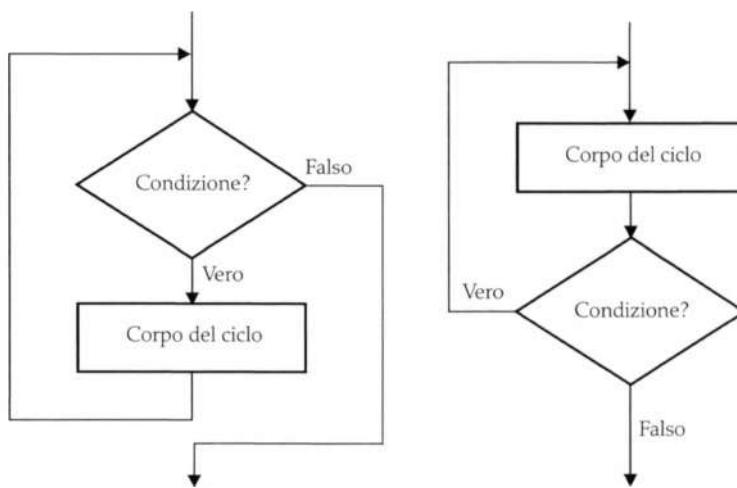
A questo punto si consiglia di svolgere gli esercizi R6.12, R6.19 e R6.20, al termine del capitolo.



## Suggerimenti per la programmazione 6.4

### Diagrammi di flusso per cicli

Nel Paragrafo 5.5 avete visto come si possano usare i diagrammi di flusso per visualizzare il flusso dell'esecuzione di un programma. I cicli che possono far parte di un diagramma di flusso sono di due tipi, che corrispondono, in Java, a un ciclo `while` e a un ciclo `do`. Differiscono tra loro per il posizionamento della verifica della condizione di terminazione, che può trovarsi, rispettivamente, prima o dopo il corpo del ciclo.



Come detto nel Paragrafo 5.5, nei diagrammi di flusso bisogna evitare il fenomeno del “codice a spaghetti”. Nel caso dei cicli, questo significa che non deve mai essere presente una freccia che arriva all'interno del corpo di un ciclo.

## 6.5 Applicazione: elaborazione di valori sentinella

In questo paragrafo imparerete a scrivere cicli che acquisiscono ed elaborano una sequenza di valori in ingresso.

Un valore sentinella segnala la fine di un insieme di dati, ma non fa parte dei dati stessi.

Ogni volta che si legge una sequenza di valori in ingresso, serve una strategia per segnalare che la sequenza è terminata. Alcune volte sarete fortunati e i valori validi in ingresso non potranno essere uguali a zero, per cui si potrà chiedere all'utente di continuare a fornire numeri, introducendo uno zero quando questi sono finiti. Analogamente, se lo zero è un valore valido ma i numeri negativi non lo sono, potete usare `-1` per segnalare la fine della sequenza.

Un tale valore, che non è un vero è proprio valore da acquisire in ingresso per l'elaborazione successiva, ma serve soltanto per segnalare la fine della sequenza, viene chiamato **sentinella**.

Rendiamo operativa questa strategia in un programma che calcoli la media di un insieme di valori, che rappresentano salari di dipendenti. In questo esempio useremo come sentinella il valore `-1`: certamente nessun dipendente lavorerebbe per un salario

negativo, ma ci potrebbero essere volontari disposti a lavorare gratis, cioè con salario uguale a zero.

All'interno del ciclo acquisiamo un valore in ingresso. Se non è uguale a -1, lo elaboriamo: per calcolare la media, ci serve la somma di tutti i valori acquisiti e il loro numero.

```
salary = in.nextDouble();
if (salary != -1)
{
    sum = sum + salary;
    count++;
}
```

Dobbiamo continuare l'esecuzione del ciclo finché non è stato rilevato il valore sentinella:

```
while (salary != -1)
{
    . . .
}
```

C'è soltanto un problema: quando si entra nel ciclo per la prima volta, non è ancora stato letto alcun valore. Dobbiamo garantire che la variabile `salary` venga inizializzata con un valore diverso dalla sentinella:

```
double salary = 0;
// qualsiasi valore diverso da -1 va bene
```

Quando il ciclo termina la propria esecuzione, calcoliamo e visualizziamo il valore medio. Ecco il programma completo.

### File `SentinelDemo.java`

```
1 import java.util.Scanner;
2
3 /**
4     Visualizza la media di valori terminati da una sentinella.
5 */
6 public class SentinelDemo
7 {
8     public static void main(String[] args)
9     {
10         double sum = 0;
11         int count = 0;
12         double salary = 0;
13         System.out.print("Enter salaries, -1 to finish: ");
14         Scanner in = new Scanner(System.in);
15
16         // elabora dati finché non arriva la sentinella
17
18         while (salary != -1)
19         {
20             salary = in.nextDouble();
21             if (salary != -1)
```

```

22     {
23         sum = sum + salary;
24         count++;
25     }
26 }
27
28 // calcola e visualizza la media
29
30 if (count > 0)
31 {
32     double average = sum / count;
33     System.out.println("Average salary: " + average);
34 }
35 else
36 {
37     System.out.println("No data");
38 }
39 }
40 }
```

### Esecuzione del programma

```
Enter salaries or -1 to finish: 10 10 40 -1
Average salary: 20
```

Per controllare un ciclo si può anche usare una variabile booleana: le si assegna un valore prima di entrare nel ciclo e le si assegna il valore opposto quando si vuole che il ciclo termini.

Ad alcuni programmati non piace il “trucco” dell’inizializzazione della variabile usata per acquisire i dati in ingresso, a cui, come abbiamo visto, viene preliminarmente assegnato un valore diverso dalla sentinella. Un diverso approccio prevede l’utilizzo di una variabile booleana:

```
System.out.print("Enter salaries, -1 to finish: ");
boolean done = false;
while (!done)
{
    value = in.nextDouble();
    if (value == -1)
    {
        done = true;
    }
    else
    {
        . . . // elabora value
    }
}
```

La sezione Argomenti avanzati 6.4 mostra un meccanismo alternativo per uscire da un tale ciclo.

Consideriamo ora il caso in cui qualsiasi numero (positivo, zero o negativo) sia accettabile come valore in ingresso. In una tale situazione è necessario utilizzare una sentinella che non sia un numero (come, ad esempio, la lettera Q, iniziale di *quit*, “smettere”). Come avete visto nel Paragrafo 5.8, l’invocazione

```
in.hasNextDouble()
```

restituisce il valore `false` se il successivo dato in ingresso non è un numero, quindi si può acquisire ed elaborare un insieme di valori numerici con un ciclo come questo:

```
System.out.print("Enter values, Q to quit: ");
while (in.hasNextDouble())
{
    value = in.nextDouble();
    . . . // elabora value
}
```



## Auto-valutazione

21. Cosa visualizza il programma `SentinelDemo.java` se l'utente digita subito `-1`, appena viene chiesto un valore?
22. Perché il programma `SentinelDemo.java` ha *due* verifiche di questo tipo?  
`salary != -1`
23. Cosa succederebbe se, nel programma `SentinelDemo.java`, la dichiarazione (con inizializzazione) della variabile `salary` venisse modificata in questo modo?  
`double salary = -1;`
24. Nell'ultimo esempio di questo paragrafo abbiamo chiesto all'utente di fornire un valore oppure di scrivere `Q` per terminare la sequenza. Cosa succede se l'utente digita una lettera diversa?
25. Cosa c'è di sbagliato nel seguente ciclo che vorrebbe acquisire in ingresso una sequenza di valori?

```
System.out.print("Enter values, Q to quit: ");
do
{
    double value = in.nextDouble();
    sum = sum + value;
    count++;
}
while (in.hasNextDouble());
```

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi R.6.16, E6.20 e E6.21, al termine del capitolo.



## Argomenti avanzati 6.2

### Redirezione di input e output

Si usa la redirezione di input per acquisire dati da un file e la redirezione di output per scrivere in un file i risultati prodotti dal programma.

Consideriamo di nuovo il programma `SentinelDemo.java` che calcola il valore medio di una sequenza di dati numerici forniti in ingresso. Se usate questo programma è molto probabile che abbiate i valori già scritti in un file ed è un vero peccato doverli scrivere di nuovo. L'interfaccia della riga di comando messa a disposizione dal sistema operativo consente di collegare un file al flusso di ingresso del programma, in modo che tutti i caratteri presenti nel file appaiano al programma come se fossero stati digitati dall'utente sulla tastiera. Se scrivete:

```
java SentinelDemo < numbers.txt
```

il programma viene eseguito, ma non si aspetta più di ricevere dati dalla tastiera. Tutti i comandi che leggono dati in ingresso ricevono le informazioni contenute nel file `numbers.txt`, mediante un meccanismo che viene chiamato **redirezione di input** (*input redirection*).

La redirezione di input è anche uno strumento molto utile per il collaudo dei programmi. Durante lo sviluppo di un programma, una delle fasi più noiose è l'eliminazione di eventuali errori, un'azione che va ripetuta più volte fornendo in ingresso sempre gli stessi dati, quelli previsti dai casi di prova che sono stati progettati. È decisamente meglio dedicare alcuni minuti all'inserimento di tali dati in file di testo, una volta per tutte, per poi usare ripetutamente la redirezione di input.

In modo analogo, si può effettuare anche la **redirezione di output**, anche se nel programma che stiamo considerando come esempio non è particolarmente utile. Se scrivete:

```
java SentinelDemo < numbers.txt > output.txt
```

il programma viene eseguito e, al termine, il file `output.txt` conterrà tutto quanto avrebbe dovuto essere visualizzato dal programma, cioè i prompt di input e il messaggio finale:

```
Enter salaries, -1 to finish: Enter salaries, -1 to finish:  
Enter salaries, -1 to finish: Enter salaries, -1 to finish:  
Average salary: 15
```

La redirezione di output è ovviamente molto utile per quei programmi che producono molte informazioni: in questo modo possono essere poi ritrovate in un file, che può anche essere stampato su carta.



## Argomenti avanzati 6.3

### Il problema del "ciclo e mezzo"

A volte l'acquisizione di un insieme di dati richiede un ciclo come questo, che è in qualche modo poco elegante:

```
boolean done = false;  
while (!done)  
{  
    String input = in.next();  
    if (input.equals("Q"))  
    {  
        done = true;  
    }  
    else  
    {  
        . . . // elabora il dato presente nella stringa input  
    }  
}
```

Il vero controllo della condizione di terminazione del ciclo si trova all'interno del corpo del ciclo stesso, non all'inizio. Questa costruzione sintattica viene detta **ciclo e mezzo** (*loop and a half*) perché, per poter sapere se il ciclo deve terminare, è necessario procedere nella prima metà del suo corpo.

Ad alcuni programmati non piace l'uso di una variabile booleana aggiuntiva per il controllo del ciclo. Ci sono due caratteristiche del linguaggio Java che possono evitare il "ciclo e mezzo": probabilmente nessuna delle due è una soluzione migliore, ma entrambi gli approcci sono abbastanza comuni, quindi è bene conoscerli per meglio comprendere il codice scritto da altri.

Nell'espressione della condizione di controllo del ciclo è possibile combinare una verifica e un'assegnazione:

```
while (!(input = in.next()).equals("Q"))
{
    . . . // elabora il dato presente nella stringa input
}
```

L'espressione

```
(input == in.next()).equals("Q")
```

significa: "per prima cosa invoca `in.next()`, poi assegna a `input` il valore restituito e, infine, verifica se `input` è uguale a "Q". Si tratta di un'espressione che ha un effetto collaterale: lo scopo principale dell'espressione è quello di fungere da condizione di terminazione per il ciclo `while`, ma svolge anche un altro compito, perché acquisisce una parola in ingresso e la memorizza nella variabile `input`. In generale, usare effetti collaterali non è una buona idea, perché i programmi risultano più difficili da comprendere e modificare, ma in questo caso la soluzione è in qualche modo attraente, perché elimina la variabile di controllo `done`, che, a suo modo, rende anch'essa più difficile la lettura e la manutenzione del codice.

L'altra soluzione consiste nell'uscita del ciclo da un punto interno al suo corpo, usando un enunciato `return` o un enunciato `break` (discusso nella sezione Argomenti avanzati 6.4).

```
public void processInput(Scanner in)
{
    while (true)
    {
        String input = in.next();
        if (input.equals("Q"))
        {
            return;
        }
        . . . // elabora il dato presente nella stringa input
    }
}
```

## Argomenti avanzati 6.4

### Gli enunciati `break` e `continue`

Avete già visto l'enunciato `break` in Argomenti avanzati 5.2, dove veniva usato per uscire da un enunciato `switch`. Oltre che per abbandonare un enunciato `switch`, un enunciato `break` si può usare anche per uscire da un ciclo `while`, `for` o `do`.

Nell'esempio seguente, l'enunciato `break` termina il ciclo quando viene raggiunta la fine dei dati in ingresso:

```

while (true)
{
    String input = in.next();
    if (input.equals("Q"))
    {
        break;
    }
    . . . // elabora il dato presente nella stringa input
}

```

La presenza di enunciati `break` all'interno di un ciclo ne può complicare la comprensione, perché occorre un'analisi attenta per determinare le modalità di uscita dal ciclo. Tuttavia, a fronte della scomodità di dover inserire una variabile dedicata solamente al controllo del ciclo, alcuni programmati ritengono che gli enunciati `break` siano utili nel caso del "ciclo e mezzo". Spesso questo tema è argomento di accesi dibattiti, piuttosto sterili: in questo libro eviteremo di usare l'enunciato `break`, lasciando a voi la scelta nei vostri programmi.

In Java, esiste anche una seconda forma per l'enunciato `break`, utilizzata per uscire da un enunciato annidato. L'enunciato `break etichetta;` fa proseguire immediatamente l'esecuzione del programma dalla fine dell'enunciato preceduto dall'etichetta. Si può contrassegnare mediante un'etichetta qualsiasi enunciato, compreso un enunciato `if` o un blocco di enunciati. La sintassi è la seguente:

*etichetta: enunciato*

L'enunciato `break` con etichetta fu inventato proprio per abbandonare una serie di cicli annidati.

```

cicloesterno:
while (condizione del ciclo esterno)
{ ...
    while (condizione del ciclo interno)
    { ...
        if (è successo qualcosa di veramente brutto)
        {
            break cicloesterno;
        }
    }
}
se "è successo qualcosa di veramente brutto" prosegue da qui

```

Questa situazione è, ovviamente, alquanto rara: invece di usare cicli annidati in modo contorto, suggeriamo di provare a introdurre metodi ausiliari.

Esiste, infine, l'enunciato `continue`, che fa proseguire l'esecuzione del programma dalla fine della *iterazione attuale* di un ciclo. Ecco un possibile utilizzo di questo enunciato:

```

while (!done)
{
    String input = in.next();
    if (input.equals("Q"))
    {
        done = true;
    }
}

```

```

        continue; // salta alla fine del corpo del ciclo
    }
    . . . // elabora il dato presente nella stringa input
    // l'enunciato continue provoca la prosecuzione da questo punto
}

```

Usando l'enunciato `continue`, non è necessario inserire all'interno di una clausola `else` la parte rimanente del codice del corpo del ciclo: si tratta, però, di un vantaggio piuttosto limitato e pochi programmatore utilizzano questo enunciato.

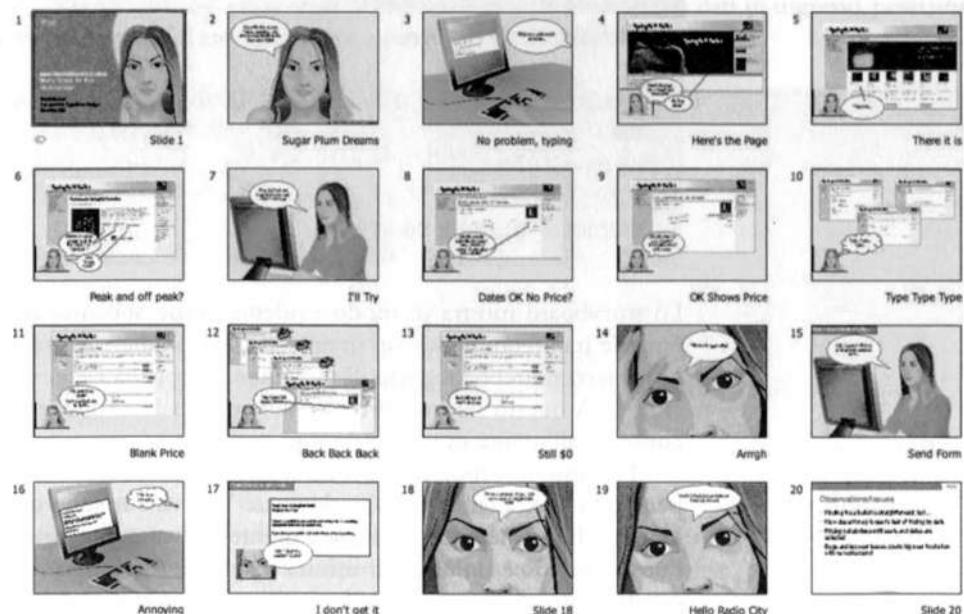
## 6.6 Problem Solving: storyboard

Quando si progetta un programma bisogna pianificare con cura la sua interazione con l'utente. Che informazioni deve fornire l'utente, e in quale ordine? Che informazioni deve visualizzare il programma, e con quale formato? Cosa deve succedere se si verifica un errore? Quando termina il programma?

Questa pianificazione è simile al progetto di un film o di un gioco per computer, dove vengono ampiamente usati gli storyboard per pianificare le azioni che si devono svolgere. Uno storyboard è costituito da immagini che mostrano una sintesi di ciascuna fase dell'azione, con annotazioni o didascalie che spiegano ciò che succede, in generale e nel caso specifico. Analogamente, si usano storyboard anche nello sviluppo del software, come si può vedere nella Figura 6.

**Figura 6**

Storyboard per il progetto di un'applicazione web



Quando si inizia a progettare un programma, delineare uno storyboard è veramente molto utile, perché siete quasi costretti a chiedervi quali siano le informazioni di cui avete bisogno per determinare le risposte desiderate dall'utente del programma e come,

Lo sviluppo di uno storyboard vi aiuta a capire quali siano i dati che il vostro programma deve acquisire e le risposte che deve fornire.

alla fine, presentare all'utente tali risposte. Si tratta di considerazioni importanti, sulle quali dovete prendere una decisione prima di progettare un algoritmo che calcoli le risposte previste.

Analizziamo un semplice esempio. Vogliamo scrivere un programma che aiuti gli utenti a rispondere a domande come "A quanti pollici (*inches*) equivalgono 30 centimetri?"

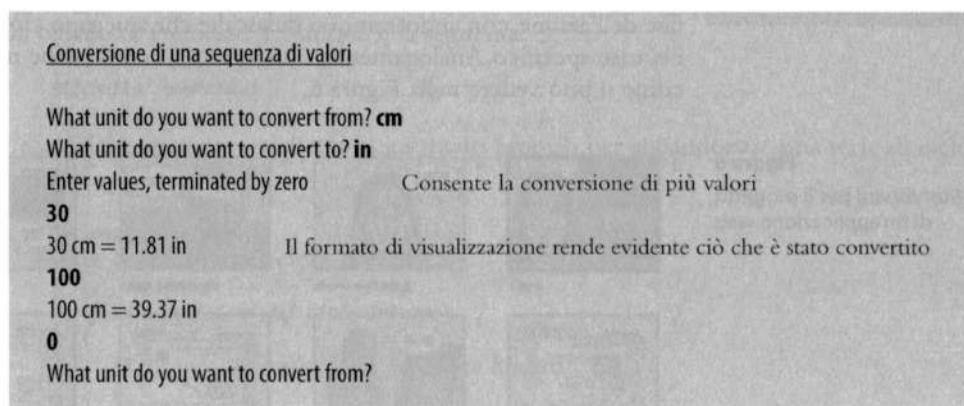
Quali sono le informazioni che devono essere fornite dall'utente?

- una quantità e l'unità di misura in cui è espressa;
- l'unità di misura verso cui effettuare la conversione.

Cosa facciamo se ci sono più quantità da convertire? L'utente potrebbe avere un'intera tabella di misure in centimetri, tutte da convertire in pollici.

Cosa facciamo se l'utente usa un'unità di misura che il nostro programma non sa gestire, come, ad esempio, ångström? E se l'utente chiede una conversione impossibile, come quella tra pollici e galloni?

Iniziamo con un primo pannello dello storyboard, nel quale scriveremo i dati che devono essere forniti dall'utente, magari usando un colore diverso (o una sottolineatura) rispetto al testo informativo che viene visualizzato dal programma.



Lo storyboard mostra in modo evidente come abbiamo intenzione di gestire un potenziale fraintendimento: un utente che voglia sapere quanti pollici corrispondono a 30 centimetri potrebbe leggere distrattamente la prima richiesta di dati e scrivere "pollici", cioè **in**. A questo punto, però, il risultato visualizzato, che sarebbe "**30 in = 76.2 cm**", consente all'utente di notare il proprio errore.

Lo stesso storyboard evidenzia anche un problema a cui non avevamo pensato: come pensiamo che l'utente possa "indovinare" che **cm** e **in** sono unità di misura valide? Il programma funzionerebbe anche se l'utente scrivesse **centimeter** e **inches**? Cosa succede se l'utente fornisce un'unità di misura sbagliata? Aggiungiamo una scena allo storyboard, con lo scopo di documentare la gestione degli errori.

Gestione delle unità non note (deve essere migliorata)

What unit do you want to convert from? **cm**

What unit do you want to convert to? **inches**

Sorry, unknown unit.

What unit do you want to convert to? **inch**

Sorry, unknown unit.

What unit do you want to convert to? **Uffa!**

Per evitare l'evidente frustrazione dell'utente, è meglio elencare le unità di misura che possono essere utilizzate.

From unit (in, ft, mi, mm, cm, m, km, oz, lb, g, kg, tsp, tbsp, pint, gal): **cm**

To unit: **in**

Non c'è bisogno di elencarle ancora

Siamo passati a un prompt decisamente più breve, per far posto ai nomi di tutte le unità di misura. L'Esercizio R.6.25 propone un approccio alternativo.

C'è un altro problema che non abbiamo ancora affrontato: come si esce dal programma? Il primo pannello dello storyboard lascia intendere che il programma possa continuare la propria esecuzione all'infinito.

Dopo aver visto la sentinella che termina una sequenza di dati in ingresso, possiamo chiedere all'utente se vuole proseguire con altre conversioni.

Uscita dal programma

From unit (in, ft, mi, mm, cm, m, km, oz, lb, g, kg, tsp, tbsp, pint, gal): **cm**

To unit: **in**

Enter values, terminated by zero

**30**

30 cm = 11.81 in

**0**

More conversions (y, n)? **n**

La sentinella provoca la visualizzazione di questo prompt  
(il programma termina)

Come potete vedere da questo caso di studio, per sviluppare un programma funzionante è quasi indispensabile utilizzare uno storyboard: per progettare la struttura di un programma, bisogna aver deciso il flusso delle interazioni con l'utente.

**Auto-valutazione**

26. Progettate un pannello di uno storyboard relativo a un programma che acquisisca una sequenza di voti ottenuti in questionari di valutazione e ne visualizzi la media. Il

- programma deve elaborare una sola sequenza di voti e la gestione degli errori non è richiesta.
27. Google dispone di una semplice interfaccia per la conversione di valori tra diverse unità di misura: basta scrivere la domanda e si ottiene la risposta.

The screenshot shows a Google search result for "How many inches in 30 cm". The search bar contains the query. Below it, the results section shows a snippet: "30 centimeters = 11.8110236 inches". There is also a link "More about calculator...".

- Progettate lo storyboard di un'interfaccia equivalente realizzata con un programma Java. Prima illustrate un caso in cui tutto va bene, poi mostrate come vengono gestiti due diversi tipi di errori.
28. Modificate il programma della domanda 26 supponendo che si voglia eliminare il voto peggiore della sequenza prima di calcolare la media. Nel relativo storyboard considerate anche il caso in cui l'utente fornisce un solo voto.
29. Che problema si riscontra nella realizzazione in Java di questo storyboard?

The application window has a title bar "Calcolo di valori medi". Inside, there are three examples of average calculation:

- Enter scores: **90 80 90 100 80**  
The average is 88
- Enter scores: **100 70 70 100 80**  
The average is 84
- Enter scores: **-1**  
*(il programma termina)*

To the right of the third example, the text "Come sentinella per terminare il programma si usa -1" is displayed.

30. Progettate lo storyboard di un programma che confronti la crescita di un investimento di \$10000 per un determinato numero di anni, con due diversi tassi di interesse.

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R6.24, R6.25 e R6.26, al termine del capitolo.

## 6.7 Algoritmi di uso frequente che utilizzano cicli

In questo paragrafo ci occupiamo di alcuni tra gli algoritmi più diffusi che fanno uso di cicli: li potete usare come punti di partenza per la progettazione dei vostri cicli.

### 6.7.1 Calcolo di somma e valor medio

Calcolare la somma di alcuni dati ricevuti in ingresso è un compito molto comune. Si usa un *totale corrente*: una variabile a cui viene sommato ciascun singolo valore e che, ovviamente, deve essere inizializzata a zero.

```
double total = 0;
while (in.hasNextDouble())
{
    double input = in.nextDouble();
    total = total + input;
}
```

Si noti che la variabile `total` viene dichiarata al di fuori del ciclo, per essere aggiornata al suo interno. La variabile `input`, invece, è dichiarata all'interno del corpo del ciclo, per cui ne viene creata una nuova copia per ogni valore acquisito, eliminandola al termine di ciascuna iterazione del ciclo.

Per calcolare il valore medio di una sequenza di valori, si accumula la somma totale mentre si contano i valori.

Per calcolare il valore medio della sequenza si contano i valori acquisiti e si divide la somma per tale conteggio, dopo aver controllato che questo non sia uguale a zero.

```
double total = 0;
int count = 0;
while (in.hasNextDouble())
{
    double input = in.nextDouble();
    total = total + input;
    count++;
}
double average = 0;
if (count > 0)
{
    average = total / count;
}
```

## 6.7.2

### Conteggio di valori che soddisfano una condizione

Per contare i valori che soddisfano una determinata condizione, si controllano uno ad uno e si incrementa un contatore ogni volta che il controllo va a buon fine.

Spesso si vuol sapere quanti valori soddisfano una determinata condizione: ad esempio, quanti spazi sono presenti in una stringa. Si usa un *contatore*, cioè una variabile che viene inizializzata a zero e incrementata ogni volta che la condizione è soddisfatta.

```
int spaces = 0;
for (int i = 0; i < str.length(); i++)
{
    char ch = str.charAt(i);
    if (ch == ' ')
    {
        spaces++;
    }
}
```

Se, ad esempio, `str` è la stringa "My Fair Lady", allora `spaces` viene incrementata due volte (quando `i` vale 2 e 7).

Si noti che la variabile `spaces` viene dichiarata al di fuori del ciclo, per essere aggiornata al suo interno. La variabile `ch`, invece, è dichiarata all'interno del corpo del ciclo, per cui ne viene creata una nuova copia per ogni iterazione del ciclo, eliminandola al termine di ciascuna iterazione.

Un ciclo di questo tipo può essere utilizzato anche per analizzare dati mentre vengono acquisiti in ingresso. Il ciclo seguente, ad esempio, acquisisce un testo analizzando una parola per volta e conta il numero di parole che hanno al massimo tre caratteri:

```
int shortWords = 0;
while (in.hasNext())
{
    String input = in.next();
    if (input.length() <= 3)
    {
        shortWords++;
    }
}
```

### 6.7.3 Identificazione della prima corrispondenza

Se l'obiettivo è quello di trovare una corrispondenza, si termina il ciclo non appena la si trova.

Quando volete contare i valori che soddisfano una determinata condizione, li dovete esaminare tutti, ma, se il vostro obiettivo è soltanto quello di trovare un valore che soddisfa la condizione, potete terminare la ricerca non appena ne trovate uno.

Ecco un ciclo che cerca il primo spazio all'interno di una stringa. Dato che non esaminiamo tutti i caratteri della stringa, invece di un ciclo `for` è meglio usare un ciclo `while`:

```
boolean found = false;
int position = 0;
while (!found && position < str.length())
{
    char ch = str.charAt(position);
    if (ch == ' ') { found = true; }
    else { position++; }
}
```

Se si trova un valore che soddisfa la condizione, `found` diventa `true` e la posizione di tale valore nella stringa rimane memorizzata nella variabile `position`. Diversamente, se il ciclo non riscontra alcuna corrispondenza, `found` rimane `false` e il ciclo prosegue finché `position` raggiunge il valore `str.length()`.

### 6.7.4 Richiesta ripetuta fino al raggiungimento di un obiettivo

Nell'esempio precedente abbiamo cercato, in una stringa, un valore che soddisfacesse una determinata condizione: la stessa procedura può essere applicata all'acquisizione di dati dall'utente. Supponiamo, ad esempio, di voler chiedere all'utente un valore positivo minore di 100. Continuiamo a chiedere un nuovo valore fino a quando l'utente non fornisce un valore valido:

```
boolean valid = false;
double input = 0;
while (!valid)
{
    System.out.print("Please enter a positive value < 100: ");
    input = in.nextDouble();
```

```
if (0 < input && input < 100) { valid = true; }
else { System.out.println("Invalid input. ");}
}
```

Si noti che la variabile `input` è stata dichiarata *al di fuori* del ciclo `while`, perché la vogliamo usare dopo che il ciclo è terminato.

## 6.7.5 Valore massimo e minimo

Per trovare il valore massimo, si aggiorna il valore massimo riscontrato fino a quel punto ogni volta che se ne trova uno maggiore.

```
double largest = in.nextDouble();
while (in.hasNextDouble())
{
    double input = in.nextDouble();
    if (input > largest)
    {
        largest = input;
    }
}
```

Questo algoritmo richiede che venga fornito in ingresso almeno un valore.

Per calcolare il valore minimo, basta semplicemente invertire il confronto:

```
double smallest = in.nextDouble();
while (in.hasNextDouble())
{
    double input = in.nextDouble();
    if (input < smallest)
    {
        smallest = input;
    }
}
```

## 6.7.6 Confronto di valori adiacenti

Per confrontare valori di ingresso adiacenti, si memorizza in una variabile il valore precedente.

Quando in un ciclo si elabora una sequenza di valori, a volte si ha la necessità di confrontare un valore che quello immediatamente precedente. Immaginiamo, ad esempio, di voler verificare se una sequenza di valori ricevuta in ingresso contiene elementi adiacenti duplicati, come avviene in 1 7 2 9 9 4 9.

Si tratta di un problema nuovo. Esaminiamo il ciclo che viene solitamente utilizzato per leggere un valore:

```
while (in.hasNextDouble())
{
    double input = in.nextDouble();
    ...
}
```

Come possiamo confrontare il valore appena letto con quello precedente? Ad ogni iterazione del ciclo, `input` contiene il valore appena letto, sovrascrivendo così il precedente.

La soluzione consiste nel memorizzare il valore precedente, in questo modo:

```
double input = 0;
while (in.hasNextDouble())
{
    double previous = input;
    input = in.nextDouble();
    if (input == previous)
    {
        System.out.println("Duplicate input");
    }
}
```

Rimane un problema: quando il ciclo viene eseguito per la prima volta, non esiste alcun valore precedente memorizzato in `input`. Si può risolvere questo problema effettuando una prima operazione di lettura all'esterno del ciclo:

```
double input = in.nextDouble();
while (in.hasNextDouble())
{
    double previous = input;
    input = in.nextDouble();
    if (input == previous)
    {
        System.out.println("Duplicate input");
    }
}
```



### Auto-valutazione

31. Quale somma totale viene calcolata dall'algoritmo visto nel Paragrafo 6.7.1 quando l'utente non fornisce alcun valore in ingresso?
32. Come si calcola la somma di tutti i valori positivi forniti in ingresso?
33. Qual è il valore di `position` quando, nell'algoritmo presentato nel Paragrafo 6.7.3, non viene trovata alcuna corrispondenza?
34. Cosa c'è di sbagliato nel ciclo seguente, che vorrebbe trovare la posizione del primo spazio nella stringa `str`?

```
boolean found = false;
for (int position = 0; !found && position < str.length(); position++)
{
    char ch = str.charAt(position);
    if (ch == ' ') { found = true; }
```

35. Come si trova la posizione dell'*ultimo* spazio in una stringa?
36. Come si comporta l'algoritmo del Paragrafo 6.7.6 quando l'utente non fornisce alcun dato? Come si può risolvere questo problema?

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi E6.6, E6.10 e E6.11, al termine del capitolo.



## Consigli pratici 6.1

### Scrivere un ciclo

In questa sezione analizzeremo, passo dopo passo, la procedura da seguire per realizzare un ciclo.

**Problema.** Acquisire dodici valori di temperatura (uno per ciascun mese dell'anno) e visualizzare il numero corrispondente al mese che ha la temperatura più alta. Ad esempio, in base ai dati riportati nel sito [worldclimate.com](http://worldclimate.com), le medie mensili della temperatura massima giornaliera nella Death Valley (la “Valle della Morte”) sono (in gradi Celsius e in ordine da gennaio a dicembre):

18.2 22.6 26.4 31.1 36.6 42.2 45.7 44.5 40.2 33.1 24.2 17.6

In questo caso, il mese con la temperatura più elevata (45.7 gradi Celsius) è luglio e il programma dovrebbe visualizzare il numero 7.

**Fase 1** Decidete ciò che va fatto *all'interno* del ciclo.

Ogni ciclo viene progettato per compiere ripetutamente alcune azioni, ad esempio:

- Acquisire un dato.
- Aggiornare un valore (come un saldo bancario o una somma).
- Incrementare un contatore.

Se non siete in grado di capire cosa debba essere fatto all'interno del ciclo, iniziate scrivendo le azioni che dovreste compiere se voleste risolvere il problema a mano. Ad esempio, nel caso dell'individuazione del mese avente la temperatura media più elevata, potreste scrivere:

Leggi il primo valore.

Leggi il secondo valore.

Se il secondo valore è maggiore del primo

Assegna tale secondo valore alla temperatura massima.

Assegna 2 al mese con la temperatura massima.

Leggi il terzo valore.

Se tale valore è maggiore del primo e del secondo

Assegna tale terzo valore alla temperatura massima.

Assegna 3 al mese con la temperatura massima.

Leggi il quarto valore.

Se tale valore è maggiore della temperatura massima vista finora

Assegna tale quarto valore alla temperatura massima.

Assegna 4 al mese con la temperatura massima.

...

Ora esaminate attentamente ciò che avete scritto e cercate di individuare un insieme di azioni che possano essere inserite nel corpo del ciclo e che siano adeguate per qualsiasi iterazione. La prima azione è facile:

Leggi il prossimo valore.

L'individuazione dell'azione successiva richiede un po' di astuzia. Nella descrizione abbiamo usato frasi come "maggiori del primo", "maggiori del primo e del secondo" e "maggiori della temperatura massima vista finora", ma dobbiamo riuscire a usare un'unica frase che funzioni in tutti i casi. È evidente che l'ultima frase, quella più generale, è quella che ci serve.

Analogamente, dobbiamo trovare un modo generale per assegnare un valore al "mese con la temperatura massima". Ci serve una variabile che tenga traccia del mese attualmente in esame, assumendo ordinatamente i valori da 1 a 12. Possiamo così esprimere la seconda azione del ciclo:

Se tale valore è maggiore della temperatura massima vista finora

Assegna tale valore alla temperatura massima.

Assegna il mese attuale al mese con la temperatura massima.

Mettendo tutto insieme, il nostro ciclo diventa:

Ripeti

Leggi il prossimo valore.

Se tale valore è maggiore della temperatura massima vista finora

Assegna tale valore alla temperatura massima.

Assegna il mese attuale al mese con la temperatura massima.

Incrementa di un'unità il mese attuale.

### Fase 2 Specificate la condizione che controlla il ciclo.

Quale obiettivo deve raggiungere il ciclo? Ecco alcuni esempi tipici:

- C'è un contatore che ha raggiunto il proprio valore finale?
- È stato acquisito l'ultimo valore in ingresso?
- C'è un valore che ha raggiunto una determinata soglia?

Nel nostro esempio vogliamo semplicemente che il valore del mese attuale arrivi a 12.

### Fase 3 Determinate di che tipo deve essere il ciclo.

Conosciamo due principali categorie di cicli. Un ciclo *controllato da un contatore* viene eseguito un numero determinato di volte, mentre in un ciclo *controllato da un evento* il numero di iterazioni non è noto a priori: il ciclo viene eseguito finché non accade un determinato evento.

I cicli controllati da contatore vengono solitamente realizzati con un enunciato `for`. Per gli altri cicli bisogna esaminare la condizione di terminazione: è necessario completare

l'esecuzione del corpo del ciclo prima di poter dire se questo è terminato? In tal caso scegliete un ciclo `do`, altrimenti usate un ciclo `while`.

A volte la condizione di terminazione di un ciclo diventa vera in un punto intermedio nel corpo del ciclo stesso: in tal caso si può usare una variabile booleana (spesso detta **flag**, *bandiera*, perché ha solo due “posizioni” diverse, “issata oppure no”) che specifichi quando è giunto il momento di uscire dal ciclo, secondo questo schema:

```
boolean done = false;
while (!done)
{
    fai alcune cose
    if (è stato fatto tutto ciò che doveva essere fatto)
    {
        done = true;
    }
    else
    {
        fai altre cose
    }
}
```

Riassumendo:

- Se conoscete a priori il numero di iterazioni del ciclo, usate un ciclo `for`.
- Altrimenti, se il corpo del ciclo deve essere eseguito almeno una volta, usate un ciclo `do`.
- Altrimenti, usate un ciclo `while`.

Nel nostro esempio leggiamo 12 valori di temperatura, per cui scegliamo un ciclo `for`.

#### Fase 4 Inizializzate le variabili con i valori che servono al primo ingresso nel ciclo.

Elencate tutte le variabili che dovete usare e aggiornare all'interno del ciclo e decidete come inizializzarle. I contatori vengono solitamente inizializzati a 0 o a 1, mentre le somme vanno inizializzate a 0.

Nel nostro esempio, le variabili sono:

**mese attuale**  
**temperatura massima**  
**mese con la temperatura massima**

Dobbiamo fare attenzione al valore iniziale per la variabile “temperatura massima”: non possiamo semplicemente usare il valore zero, perché il programma deve poter funzionare anche con i valori di temperatura tipici dell'Antartide, che possono anche essere tutti negativi.

Una buona idea: come valore iniziale della temperatura usiamo il primo valore acquisito in ingresso. Facendo così dobbiamo poi, ovviamente, ricordarci che ci rimangono da leggere soltanto 11 valori, con il “mese attuale” che inizia dal valore 2.

Conseguentemente dobbiamo inizializzare al valore 1 la variabile “mese con la temperatura massima” (in fin dei conti, in una città australiana difficilmente troveremo un mese che sia più caldo di gennaio).

**Fase 5** Elaborate il risultato dopo che il ciclo è terminato.

In molti casi il risultato cercato sarà semplicemente una delle variabili che vengono aggiornate nel corpo del ciclo (ad esempio, nel nostro programma il risultato è il valore finale della variabile “mese con la temperatura massima”), ma a volte il ciclo calcola valori che vanno utilizzati per il calcolo del risultato. Supponiamo, ad esempio, che ci venga chiesto di calcolare il valore medio delle temperature acquisite: il ciclo calcolerà la loro somma, non il loro valore medio, ma dopo la sua terminazione avremo a disposizione tutti i dati che servono per calcolare quest’ultimo, dividendo la somma dei valori per il numero di valori acquisiti.

Ecco, infine, il ciclo completo:

Leggi il primo valore e memorizzalo come temperatura massima.

mese con la temperatura massima = 1

Per mese attuale che va da 2 a 12

Leggi il prossimo valore.

Se tale valore è maggiore della temperatura massima vista finora

Assegna tale valore alla temperatura massima.

Assegna il mese attuale al mese con la temperatura massima.

**Fase 6** Eseguite il ciclo a mano in alcuni casi tipici.

Eseguite a mano il vostro codice, passo dopo passo, come visto nel Paragrafo 6.2. Scegliete alcuni valori di prova in modo che la loro elaborazione non sia troppo complessa: eseguire quattro o cinque volte il corpo del ciclo è solitamente sufficiente per individuare la maggior parte degli errori più frequenti. Fate attenzione, in modo particolare, al momento in cui si entra nel ciclo per la prima e per l’ultima volta.

A volte può essere ragionevole fare una piccola modifica per agevolare l’esecuzione manuale. Ad esempio, seguendo l’esecuzione del codice del programma che raddoppia l’investimento, può essere più comodo usare un tasso di interesse del 20%, invece del 5% previsto nel codice. Analogamente, nel seguire a mano l’esecuzione del codice che analizza le temperature, usate 4 valori invece di 12.

Supponiamo che i dati siano 22.6, 36.6, 44.5 e 24.2. La traccia dell’esecuzione, in questo caso, è la seguente:

meße attuale	valore acquisito	meße con temperatur massima	temperatur massima
X		X	22.6
Z	36.6	Z	36.6
Y	44.5	3	44.5
4	24.2		

Come si può vedere, al termine dell'esecuzione le variabili "mese con temperatura massima" e "temperatura massima" hanno i valori corretti.

#### Fase 7 Realizzate il ciclo in Java.

Riportiamo qui il ciclo relativo al nostro esempio. L'Esercizio E6.5, al termine del capitolo, vi chiederà di completare il programma.

```
double highestValue = in.nextDouble();
int highestMonth = 1;
for (int currentMonth = 2; currentMonth <= 12; currentMonth++)
{
    double nextValue = in.nextDouble();
    if (nextValue > highestValue)
    {
        highestValue = nextValue;
        highestMonth = currentMonth;
    }
}
System.out.println(highestMonth);
```



## Esempi completi 6.1

### Elaborazione di carte di credito

Quando si fanno acquisti online, molti siti Web chiedono di inserire il numero della carta di credito senza usare spazi né trattini, una cosa piuttosto scomoda che rende difficoltosa la verifica del numero da parte dell'utente. Sarebbe così difficile eliminare gli spazi e i trattini da una stringa? Assolutamente no, come vedrete in questo esempio.

**Problema.** Vogliamo rimuovere tutti gli spazi e i trattini presenti nella stringa creditCardNumber: ad esempio, se la stringa fosse "4123-5678-9012-3450", dovremmo trasformarla in "4123567890123450".

#### Fase 1 Decidete ciò che va fatto *all'interno* del ciclo.

Nel ciclo, ispezioniamo ciascun singolo carattere della stringa, in sequenza. Si può accedere al carattere in posizione *i* in questo modo:

```
char ch = creditCardNumber.charAt(i);
```

Se non è un trattino né uno spazio, passiamo al carattere successivo, altrimenti lo eliminiamo.

Ripeti

ch = carattere in posizione *i* in creditCardNumber

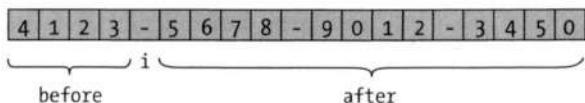
Se ch è uno spazio o un trattino

    Elimina il carattere da creditCardNumber

Altrimenti

    Incrementa i

Potreste chiedervi, meravigliati, come si possa eliminare un carattere da una stringa in Java. Per eliminare il carattere che si trova in posizione *i* basta concatenare due sottostringhe, quella che termina prima di *i* e quella che inizia dopo *i*.



```
String before = creditCardNumber.substring(0, i);
String after = creditCardNumber.substring(i + 1);
creditCardNumber = before + after;
```

Osservate che, dopo aver rimosso un carattere, *non* incrementiamo *i*: nell'esempio riportato nella figura, *i* valeva 4 prima dell'eliminazione e abbiamo rimosso il trattino che si trovava in posizione 4; alla prossima iterazione del ciclo, vogliamo esaminare nuovamente la posizione 4, che conterrà il carattere 5.

**Fase 2** Specificate la condizione che controlla il ciclo.

Il ciclo va eseguito finché l'indice *i* fa riferimento a una posizione valida, cioè

```
i < creditCardNumber.length()
```

**Fase 3** Determinate di che tipo deve essere il ciclo.

Non sappiamo a priori quante iterazioni del ciclo verranno eseguite, perché questo dipende dal numero di trattini e di spazi che troveremo, quindi sceglieremo un ciclo `while`. Perché non un ciclo `do`? Perché se ci troviamo a esaminare una stringa vuota (ad esempio, perché l'utente non ha fornito alcun numero di carta di credito), non vogliamo eseguire alcuna iterazione del ciclo.

**Fase 4** Iniziate le variabili con i valori che servono al primo ingresso nel ciclo.

È molto semplice: dobbiamo porre *i* = 0.

**Fase 5** Elaborate il risultato dopo che il ciclo è terminato.

In questo caso, il risultato prodotto è proprio la stringa che ci serve.

**Fase 6** Eseguite il ciclo a mano in alcuni casi tipici.

Ecco lo pseudocodice completo:

```
i = 0
Finché i < creditCardNumber.length()
    ch = carattere in posizione i in creditCardNumber
    Se ch è uno spazio o un trattino
        Elimina il carattere da creditCardNumber
    Altrimenti
        Incrementa i
```

Eseguire il ciclo a mano, su carta, con una stringa di 20 caratteri è un po' noioso, per cui lo facciamo con un esempio più breve.

creditCardNumber	i	ch
4-56-7	0	4
4-56-7	1	-
456-7	1	5
456-7	2	6
456-7	3	-
4567	3	7

**Fase 7** Realizzate il ciclo in Java.

Ecco il programma completo.

### File CCNumber.java

```

1  /**
2   * Elimina spazi e trattini da un numero di carta di credito.
3  */
4 public class CCNumber
5 {
6     public static void main(String[] args)
7     {
8         String creditCardNumber = "4123-5678-9012-3450";
9
10        int i = 0;
11        while (i < creditCardNumber.length())
12        {
13            char ch = creditCardNumber.charAt(i);
14            if (ch == ' ' || ch == '-')
15            {
16                // elimina il carattere in posizione i
17
18                String before = creditCardNumber.substring(0, i);
19                String after = creditCardNumber.substring(i + 1);
20                creditCardNumber = before + after;
21            }
22            else
23            {
24                i++;
25            }
26        }
27
28        System.out.println(creditCardNumber);
29    }
30 }
```

## 6.8 Cicli annidati

Quando il corpo di un ciclo contiene un altro ciclo, i due cicli si dicono annidati. La visualizzazione di una tabella, con righe e colonne, è un tipico esempio di utilizzo di cicli annidati.

Nel Paragrafo 5.4 avete visto come si possano annidare due enunciati `if`, uno dentro l'altro: in modo del tutto analogo, a volte le iterazioni più complesse richiedono un **ciclo annidato** (*nested loop*). Ad esempio, nell'elaborazione di tabelle l'esigenza di cicli annidati sorge quasi spontanea: un ciclo esterno esamina tutte le righe della tabella, mentre un ciclo interno elabora le colonne della riga in esame.

In questo paragrafo vedrete come visualizzare una tabella: per semplicità, visualizzeremo le potenze di  $x$ , in questo modo:

$x^1$	$x^2$	$x^3$	$x^4$
1	1	1	1
2	4	8	16
3	9	27	81
...	...	...	...
10	100	1000	10000

usando l'algoritmo descritto da questo pseudocodice:

```

Visualizza l'intestazione della tabella.
Per x che va da 1 a 10
    Visualizza una riga della tabella.
    Vai a capo.

```

Come si visualizza una riga della tabella? Bisogna scrivere un valore per ciascuno degli esponenti considerati e questo richiede un secondo ciclo:

```

Per n che va da 1 a 4
    Visualizza  $x^n$ .

```

Quest'ultimo ciclo deve essere inserito nel corpo del ciclo precedente: diciamo che questo ciclo più interno viene **annidato** (*nested*) dentro quello più esterno.

Nel ciclo esterno vengono visualizzate 10 righe e, per ciascun valore di  $x$ , il programma visualizza quattro colonne, usando il ciclo interno (come si può vedere nel diagramma di flusso della Figura 7): in totale, quindi, vengono visualizzati  $10 \times 4 = 40$  valori.

Ecco il programma completo. Si noti che usiamo due cicli anche per visualizzare l'intestazione della tabella, ma non sono annidati.

### File PowerTable.java

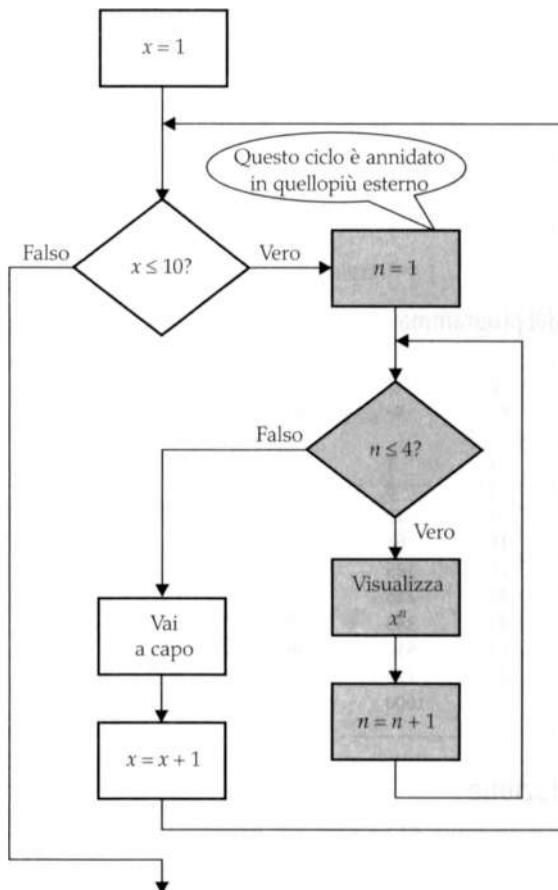
```

1  /**
2   * Questo programma visualizza una tabella con le potenze di x.
3  */
4  public class PowerTable
5  {

```

**Figura 7**

Diagramma di flusso  
di un ciclo annidato



```

6   public static void main(String[] args)
7   {
8       final int NMAX = 4;
9       final double XMAX = 10;
10
11      // visualizza l'intestazione della tabella
12
13      for (int n = 1; n <= NMAX; n++)
14      {
15          System.out.printf("%10d", n);
16      }
17      System.out.println();
18      for (int n = 1; n <= NMAX; n++)
19      {
20          System.out.printf("%10s", "x ");
21      }
22      System.out.println();
23
24      // visualizza il corpo della tabella
25
26      for (double x = 1; x <= XMAX; x++)
27      {

```

```

28         // visualizza una riga della tabella
29
30     for (int n = 1; n <= NMAX; n++)
31     {
32         System.out.printf("%10.0f", Math.pow(x, n));
33     }
34     System.out.println();
35 }
36 }
37 }

```

### Esecuzione del programma

1	2	3	4
x	x	x	x
1	1	1	1
2	4	8	16
3	9	27	81
4	16	64	256
5	25	125	625
6	36	216	1296
7	49	343	2401
8	64	512	4096
9	81	729	6561
10	100	1000	10000



### Auto-valutazione

37. Perché c'è un enunciato `System.out.println()` nel ciclo esterno ma non nel ciclo interno?
38. Come modifichereste il programma per fare in modo che visualizzi tutte le potenze di  $x$  che vanno da  $x^0$  a  $x^5$ ?
39. Con le modifiche apportate per rispondere alla domanda precedente, quanti valori vengono visualizzati?
40. Cosa visualizzano i seguenti cicli annidati?

```

for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < 4; j++)
    {
        System.out.print(i + j);
    }
    System.out.println();
}

```

41. Scrivete cicli annidati che visualizzino questa scacchiera di parentesi quadre:

```

[]][][]
[]][][
[]][][

```

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R6.30, E6.17 e E6.19, al termine del capitolo.

**Tabella 3**

Esempi di cicli annidati

Cicli annidati	Visualizza	Spiegazione
<pre>for (i = 1; i &lt;= 3; i++) {     for (j = 1; j &lt;= 4; j++) { Visualizza "*" }     System.out.println(); }</pre>	**** **** ****	Visualizza 3 righe, ciascuna con 4 asterischi.
<pre>for (i = 1; i &lt;= 4; i++) {     for (j = 1; j &lt;= 3; j++) { Visualizza "*" }     System.out.println(); }</pre>	*** *** *** ***	Visualizza 4 righe, ciascuna con 3 asterischi.
<pre>for (i = 1; i &lt;= 4; i++) {     for (j = 1; j &lt;= i; j++) { Visualizza "*" }     System.out.println(); }</pre>	*	Visualizza 4 righe aventi lunghezza 1, 2, 3 e 4.
<pre>for (i = 1; i &lt;= 3; i++) {     for (j = 1; j &lt;= 5; j++)     {         if (j % 2 == 0) { Visualizza "*" }         else { Visualizza "-" }     }     System.out.println(); }</pre>	-*- -*- -*-	Visualizza asterischi nelle colonne pari e trattini nelle colonne dispari.
<pre>for (i = 1; i &lt;= 3; i++) {     for (j = 1; j &lt;= 5; j++)     {         if (i % 2 == j % 2) { Visualizza "*" }         else { Visualizza " " }     }     System.out.println(); }</pre>	* * * * * * * *	Visualizza uno schema a scacchiera.



## Esempi completi 6.2

### Manipolare i pixel in un'immagine

Un'immagine digitale è costituita da *pixel*, ciascuno dei quali è un piccolo quadrato di un determinato colore. In questo esempio completo useremo una classe, `Picture`, dotata di metodi che consentono di leggere (“caricare”, *load*) un'intera immagine e di accedere ai suoi singoli pixel.

**Problema.** Il programma deve convertire un'immagine nella sua immagine “in negativo”, trasformando il bianco in nero, il ciano in rosso e così via: si ottiene un'immagine “in negativo”, simile a quelle che venivano prodotte dalle vecchie macchine fotografiche.

L'implementazione della classe `Picture` usa la libreria di Java dedicata all'elaborazione di immagini e ciò va ben al di là degli obiettivi di questo libro, per cui presentiamo qui soltanto le porzioni più rilevanti della sua interfaccia pubblica:

```

public class Picture
{
    . . .
    /**
     * Restituisce la larghezza di questa immagine.
     * @return la larghezza
    */
    public int getWidth() { . . . }

    /**
     * Restituisce l'altezza di questa immagine.
     * @return l'altezza
    */
    public int getHeight() { . . . }

    /**
     * Legge e carica un'immagine da una sorgente assegnata.
     * @param source la sorgente dell'immagine. Se inizia con http://
     *               si tratta di un URL, altrimenti è il nome di un file.
    */
    public void load(String source) { . . . }

    /**
     * Restituisce il colore di un pixel.
     * @param x l'indice di colonna (fra 0 e getWidth() - 1)
     * @param y l'indice di riga (fra 0 e getHeight() - 1)
     * @return il colore del pixel che si trova nella posizione (x, y)
    */
    public Color getColorAt(int x, int y) { . . . }

    /**
     * Assegna il colore a un pixel.
     * @param x l'indice di colonna (fra 0 e getWidth() - 1)
     * @param y l'indice di riga (fra 0 e getHeight() - 1)
     * @param c il colore da assegnare al pixel che si trova nella posizione (x, y)
    */
    public void setColorAt(int x, int y, Color c) { . . . }

    . . .
}

```

Vediamo ora come si può convertire in negativo un'immagine. Per calcolare il colore "in negativo" corrispondente a un determinato oggetto di tipo `Color` possiamo procedere così:

```

Color original = . . .;
Color negative = new Color(
    255 - original.getRed(),
    255 - original.getGreen(),
    255 - original.getBlue()
);

```

Vogliamo, naturalmente, effettuare questa elaborazione per ciascun pixel che compone l'immagine.

Per far ciò, possiamo usare una di queste due strategie:

Per ogni riga  
Per ogni pixel della riga  
Elabora il pixel.

oppure

Per ogni colonna  
Per ogni pixel della colonna  
Elabora il pixel.

Dato che la nostra classe usa le coordinate  $x$  e  $y$  per accedere ai pixel, sembra più naturale usare la seconda strategia (nel Capitolo 7 vedrete array bidimensionali che usano, per l'accesso, coordinate di tipo riga e colonna, rendendo preferibile la prima strategia).

Per esaminare tutte le colonne, la coordinata  $x$  parte da zero. Dato che il numero di colonne è `pic.getWidth()`, usiamo questo ciclo:

```
for (int x = 0; x < pic.getWidth(); x++)
```

Dopo aver individuato una colonna, dobbiamo scandire tutte i valori della coordinata  $y$  all'interno di essa, partendo da zero. Dato che il numero di righe è `pic.getHeight()`, i nostri cicli annidati sono:

```
for (int x = 0; x < pic.getWidth(); x++)
{
    for (int y = 0; y < pic.getHeight(); y++)
    {
        Color original = pic.getColorAt(x, y);
        . . .
    }
}
```

Ecco, infine, il programma che risolve il nostro problema di elaborazione di immagini:

## File Negative.java

```

17         pic.setColorAt(x, y, negative);
18     }
19 }
20 }
21 }

```

## 6.9 Applicazione: numeri casuali e simulazioni

In una simulazione si usa il calcolatore per simulare un'attività.

Un **programma di simulazione** usa il computer per simulare un'attività del mondo reale (o di un mondo immaginario). Si usano simulazioni per prevedere i cambiamenti climatici, per analizzare il traffico, per scambiare azioni in borsa e per molte altre applicazioni, tanto in campo scientifico quanto economico e industriale. In molte simulazioni vengono usati cicli per modificare lo stato di un sistema e per osservarne i cambiamenti: nei prossimi paragrafi vedrete alcuni esempi.

### 6.9.1 Generare numeri casuali

Anche se molti eventi del mondo reale sono difficilmente prevedibili con precisione assoluta, a volte ne conosciamo piuttosto bene l'andamento medio. Ad esempio, un negoziante potrebbe sapere, basandosi sull'esperienza, che arriva un cliente ogni cinque minuti circa: ovviamente si tratta di una media, nel senso che i clienti non arrivano a intervalli regolari di cinque minuti. Per costruire un modello accurato dell'arrivo dei clienti bisogna tener conto di fluttuazioni casuali: come pensate che sia possibile eseguire una tale simulazione al calcolatore?

Si può introdurre casualità in un programma invocando un generatore di numeri casuali.

La classe `Random` della libreria di Java realizza un **generatore di numeri casuali** (*random number generator*) che produce numeri apparentemente casuali. Per generare numeri casuali, si costruisce un oggetto di tipo `Random` e se ne invoca uno dei metodi:

Metodo	Restituisce
<code>nextInt(n)</code>	un numero intero casuale, compreso fra zero (incluso) e <code>n</code> (escluso)
<code>nextDouble()</code>	un numero casuale in virgola mobile, compreso fra zero (incluso) e uno (escluso)

In questo modo, ad esempio, si può simulare il lancio di un dado:

```

Random generator = new Random();
int d = 1 + generator.nextInt(6);

```

L'invocazione `generator.nextInt(6)` restituisce un numero intero compreso tra 0 e 5 (estremi inclusi): aggiungendo uno si ottiene un numero compreso tra 1 e 6.

Ecco, quindi, un programma che simula dieci lanci di un dado.

#### File Die.java

```

1 import java.util.Random;
2
3 /**
4  * Questa classe rappresenta un dato che, quando viene

```

```
5     lanciato, visualizza una delle sue facce a caso.
6 */
7 public class Die
8 {
9     private Random generator;
10    private int sides;
11
12    /**
13     Costruisce un dado avente un dato numero di facce.
14     @param s il numero di facce, ad esempio 6 per un dado comune
15    */
16    public Die(int s)
17    {
18        sides = s;
19        generator = new Random();
20    }
21
22    /**
23     Simula il lancio di un dado.
24     @return la faccia del dado
25    */
26    public int cast()
27    {
28        return 1 + generator.nextInt(sides);
29    }
30 }
```

### File DieSimulator.java

```
1 /**
2  * Questo programma simula dieci lanci di un dado.
3 */
4 public class DieSimulator
5 {
6     public static void main(String[] args)
7     {
8         Die d = new Die(6);
9         final int TRIES = 10;
10        for (int i = 1; i <= TRIES; i++)
11        {
12            int n = d.cast();
13            System.out.print(n + " ");
14        }
15        System.out.println();
16    }
17 }
```

### Esecuzione del programma (un esempio)

6 5 6 3 2 6 3 4 4 1

### Esecuzione del programma (un altro esempio)

3 2 2 1 6 5 3 4 1 2

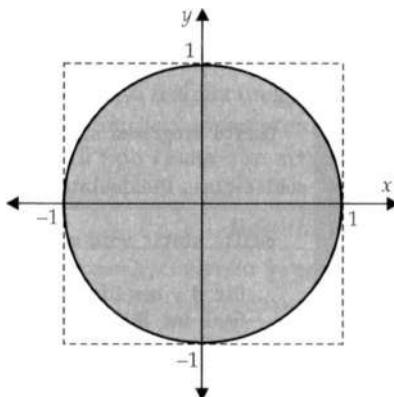
Come si può notare, ogni esecuzione del programma genera una diversa sequenza di lanci simulati del dado.

In realtà, i numeri non sono completamente casuali, ma sono estratti da sequenze di numeri molto lunghe che non si ripetono per un lungo intervallo. Tali sequenze sono calcolate mediante formule piuttosto semplici e i numeri si comportano proprio come numeri casuali: per questo motivo vengono spesso chiamati **numeri pseudocasuali** (*pseudorandom numbers*). La generazione di buone sequenze di numeri, che si comportino come sequenze veramente casuali, è un problema importante e studiato a fondo nell'informatica, tuttavia non vogliamo indagare ulteriormente e ci limiteremo a usare i numeri casuali prodotti dalla classe `Random`.

### 6.9.2 Il metodo Monte Carlo

Il metodo Monte Carlo è un'ingegnosa strategia per trovare soluzioni approssimate per problemi che non possono essere risolti in modo esatto (il metodo prende il nome dal famoso casinò di Monte Carlo). Ecco un esempio tipico: è difficile calcolare il numero  $\pi$ , ma lo si può approssimare abbastanza bene con la simulazione seguente.

Simulate il lancio di una freccetta all'interno di un quadrato circoscritto a un cerchio di raggio 1. Questo è facile: basta generare casualmente due valori compresi tra -1 e 1, uno per la coordinata  $x$  e uno per la coordinata  $y$ .



Se il punto generato si trova all'interno del cerchio, lo contiamo come un "bersaglio colpito" (*hit*): questo accade se  $x^2 + y^2 \leq 1$ . Dato che i nostri lanci sono completamente casuali, ci aspettiamo che il rapporto *hits / tries* (cioè il numero di "bersagli colpiti", *hits*, diviso per il numero di "tentativi", *tries*) sia approssimativamente uguale al rapporto tra le aree del cerchio e del quadrato, cioè  $\pi/4$ . Conseguentemente, la nostra stima del valore di  $\pi$  è  $4 \cdot \text{hits} / \text{tries}$ . Questo metodo fornisce una stima di  $\pi$  senza usare altro che non sia la semplice aritmetica.

Per generare un numero casuale in virgola mobile, compreso tra -1 e 1, si può fare così:

```
double r = generator.nextDouble(); // 0 ≤ r < 1
double x = -1 + 2 * r; // -1 ≤ x < 1
```

Dato che  $r$  varia tra 0 (compreso) e 1 (escluso),  $x$  varia tra  $-1 + 2 \cdot 0 = -1$  (compreso) e  $-1 + 2 \cdot 1 = 1$  (escluso). Per la nostra applicazione, il fatto che  $x$  non assuma mai il valore 1 è ininfluente: i punti che soddisfano l'equazione  $x = 1$  si trovano su un segmento avente area uguale a zero.

Ecco il programma che effettua la simulazione:

### File MonteCarlo.java

```
1 import java.util.Random;
2
3 /**
4     Stima il valore di pigreco simulando lanci di una freccetta in un quadrato.
5 */
6 public class MonteCarlo
7 {
8     public static void main(String[] args)
9     {
10         final int TRIES = 10000;
11         Random generator = new Random();
12
13         int hits = 0;
14         for (int i = 1; i <= TRIES; i++)
15         {
16             // Genera due numeri casuali compresi tra -1 e 1
17
18             double r = generator.nextDouble();
19             double x = -1 + 2 * r;
20             r = generator.nextDouble();
21             double y = -1 + 2 * r;
22
23             // Controlla se il punto si trova sul cerchio di raggio unitario
24
25             if (x * x + y * y <= 1) { hits++; }
26
27         /*
28             Il rapporto hits / tries è circa uguale al rapporto tra
29             l'area del cerchio e l'area del quadrato, che è pigreco / 4
30
31         */
32
33         double piEstimate = 4.0 * hits / TRIES;
34         System.out.println("Estimate for pi: " + piEstimate);
35     }
36 }
```

### Esecuzione del programma (un esempio)

Estimate for pi: 3.1504



### Auto-valutazione

42. Come si può simulare il lancio di una moneta usando la classe `Random`?
43. Come si può simulare la scelta casuale di una carta da gioco da un mazzo completo?

44. Come modifichereste il programma `DieSimulator` per simulare il lancio di una coppia di dadi?
45. In molti giochi si lancia una coppia di dadi per ottenere un valore compreso tra 2 e 12. Cosa c'è di sbagliato nella seguente simulazione del lancio di una coppia di dadi?

```
int sum = 2 + generator.nextInt(11);
```

46. Come si genera un numero casuale in virgola mobile che sia  $\geq 0$  e  $< 100$ ?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R6.31, E6.8 e E6.22, al termine del capitolo.

## 6.10 Usare un *debugger*

Come avrete certamente ormai capito, raramente i programmi funzionano perfettamente al primo tentativo e spesso individuare gli errori può essere molto frustrante. Naturalmente potete inserire enunciati che visualizzino messaggi opportuni, per poi eseguire il programma e tentare di analizzare quanto viene prodotto. Se tale analisi non indica chiaramente il problema, potrebbe essere necessario aggiungere o eliminare comandi di visualizzazione ed eseguire nuovamente il programma. È un meccanismo che può richiedere molto tempo.

I moderni ambienti di sviluppo contengono speciali programmi, detti **debugger**, che aiutano a localizzare gli errori, permettendo di seguire l'esecuzione di un programma. Potete interrompere e far proseguire il programma, osservando il contenuto delle variabili nei momenti in cui l'esecuzione è temporaneamente sospesa. Durante ciascuna pausa potete scegliere quali variabili esaminare e quanti enunciati eseguire prima dell'interruzione successiva.

Alcuni credono che i debugger siano strumenti utili soltanto per rendere pigri i programmatori. Effettivamente qualcuno ha l'abitudine di scrivere programmi senza fare molta attenzione, per poi sistemarli usando il debugger, ma la maggioranza dei programmatori tenta onestamente di scrivere il miglior programma possibile, prima di provare a eseguirlo mediante il debugger. Questi ultimi programmatori sanno che il debugger, sebbene sia più comodo degli enunciati di visualizzazione aggiunti al codice, non è esente da costi, perché occorre tempo per impostare e per eseguire una efficace sessione di questa attività.

Nella programmazione reale non potete evitare di usare il debugger: più grandi sono i programmi, più difficile sarà correggerli inserendo semplicemente enunciati di visualizzazione. Scoprirete che il tempo investito nell'apprendimento dell'utilizzo di un debugger verrà ampiamente ripagato durante la vostra carriera di programmatori.

Come avviene per i compilatori, anche i debugger differiscono ampiamente da un sistema all'altro. In alcuni casi sono alquanto rudimentali e obbligano il programmatore a memorizzare una piccola serie di comandi misteriosi, in altri hanno, invece, un'intuitiva interfaccia a finestre. La Figura 8 mostra il debugger dell'ambiente di sviluppo Eclipse, che si può reperire gratuitamente nel sito Web della Eclipse Foundation ([eclipse.org](http://eclipse.org)). Anche altri ambienti di sviluppo integrati, come BlueJ, Netbeans e IntelliJ IDEA, contengono un debugger.

Un *debugger* è un programma che si può usare per eseguire un altro programma e analizzare il suo comportamento durante l'esecuzione.

Dovrete scoprire autonomamente come preparare un programma per l'attività di *debugging* (che consiste, appunto, nell'utilizzo di un debugger) e come avviare il debugger nel vostro sistema di programmazione. Se usate un ambiente di sviluppo integrato, che contenga un editor, un compilatore e un debugger, normalmente questa operazione è molto facile: semplicemente, progettate il programma nel modo abituale e selezionate un comando di menu per avviare il debugger. In altri sistemi dovete invece costruire manualmente un'apposita versione del vostro programma adatta per il debugging e, quindi, invocare il debugger.

Una volta avviato il debugger, potete fare molto lavoro con tre soli comandi: "imposta un punto di arresto" (detto *breakpoint*), "esegui la riga di codice successiva" (esecuzione *single step* o "passo dopo passo") e "ispeziona la variabile ...". I nomi da digitare sulla tastiera o le selezioni del mouse che servono per eseguire questi comandi differiscono molto nei diversi debugger, ma tutti mettono a disposizione queste operazioni fondamentali. Dovete scoprirlne il modo di utilizzo consultando la documentazione oppure chiedendo a qualcuno che ha già usato il debugger.

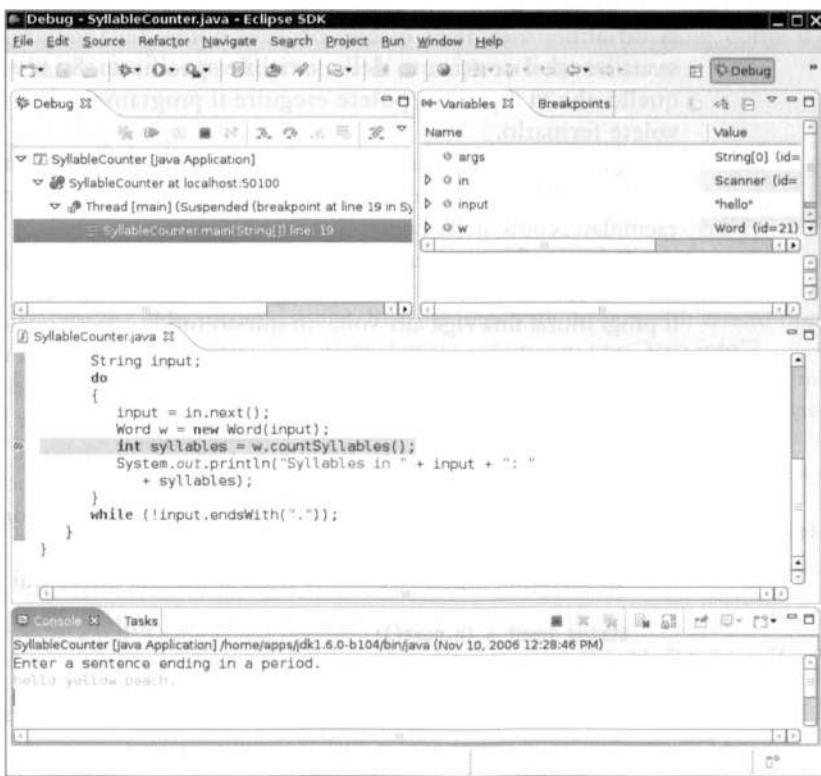
Quando fate partire il debugger, il programma viene eseguito a velocità piena fino al raggiungimento di un punto di arresto, in corrispondenza del quale l'esecuzione viene sospesa e viene visualizzato, come in Figura 8, il breakpoint che ha provocato l'arresto. A questo punto potete ispezionare le variabili che vi interessano ed eseguire il programma una riga alla volta, oppure continuare l'esecuzione del programma a velocità piena fino al raggiungimento del breakpoint successivo. Quando il programma termina, termina anche l'esecuzione del debugger.

Quando il debugger esegue  
il programma, l'esecuzione viene  
interrotta ogni volta che viene  
raggiunto un punto di arresto.

Quando il debugger esegue  
il programma, l'esecuzione viene  
interrotta ogni volta che viene  
raggiunto un punto di arresto.

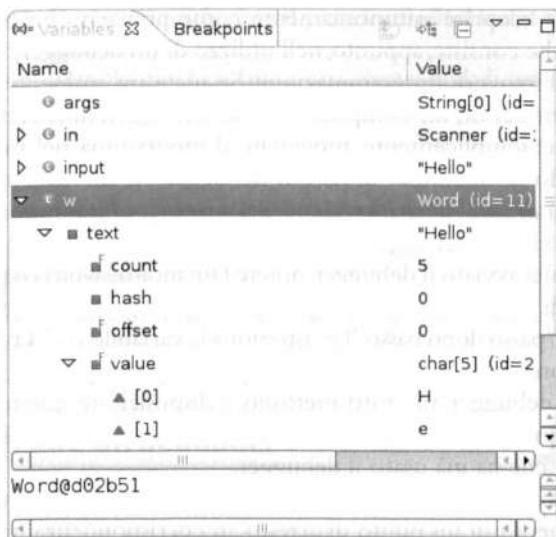
**Figura 8**

Il debugger fermo  
a un punto di arresto



**Figura 9**

Ispezione di variabili



I punti di arresto rimangono attivi finché non vengono esplicitamente rimossi, per cui dovrete eliminare di tanto in tanto i punti di arresto che non vi servono più.

Dopo che il programma si è fermato, potete osservare i valori memorizzati nelle variabili. Ancora una volta il metodo per selezionare le variabili dipende dallo specifico debugger utilizzato. Alcuni visualizzano sempre una finestra che contiene i valori delle variabili locali, in altri dovrete eseguire un comando del tipo “esamina la variabile” e digitare o selezionare il nome della variabile, dopodiché il debugger visualizzerà il contenuto della variabile specificata. Se tutte le variabili contengono quello che vi aspettate, potete eseguire il programma fino al punto successivo in cui volete fermarlo.

Il comando *single step* esegue il programma una riga alla volta.

Quando si esaminano oggetti, spesso è necessario fornire uno specifico comando per “aprire” l’oggetto stesso: una volta aperto l’oggetto, potrete vedere le sue variabili di esemplare, come nella Figura 9.

Eseguire il programma fino a un breakpoint vi porta velocemente al punto che vi interessa, ma non sapete come il programma vi sia arrivato. In alternativa, potete eseguire il programma una riga alla volta: in questo modo sapete quale sia il flusso di esecuzione del programma, ma può darsi che ci voglia molto tempo per arrivare nel punto che vi interessa. Il comando “single step” esegue la riga di codice attuale e si interrompe prima di eseguire la riga successiva. La maggior parte dei debugger hanno due tipi di comandi “single step”: uno chiamato “step into”, cioè “fai un passo all’interno”, che fa procedere l’esecuzione di una riga per volta all’interno dei metodi invocati, e uno chiamato “step over”, cioè “fai un passo scavalcando”, che esegue le invocazioni di metodi senza arrestarsi al loro interno.

Ad esempio, supponete che la linea attuale sia quella qui evidenziata:

```
String input = in.next();
Word w = new Word(input);
int syllables = w.countSyllables();
System.out.println("Syllables in " + input + ": " + syllables);
```

Quando eseguite un comando di tipo “step over” in una riga in cui è presente l’invocazione di un metodo, l’esecuzione procede fino alla riga successiva:

```
String input = in.next();
Word w = new Word(input);
int syllables = w.countSyllables();
System.out.println("Syllables in " + input + ": " + syllables);
```

Se, invece, eseguite un comando di tipo “step into”, vi troverete nella prima riga del metodo `countSyllables`.

```
public int countSyllables()
{
    int count = 0;
    int end = text.length() - 1;
    ...
}
```

Dovreste *entrare* in un metodo (“step into”) per verificare che svolga correttamente il suo compito, mentre dovreste *scavalcarlo* (“step over”) se sapete già che funziona bene.

Infine, quando il programma termina completamente la propria esecuzione, si conclude anche la sessione di debug: per eseguire il programma nuovamente, bisogna far ripartire il debugger.

Il debugger può essere uno strumento veramente utilissimo per individuare ed eliminare gli errori presenti nei programmi, ma non può sostituire un buon progetto e una programmazione attenta: se il debugger non trova errori, non significa che il programma non ne abbia. Ricordate che l’attività di debugging, come il collaudo, può soltanto testimoniare la presenza di errori, non la loro assenza.



## Auto-valutazione

47. Durante una sessione di debugging avete raggiunto un’invocazione di `System.out.println`: è più opportuno eseguire uno “step into” oppure uno “step over”?
48. Durante una sessione di debugging avete raggiunto l’inizio di un metodo che contiene un paio di cicli e volete scoprire quale valore venga calcolato e restituito dal metodo al termine della sua esecuzione. Bisogna impostare un punto di arresto oppure eseguire il metodo passo dopo passo, dopo esservi entrati con uno “step into”?
49. Durante una sessione di debugging avete scoperto che una variabile ha un valore diverso da quello previsto. In che modo si può tornare indietro per scoprire quando la variabile ha cambiato valore?
50. Quando si usa un debugger, è opportuno aggiungere al programma enunciati che visualizzino i valori delle variabili?
51. Invece di usare un debugger, è sempre possibile seguire a mano, passo dopo passo, l’esecuzione di un programma?

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi R6.33, R6.34 e R6.35, al termine del capitolo.



## Consigli pratici 6.2

### L'attività di debugging

Ora conoscete i meccanismi dell'attività di debugging, ma tutta questa conoscenza può ancora lasciarvi disarmati quando eseguite il debugger per esaminare un programma difettoso. Ecco alcune strategie che potete utilizzare per individuare gli errori e le loro cause.

#### Fase 1 Riproducete l'errore

Mentre collaudate il vostro programma, vi accorgrete che talvolta il codice fa qualcosa di sbagliato: fornisce un valore errato in uscita, sembra stampare qualcosa completamente a caso, entra in un ciclo infinito o interrompe bruscamente la propria esecuzione. Scoprite esattamente come riprodurre questo comportamento. Quali numeri avete digitato? Dove avete premuto il pulsante del mouse?

Eseguite nuovamente il programma: digitate esattamente gli stessi numeri e premete il pulsante del mouse negli stessi punti (o il più vicino possibile). Se il programma esibisce lo stesso comportamento, è il momento di eseguire il debugger per studiare questo particolare problema. I debugger vanno bene per analizzare anomalie precise, mentre non sono molto utili per studiare, in generale, il comportamento di un programma.

#### Fase 2 Semplificate l'errore

Prima di eseguire il debugger, è utile spendere alcuni minuti cercando di individuare una situazione più semplice che provochi lo stesso errore. Ottenete comunque un comportamento errato del programma, anche inserendo parole più brevi o numeri più semplici? Se ciò accade, usate tali valori durante il debugging.

#### Fase 3 “Dividere per vincere” (*divide and conquer*)

Per identificare il punto in cui il programma fallisce, usate la tecnica del “dividere per vincere”.

Ora che siete alle prese con una precisa anomalia, volete avvicinarvi il più possibile all'errore. Durante l'attività di debugging è fondamentale la localizzazione del punto preciso nel codice che produce il comportamento errato: trovare un errore può essere difficile, ma, una volta che l'avete trovato, eliminarlo è solitamente la parte più facile.

Immaginate che il programma termini l'esecuzione con una divisione per zero. Dal momento che un tipico programma contiene molte operazioni di divisione, spesso non è possibile impostare punti di interruzione per tutte. Piuttosto, usate una tecnica che consiste nel *dividere per vincere* (“divide and conquer”). Eseguite con il debugger i metodi invocati dal `main`, senza entrare al loro interno (cioè usate la modalità *step over*). A un certo punto apparirà l'anomalia e saprete quale metodo contiene l'errore: è l'ultimo chiamato da `main` prima della brusca terminazione del programma. Eseguite nuovamente il debugger e tornate nel `main` alla stessa riga, poi entrate all'interno del metodo (*step into*), ripetendo il procedimento.

Alla fine individuerete la riga che contiene la divisione sbagliata. Può darsi che il codice riveli chiaramente perché il denominatore non è corretto, altrimenti dovrete trovare il punto dove questo viene calcolato. Sfortunatamente, nel debugger non potete andare *a ritroso*, ma dovete eseguire nuovamente il programma e portarvi al punto in cui viene calcolato il denominatore.

**Fase 4** Siate consapevoli di ciò che il programma dovrebbe fare

Durante il debugging confrontate il contenuto delle variabili con i valori che avevate previsto.

Il debugger vi mostra cosa *fa* il programma, ma voi dovete sapere cosa *dovrebbe fare*, altrimenti non sarete in grado di scoprire gli errori. Prima di seguire passo dopo passo l'esecuzione di un ciclo, chiedetevi quante iterazioni vi aspettate che vengano eseguite. Prima di ispezionare il valore di una variabile, chiedetevi cosa prevedete di osservare. Se non ne avete idea, prendetevi un po' di tempo e riflettete prima di agire. Munitevi di una calcolatrice tascabile per eseguire calcoli in modo indipendente dal programma ed esamineate la variabile soltanto quando sapete quale deve essere il suo valore corretto. Se il valore è quello previsto, dovete cercare l'errore più avanti, altrimenti potreste aver trovato qualcosa. Controllate nuovamente i vostri calcoli: se siete sicuri che il valore previsto sia corretto, scoprite perché il programma giunge a una conclusione diversa.

In molti casi gli errori presenti in un programma sono il risultato di sviste banali, come l'errore per scarto di uno nell'espressione che determina la terminazione di un ciclo. Piuttosto spesso, tuttavia, i programmi cadono su errori di calcolo: magari si pensava di sommare due numeri, ma, per sbaglio, si è scritto il codice per sottrarli. I programmi (come i problemi che accadono nel mondo reale) non fanno alcuno sforzo speciale per assicurarsi che tutti i calcoli si svolgano con numeri interi di piccola entità, per cui vi ritroverete a fare alcune operazioni con numeri elevati oppure con complicate cifre in virgola mobile. A volte questi calcoli si possono evitare, domandandosi semplicemente se una certa quantità deve essere positiva oppure se deve superare un certo valore, esaminando poi le variabili per verificarlo.

**Fase 5** Controllate tutti i dettagli

Anche se quando fate il debugging di un programma spesso avete già un'idea della natura del problema, mantenete una mentalità aperta e osservate tutti i particolari. Quali strani messaggi vengono visualizzati? Perché il programma intraprende un'azione diversa e inaspettata? Questi dettagli hanno la loro importanza. Quando eseguite una sessione di debugging, siete veramente nella posizione di un detective che deve cogliere qualsiasi indizio disponibile.

Se, mentre state focalizzando la vostra attenzione su un problema, notate un'altra anomalia, non limitatevi a pensare di tornarvi sopra più tardi, perché potrebbe essere la vera causa del problema di cui vi state interessando. È meglio prendere un appunto per il primo problema, correggere l'anomalia appena trovata, e poi tornare al problema originario.

**Fase 6** Siate certi di avere ben compreso un errore prima di correggerlo

Quando si scopre che un ciclo esegue troppe iterazioni si ha la forte tentazione di "mettere una toppa", magari sottraendo un'unità da una variabile, in modo che quel particolare problema non si ripresenti più. Una soluzione rapida di questo tipo ha probabilità schiaccianti di creare problemi in qualche altro punto del codice. In realtà, dovete capire a fondo come andrebbe scritto il programma, prima di applicare un rimedio.

Talvolta succede di scoprire un errore dopo l'altro e, conseguentemente, di inserire correzioni dopo correzioni, mentre si gira semplicemente intorno al problema. Generalmente questo è il sintomo dell'esistenza di un problema importante nella logica del programma: con il debugger si può fare poco, occorre rivedere la struttura del programma e organizzarlo in modo diverso.



## Esempi completi 6.3

### Un esempio di sessione di *debugging*

Questo esempio completo presenta una situazione di debugging realistico, occupandosi di una classe, `Word`, il cui compito principale è quello di contare le sillabe presenti in una parola.

**Problema.** La classe `Word` individua le sillabe usando la regola seguente, valida per la lingua inglese:

Ogni gruppo di vocali (a, e, i, o, u, y) adiacenti fa parte di una sillaba: ad esempio, il gruppo “ea” in “peach” appartiene a un’unica sillaba, mentre le vocali “e...o” in “yellow” danno luogo a due sillabe. Tuttavia, una “e” alla fine di una parola non genera una sillaba. Ogni parola ha almeno una sillaba, anche se le regole precedenti forniscono un conteggio nullo.

Infine, quando viene costruita una parola a partire da una stringa, eventuali caratteri all’inizio o alla fine della stringa che non siano lettere vengono eliminati. Questo è utile quando si leggono i dati in ingresso usando il metodo `next` della classe `Scanner`: le stringhe acquisite possono contenere segni di punteggiatura, ma non vogliamo che questi facciano parte della parola.

L’obiettivo è quello di individuare gli errori presenti in questo programma e correggerli.

Ecco il codice sorgente della classe, contenente un paio di errori.

### File Word.java

```

1  /**
2   * Questa classe descrive parole di un documento.
3   * La classe contiene un paio di errori.
4  */
5  public class Word
6  {
7      private String text;
8
9      /**
10       * Costruisce una parola eliminando caratteri iniziali e finali
11       * che non siano lettere, come i segni di punteggiatura.
12       * @param s la stringa di ingresso
13     */
14    public Word(String s)
15    {
16        int i = 0;
17        while (i < s.length() && !Character.isLetter(s.charAt(i)))
18        {
19            i++;
20        }
21        int j = s.length() - 1;
22        while (j > i && !Character.isLetter(s.charAt(j)))

```

```
23     {
24         j--;
25     }
26     text = s.substring(i, j);
27 }
28
29 /**
30  Restituisce il testo della parola, dopo aver rimosso
31  i caratteri iniziali e finali che non siano lettere.
32  @return il testo della parola
33 */
34 public String getText()
35 {
36     return text;
37 }
38
39 /**
40  Conta le sillabe nella parola.
41  @return il numero di sillabe
42 */
43 public int countSyllables()
44 {
45     int count = 0;
46     int end = text.length() - 1;
47     if (end < 0) { return 0; } // la stringa vuota non ha sillabe
48
49     // una 'e' alla fine della parola non conta come vocale
50     char ch = text.charAt(end);
51     if (ch == 'e' || ch == 'E') { end--; }

52     boolean insideVowelGroup = false;
53     for (int i = 0; i <= end; i++)
54     {
55         ch = text.charAt(i);
56         String vowels = "aeiouyAEIOUY";
57         if (vowels.indexOf(ch) >= 0)
58         {
59             // ch è una vocale
60             if (!insideVowelGroup)
61             {
62                 // inizia un nuovo gruppo di vocali
63                 count++;
64                 insideVowelGroup = true;
65             }
66         }
67     }
68
69     // ogni parola ha almeno una sillaba
70     if (count == 0) { count = 1; }
71
72     return count;
73 }
74 }
```

Ecco una semplice classe di prova. Digitate una frase e verrà visualizzato il conteggio delle sillabe di tutte le sue parole.

### File SyllableCounter.java

```

1 import java.util.Scanner;
2
3 /**
4  * Questo programma conta le sillabe di tutte le parole di una frase.
5 */
6 public class SyllableCounter
7 {
8     public static void main(String[] args)
9     {
10         Scanner in = new Scanner(System.in);
11
12         System.out.println("Enter a sentence ending in a period.");
13
14         String input;
15         do
16         {
17             input = in.next();
18             Word w = new Word(input);
19             int syllables = w.countSyllables();
20             System.out.println("Syllables in " + input + ": " + syllables);
21         }
22         while (!input.endsWith("."));
23     }
24 }
```

Fornendo questi dati in ingresso:

Hello yellow peach.

viene visualizzato quanto segue:

```

Syllables in Hello: 1
Syllables in yellow: 1
Syllables in peach.: 1
```

che non è molto promettente.

Prima di tutto impostate un punto di arresto nella prima linea del metodo `countSyllables` della classe `Word`, poi fate partire il programma, che chiederà i dati in ingresso, ricevuti i quali si arresterà al breakpoint che avete impostato.

Innanzitutto il metodo `countSyllables` controlla l'ultimo carattere della parola per vedere se è una lettera 'e'. Vediamo se questo funziona correttamente: eseguite il programma fino alla riga 51 (osservate la Figura 12).

Ora ispezionate la variabile `ch`. Come potete vedere nella Figura 13, questo particolare debugger visualizza autonomamente tutte le variabili locali e di esemplare; se il vostro non lo fa, può darsi che dobbiate ispezionare `ch` manualmente. Come potete vedere, `ch` contiene stranamente il valore '1'. Osservate il codice sorgente: la variabile `end` è stata impostata al valore `text.length() - 1`, corrispondente all'ultima posizione nella stringa `text`, e `ch` è il carattere che si trova in tale posizione.

Proseguendo nell'analisi, scoprirete che `end` vale 3, non 4, come invece vi aspettereste, e `text` contiene la stringa "Hell", non "Hello". Quindi, non c'è da meravigliarsi che `countSyllables` restituisca il valore 1. Per risolvere il problema dobbiamo guardare altrove: sembra che l'errore sia nel costruttore di `Word`.

**Figura 12**

Attività di debugging  
nel metodo countSyllables

```

public int countSyllables()
{
    int count = 0;
    int end = text.length() - 1;
    if (end < 0) { return 0; } // The empty string has no syllables

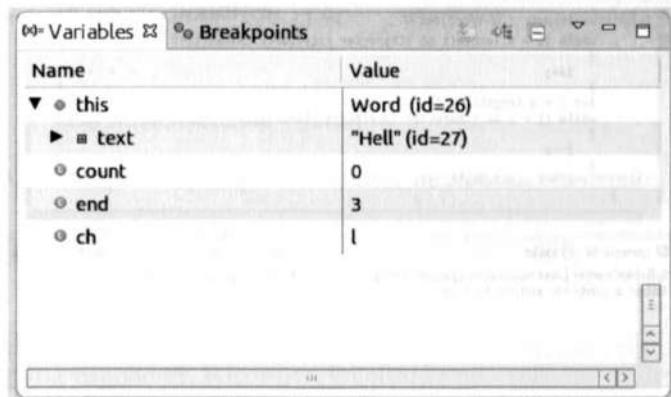
    // An e at the end of the word doesn't count as a vowel
    char ch = text.charAt(end);
    if (ch == 'e' || ch == 'E') { end--; }

    boolean insideVowelGroup = false;
}

```

**Figura 13**

Valori delle variabili locali  
e di esemplare



Sfortunatamente, un debugger non può tornare indietro nel tempo, per cui dovete interrompere il programma, impostare un breakpoint nel costruttore di `Word` e far ripartire il debugger, fornendo di nuovo in ingresso gli stessi dati. Il debugger si arresterà all'inizio del costruttore di `Word`, che imposta i valori di due variabili, `i` e `j`, ignorando tutti i caratteri che non siano lettere all'inizio e alla fine della stringa di ingresso. Come si vede nella Figura 14, impostate un breakpoint dopo la fine del secondo ciclo, in modo da poter ispezionare i valori di `i` e `j`.

A questo punto, l'ispezione di `i` e `j` mostra che `i` vale 0 e che `j` vale 4. Ciò ha senso, perché non ci sono segni di punteggiatura da ignorare. Quindi, perché a `text` viene assegnata la stringa "Hell"? Ricordate che il metodo `substring` considera le posizioni fino al proprio secondo parametro *senza includerlo*, per cui l'invocazione corretta dovrebbe essere

```
text = s.substring(i, j + 1);
```

Questo è un tipico errore "per scarto di uno".

Correggete questo errore, compilate il programma e provate di nuovo i tre casi di prova. Otterrete la seguente visualizzazione in uscita:

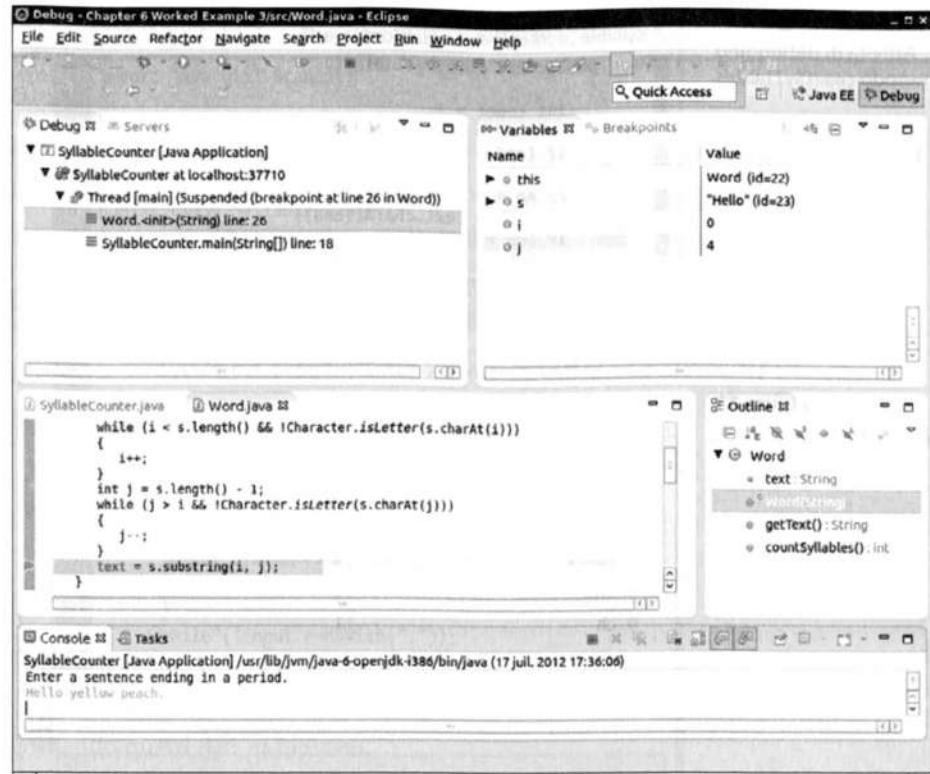
```

Syllables in Hello: 1
Syllables in yellow: 1
Syllables in peach.: 1

```

Figura 14

Attività di debugging nel costruttore di Word



Come si può notare, c'è ancora un problema. Eliminate tutti i breakpoint e impostatene uno nuovo nel metodo `countSyllables`. Fate partire il debugger e fornite in ingresso la stringa "Hello.". Quando il debugger si arresta al punto prestabilito, iniziate a eseguire il programma passo dopo passo (cioè un'istruzione per volta, in modalità *single step*) all'interno del metodo. Ecco il codice del ciclo che conta le sillabe:

```
boolean insideVowelGroup = false;
for (int i = 0; i <= end; i++)
{
    ch = text.charAt(i);
    String vowels = "aeiouyAEIOUY";
    if (vowels.indexOf(ch) >= 0)
    {
        // ch è una vocale
        if (!insideVowelGroup)
        {
            // inizia un nuovo gruppo di vocali
            count++;
            insideVowelGroup = true;
        }
    }
}
```

Nella prima iterazione del ciclo il debugger non esegue l'enunciato `if`: ciò è corretto, perché la prima lettera, 'H' non è una vocale. Nella seconda iterazione il debugger entra

nell'enunciato `if`, come previsto, perché la seconda lettera, 'e', è una vocale. La variabile `insideVowelGroup` viene impostata a `true` e il contatore delle vocali viene incrementato. Nella terza iterazione l'enunciato `if` viene nuovamente ignorato, perché la lettera 'l' non è una vocale, ma nella quinta iterazione accade qualcosa di strano. La lettera 'o' è una vocale e l'enunciato `if` viene eseguito, ma il secondo enunciato `if` viene ignorato e la variabile `count` non viene nuovamente incrementata.

Perché? La variabile `insideVowelGroup` è ancora `true`, anche se il primo gruppo di vocali era terminato quando è stata esaminata la consonante 'l'. La lettura di una consonante dovrebbe riportare `insideVowelGroup` al valore `false`. Questo è un errore logico più subdolo, ma non inusuale quando si progetta un ciclo che tiene traccia dello stato di una elaborazione. Per correggerlo, interrompete il debugger e aggiungete la clausola seguente:

```
if (vowels.indexOf(ch) >= 0)
{
    ...
}
else insideVowelGroup = false;
```

Ora compilate ed eseguite di nuovo il collaudo, ottenendo:

```
Syllables in Hello: 2
Syllables in yellow: 2
Syllables in peach.: 1
```

Il programma è ora privo di errori? Questa non è una domanda alla quale il debugger possa rispondere. Ricordate: il collaudo può soltanto evidenziare la presenza di errori, non la loro assenza.



## Computer e società 6.2

### Il primo bug

Secondo la leggenda, il primo *bug* ("Insetto") trovato in un calcolatore fu quello rinvenuto nel Mark II, un enorme computer elettromeccanico presso la Harvard University. Fu veramente causato da un insetto (*bug*), una falena intrappolata in un interruttore a relè.

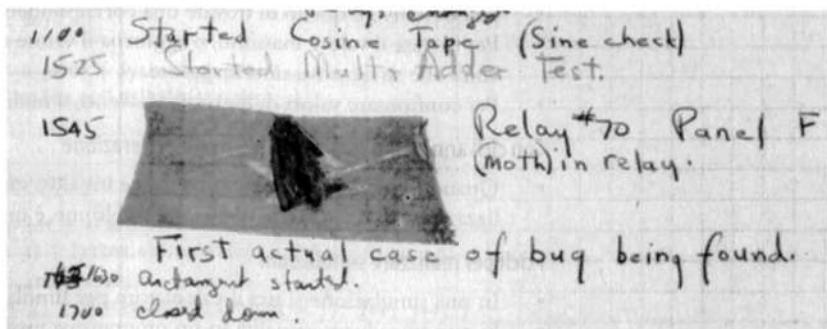
In realtà, dalla nota che l'operatore lasciò nel registro, accanto alla falena, sembra che all'epoca il termine "bug" venisse già impiegato nell'uso quotidiano come sinonimo di "errore nel computer".

Maurice Wilkes, un pioniere della ricerca informatica, scrisse: "Per

qualche motivo, alla Moore School e in seguito, si è sempre pensato che non vi sarebbero state particolari difficoltà a scrivere programmi corretti. Posso, invece, ricordarmi il momento esatto in cui mi fu chiaro che gran

parte della mia vita futura sarebbe stata dedicata alla ricerca degli errori presenti nei miei programmi".

### Il primo bug



## Riepilogo degli obiettivi di apprendimento

### Il flusso di esecuzione di un ciclo

- Un ciclo esegue ripetutamente un blocco di codice fintantoché una specifica condizione risulta vera.
- Gli errori per scarto di uno sono molto comuni nella programmazione dei cicli: per evitarli, ragionate con cura sui casi più semplici.

### Analisi manuale del comportamento di un programma, passo dopo passo

- Per tenere traccia dell'esecuzione di un programma, lo si simula a mano, eseguendo le istruzioni passo dopo passo e annotando i valori delle variabili.
- L'esecuzione a mano, passo dopo passo, vi può aiutare a comprendere il funzionamento di un algoritmo che non conoscete.
- L'esecuzione a mano, passo dopo passo, può evidenziare errori nel codice o nello pseudocodice.

### Il ciclo `for` per realizzare cicli a contatore

- Si usa un ciclo `for` quando una variabile varia da un valore iniziale a un valore finale con un incremento o decremento costante.

### Scegliere tra un ciclo `while` e un ciclo `do`

- Si usa un ciclo `do` quando il corpo del ciclo deve essere eseguito almeno una volta.

### Cicli che acquisiscono sequenze di dati

- Un valore sentinella segnala la fine di un insieme di dati, ma non fa parte dei dati stessi.
- Per controllare un ciclo si può anche usare una variabile booleana: le si assegna un valore prima di entrare nel ciclo e le si assegna il valore opposto quando si vuole che il ciclo termini.
- Si usa la redirezione di input per acquisire dati da un file e la redirezione di output per scrivere in un file i risultati prodotti dal programma.

### Usare lo storyboard per pianificare le interazioni con l'utente

- Uno storyboard è una sequenza di immagini con didascalie: una per ciascuna fase di un'azione.
- Lo sviluppo di uno storyboard vi aiuta a capire quali siano i dati che il vostro programma deve acquisire e le risposte che deve fornire.

### Gli algoritmi più comuni che si realizzano con cicli

- Per calcolare il valore medio di una sequenza di valori, si accumula la somma totale mentre si contano i valori.
- Per contare i valori che soddisfano una determinata condizione, si controllano uno ad uno e si incrementa un contatore ogni volta che il controllo va a buon fine.
- Se l'obiettivo è quello di trovare una corrispondenza, si termina il ciclo non appena la si trova.
- Per trovare il valore massimo, si aggiorna il valore massimo riscontrato fino a quel punto ogni volta che se ne trova uno maggiore.
- Per confrontare valori di ingresso adiacenti, si memorizza in una variabile il valore precedente.

### Con cicli annidati si realizzano più livelli di iterazione

- Quando il corpo di un ciclo contiene un altro ciclo, i due cicli si dicono annidati. La visualizzazione di una tabella, con righe e colonne, è un tipico esempio di utilizzo di cicli annidati.

### I cicli per realizzare simulazioni

- In una simulazione si usa il calcolatore per simulare un'attività.
- Si può introdurre casualità in un programma invocando un generatore di numeri casuali.

### Con un debugger si individuano errori durante l'esecuzione

- Un *debugger* è un programma che si può usare per eseguire un altro programma e analizzare il suo comportamento durante l'esecuzione.
- Potete usare efficacemente il debugger comprendendo a fondo tre concetti: punti di arresto (*breakpoint*), esecuzione passo dopo passo (*single step*) e ispezione di variabili.
- Quando il debugger esegue un programma, l'esecuzione viene interrotta ogni volta che viene raggiunto un punto di arresto.
- Il comando *single step* esegue il programma una riga alla volta.
- Per identificare il punto in cui il programma fallisce, usate la tecnica del “dividere per vincere”.
- Durante il debugging confrontate il contenuto delle variabili con i valori che avevate previsto.

## Elementi di libreria presentati in questo capitolo

```
java.util.Random  
    nextDouble  
    nextInt
```

## Esercizi di riepilogo e approfondimento

★ **R6.1.** Date le variabili

```
String stars = "*****";  
String stripes = "=====";
```

cosa visualizzano i cicli seguenti?

- ```
int i = 0;  
while (i < 5)  
{  
    System.out.println(stars.substring(0, i));  
    i++;  
}
```
- ```
int i = 0;  
while (i < 5)  
{  
    System.out.print(stars.substring(0, i));  
    System.out.println(stripes.substring(i, 5));  
    i++;  
}
```
- ```
int i = 0;  
while (i < 10)  
{  
    if (i % 2 == 0) { System.out.println(stars); }  
    else { System.out.println(stripes); }  
}
```

★ **R6.2.** Cosa visualizzano i cicli seguenti?

- ```
int i = 0; int j = 10;  
while (i < j) { System.out.println(i + " " + j); i++; j--; }
```
- ```
int i = 0; int j = 10;  
while (i < j) { System.out.println(i + j); i++; j++; }
```

\* **R6.3.** Cosa visualizzano i cicli seguenti?

- ```
int result = 0;
for (int i = 1; i <= 10; i++) { result = result + i; }
System.out.println(result);
```
- ```
int result = 1;
for (int i = 1; i <= 10; i++) { result = i - result; }
System.out.println(result);
```
- ```
int result = 1;
for (int i = 5; i > 0; i--) { result = result * i; }
System.out.println(result);
```
- ```
int result = 1;
for (int i = 1; i <= 10; i = i * 2) { result = result * i; }
System.out.println(result);
```

\* **R6.4.** Scrivete un ciclo while che visualizzi:

- tutti i numeri minori di n che sono quadrati perfetti (ad esempio, se n vale 100, visualizza 0 1 4 9 16 25 36 49 64 81);
- tutti i numeri positivi minori di n che sono divisibili per 10 (ad esempio, se n vale 100, visualizza 10 20 30 40 50 60 70 80 90);
- tutti i numeri minori di n che sono potenze di 2 (ad esempio, se n vale 100, visualizza 1 2 4 8 16 32 64).

\*\* **R6.5.** Scrivete un ciclo che calcoli:

- la somma di tutti i numeri pari compresi tra 2 e 100 (estremi inclusi);
- la somma di tutti i numeri compresi tra 1 e 100 (estremi inclusi) che sono quadrati perfetti;
- la somma di tutti i numeri dispari compresi tra a e b (estremi inclusi);
- la somma di tutte le cifre dispari del numero n (ad esempio, se n vale 32677, la somma descritta è 3 + 7 + 7 = 17).

\* **R6.6.** Scrivete le tabelle relative al tracciamento a mano dell'esecuzione dei cicli seguenti:

- ```
int i = 0; int j = 10; int n = 0;
while (i < j) { i++; j--; n++; }
```
- ```
int i = 0; int j = 0; int n = 0;
while (i < 10) { i++; n = n + i + j; j++; }
```
- ```
int i = 10; int j = 0; int n = 0;
while (i > 0) { i--; j++; n = n + i - j; }
```
- ```
int i = 0; int j = 10; int n = 0;
while (i != j) { i = i + 2; j = j - 2; n++; }
```

\* **R6.7.** Cosa visualizzano i cicli seguenti?

- ```
for (int i = 1; i < 10; i++) { System.out.print(i + " "); }
```
- ```
for (int i = 0; i < 10; i += 2) { System.out.print(i + " "); }
```
- ```
for (int i = 10; i > 1; i--) { System.out.print(i + " "); }
```
- ```
for (int i = 0; i < 10; i++) { System.out.print(i + " "); }
```
- ```
for (int i = 1; i < 10; i = i * 2) { System.out.print(i + " "); }
```
- ```
for (int i = 1; i < 10; i++) { if (i % 2 == 0) { System.out.print(i + " "); } }
```

\* **R6.8.** Cos'è un ciclo infinito? Nel vostro computer, in che modo potete terminare un programma che sta eseguendo un ciclo infinito?

- \* **R6.9.** Nell'ipotesi che i valori in ingresso siano 4 7 -2 -5 0, eseguite a mano lo pseudocodice dell'Esercizio E6.7, tenendone traccia su carta.
- \*\* **R6.10.** Cos'è un errore "per scarto di uno"? Fornite un esempio tratto dalla vostra esperienza di programmazione.
- \* **R6.11.** Cos'è un valore sentinella? Fornite una regola semplice che dica quando l'utilizzo di un valore sentinella di tipo numerico è appropriato.
- \* **R6.12.** Quali enunciati di ciclo esistono in Java? Fornite regole semplici che dicano quando è più opportuno usare ciascun tipo di ciclo.
- \*\* **R6.13.** Quante iterazioni eseguono i cicli seguenti, se la variabile *i* non viene modificata nel corpo del ciclo?
  - a. `for (int i = 1; i <= 10; i++) . . .`
  - b. `for (int i = 0; i < 10; i++) . . .`
  - c. `for (int i = 10; i > 0; i--) . . .`
  - d. `for (int i = -10; i <= 10; i++) . . .`
  - e. `for (int i = 10; i >= 0; i++) . . .`
  - f. `for (int i = -10; i <= 10; i = i + 2) . . .`
  - g. `for (int i = -10; i <= 10; i = i + 3) . . .`

- \*\* **R6.14.** Scrivete lo pseudocodice per un programma che visualizzi un calendario come questo:

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| Su | M  | T  | W  | Th | F  | Sa |
|    | 1  | 2  | 3  | 4  |    |    |
| 5  | 6  | 7  | 8  | 9  | 10 | 11 |
| 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| 19 | 20 | 21 | 22 | 23 | 24 | 25 |
| 26 | 27 | 28 | 29 | 30 | 31 |    |

- \* **R6.15.** Scrivete lo pseudocodice per un programma che visualizzi una tabella di conversione tra gradi Celsius e Fahrenheit come questa:

| Celsius |     | Fahrenheit |
|---------|-----|------------|
| 0       |     | 32         |
| 10      |     | 50         |
| 20      |     | 68         |
| ...     | ... | ...        |
| 100     |     | 212        |

- \* **R6.16.** Scrivete lo pseudocodice per un programma che legga i dati di uno studente (nome e cognome, seguiti da una sequenza di punteggi di valutazione e dal valore sentinella -1), per poi visualizzare il suo punteggio medio. Poi, tenendone traccia su carta, eseguite a mano il programma con questi dati di ingresso:

Harry Morgan 94 71 86 95 -1

- \*\* **R6.17.** Scrivete lo pseudocodice per un programma che legga i dati relativi a una sequenza di studenti, per poi visualizzare il punteggio totale di ciascuno. I dati relativi a uno studente sono il nome e il cognome, seguiti da una sequenza di punteggi di valutazione e dal valore sentinella -1. La sequenza di studenti viene terminata dalla parola END. Ecco un esempio di sequenza di dati in ingresso:

```

Harry Morgan 94 71 86 95 -1
Sally Lin 99 98 100 95 90 -1
END

```

Successivamente, tenendone traccia su carta, eseguite a mano il programma con tali dati di ingresso.

- ★ **R6.18.** Riscrivete il seguente ciclo `for` sotto forma di ciclo `while`.

```

int s = 0;
for (int i = 1; i <= 10; i++)
{
    s = s + i;
}

```

- ★ **R6.19.** Riscrivete il seguente ciclo `do` sotto forma di ciclo `while`.

```

int n = in.nextInt();
double x = 0;
double s;
do
{
    s = 1.0 / (1 + n * n);
    n++;
    x = x + s;
}
while (s > 0.01);

```

- ★ **R6.20.** Tenendone traccia su carta, eseguite a mano i cicli seguenti:

- a. int s = 1;  
int n = 1;  
while (s < 10) { s = s + n; }  
n++;
- b. int s = 1;  
for (int n = 1; n < 5; n++) { s = s + n; }
- c. int s = 1;  
int n = 1;  
do  
{  
 s = s + n;  
 n++;  
}  
while (s < 10 \* n);

- ★ **R6.21.** Cosa visualizzano i cicli seguenti? Trovate la risposta senza usare il computer, eseguendoli a mano e tenendo traccia dell'esecuzione su carta.

- a. int s = 1;  
for (int n = 1; n <= 5; n++)  
{  
 s = s + n;  
 System.out.print(s + " ");  
}
- b. int s = 1;  
for (int n = 1; s <= 10; System.out.print(s + " "))  
{  
 n = n + 2;

```

        s = s + n;
    }
c. int s = 1;
int n;
for (n = 1; n <= 5; n++)
{
    s = s + n;
    n++;
}
System.out.print(s + " " + n);

```

- \* **R6.22.** Cosa visualizzano i frammenti di codice seguenti? Trovate la risposta senza usare il computer, eseguendoli a mano e tenendo traccia dell'esecuzione su carta.

```

a. int n = 1;
for (int i = 2; i < 5; i++) { n = n + i; }
System.out.print(n);

b. int i;
double n = 1 / 2;
for (i = 2; i < 5; i++) { n = n + 1.0 / i; }
System.out.print(i);

c. double x = 1;
double y = 1;
int i = 0;
do
{
    y = y / 2;
    x = x + y;
    i++;
}
while (x < 1.8);
System.out.print(i);

d. double x = 1;
double y = 1;
int i = 0;
while (y >= 1.5)
{
    x = x / 2;
    y = x + y;
    i++;
}
System.out.print(i);

```

- \*\* **R6.23.** Fornite un esempio di ciclo `for` in cui i limiti simmetrici siano la scelta più naturale. Fornite un altro esempio di ciclo `for` in cui siano preferibili limiti asimmetrici.

- \* **R6.24.** Aggiungete allo storyboard del programma di conversione visto nel Paragrafo 6.6 una scheda che descriva uno scenario in cui l'utente fornisce in ingresso unità di misura incompatibili.
- \* **R6.25.** Nel Paragrafo 6.6 abbiamo deciso di mostrare all'utente, all'interno del prompt, un elenco di tutte le unità di misura valide. Se il programma consentisse l'utilizzo di un numero di unità di misura molto più elevato, questo approccio non sarebbe praticabile. Progettate una scheda, da aggiungere allo storyboard, che illustri un approccio alternativo: se l'utente fornisce un'unità sconosciuta, viene visualizzato un elenco di tutte le unità note al programma.
- \* **R6.26.** Modificate lo storyboard del Paragrafo 6.6 in modo che mostri un menu per chiedere all'utente se vuole convertire una misura, vedere le informazioni di ausilio all'utilizzo del pro-

gramma (*program help*) o terminare l'esecuzione. Il menu deve essere visualizzato all'inizio del programma, dopo aver convertito una sequenza di valori e dopo aver visualizzato un messaggio d'errore.

- \* **R6.27.** Disegnate il diagramma di flusso di un programma che effettui conversioni da un'unità di misura a un'altra, come descritto nel Paragrafo 6.6.
- \*\* **R6.28.** Nel Paragrafo 6.7.5, il codice che trova il minimo e il massimo valore tra quelli forniti in ingresso inizializza le variabili `largest` e `smallest` usando il primo valore acquisito. Perché non si possono inizializzare al valore zero?
- \* **R6.29.** Cosa sono i cicli annidati? Fornite un esempio in cui solitamente si usano cicli annidati.
- \*\* **R6.30.** Questi cicli annidati

```
for (int i = 1; i <= height; i++)
{
    for (int j = 1; j <= width; j++) { System.out.print("*"); }
    System.out.println();
}
```

visualizzano un rettangolo di larghezza (`width`) e altezza (`height`) assegnate, come questo:

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

Scrivete un *unico* ciclo che visualizzi lo stesso rettangolo.

- \*\* **R6.31.** Immaginate di dove progettare un gioco educativo che insegni ai bambini a leggere l'orologio. Come pensate di generare valori casuali per le ore e i minuti?
- \*\*\* **R6.32.** Nella simulazione di un viaggio, Harry farà visita a uno dei suoi amici, che abitano in tre stati diversi: ha dieci amici in California, tre in Nevada e due nello Utah. Come pensate di generare un numero casuale, compreso tra 1 e 3, che indichi lo stato di destinazione, con una probabilità proporzionale al numero di amici di Harry che abitano in quello stato?
- \* **R6.33 (collaudo).** Spiegate le differenze fra queste operazioni di debugging:
  - Procedere con l'esecuzione all'interno di un metodo ("step into")
  - Procedere con l'esecuzione senza entrare in un metodo ("step over")
- \*\* **R6.34 (collaudo).** Spiegate in dettaglio come usare il vostro debugger per esaminare le informazioni contenute in un oggetto di tipo `String`.
- \*\* **R6.35 (collaudo).** Spiegate in dettaglio come usare il vostro debugger per esaminare le informazioni contenute in un oggetto di tipo `Rectangle`.
- \*\* **R6.36 (collaudo).** Spiegate in dettaglio come usare il vostro debugger per esaminare il saldo contenuto in un oggetto di tipo `BankAccount`.
- \*\* **R6.37 (collaudo).** Spiegate la strategia "dividere per vincere", utilizzata per avvicinarsi a un errore usando il debugger.

## Esercizi di programmazione

- ★ **E6.1.** Scrivete un programma che legga il saldo iniziale di un investimento e un tasso di interesse annuo, per poi visualizzare il numero di anni necessari perché l'investimento raggiunga un saldo uguale a un milione di dollari.
- ★ **E6.2.** Scrivete programmi che, usando cicli, calcolino:
  - a. la somma di tutti i numeri pari compresi tra 2 e 100 (estremi inclusi);
  - b. la somma di tutti i numeri compresi tra 1 e 100 (estremi inclusi) che siano quadrati perfetti;
  - c. tutte le potenze di 2, da  $2^0$  a  $2^{20}$ ;
  - d. la somma di tutti i numeri dispari compresi tra a e b (estremi inclusi), dove a e b sono valori acquisiti in ingresso;
  - e. la somma di tutte le cifre dispari di un numero acquisito in ingresso (se, ad esempio, il numero è 32677, la somma da calcolare è  $3 + 7 + 7 = 17$ ).
- ★★ **E6.3.** Scrivete programmi che leggano una sequenza di numeri interi e visualizzino:
  - a. il valore minimo e il valore massimo tra quelli acquisiti;
  - b. il numero di valori pari e il numero di valori dispari tra quelli acquisiti;
  - c. le somme parziali dei numeri acquisiti, calcolate e visualizzate dopo ogni acquisizione; se, ad esempio, i valori in ingresso sono 1 7 2 9, il programma deve visualizzare 1, 8 (= 1 + 7), 10 (= 1 + 7 + 2), 19 (= 1 + 7 + 2 + 9);
  - d. i valori adiacenti duplicati; se, ad esempio, i valori acquisiti sono 1 3 3 4 5 5 6 6 6 3, il programma deve visualizzare 3 5 6.
- ★★ **E6.4.** Scrivete programmi che leggano una riga di dati in ingresso sotto forma di stringa e visualizzino:
  - a. le sole lettere maiuscole della stringa;
  - b. a partire dalla seconda lettera della stringa, una lettera viene visualizzata e l'altra no, alternativamente;
  - c. la stringa con tutte le vocali sostituite da un carattere di sottolineatura (*underscore*);
  - d. il numero di vocali presenti nella stringa;
  - e. le posizioni di tutte le vocali presenti nella stringa.
- ★★ **E6.5.** Completate il programma progettato nella sezione Consigli pratici 6.1, che legga dodici valori di temperatura e visualizzi il numero corrispondente al mese con la temperatura più elevata.
- ★★ **E6.6.** Scrivete un programma che (visualizzando un unico messaggio) chieda all'utente di fornire in ingresso un insieme di valori in virgola mobile e, poi, visualizzi:
  - la media dei valori;
  - il valore minore;
  - il valore maggiore;
  - la dinamica dei valori (*range*), cioè la differenza tra il valore maggiore e il valore minore.

Il programma deve usare un'apposita classe `DataSet` per rappresentare l'insieme dei dati acquisiti, dotata del metodo

```
public void add(double value)
```

per aggiungere un valore all'insieme, oltre ai metodi `getAverage`, `getSmallest`, `getLargest` e `getRange`, che risolvono, nell'ordine, i quattro problemi posti.

- \* **E6.7.** Traducete in un programma Java la seguente descrizione in pseudocodice, che trova il valore minimo in un insieme di dati acquisiti in ingresso.

```

primo valore = True.
Finché è stato acquisito con successo un nuovo valore
    Se primo valore è True
        valore minimo = valore acquisito.
        primo valore = False.
    Altrimenti se valore acquisito < valore minimo
        valore minimo = valore acquisito.
Visualizza valore minimo.

```

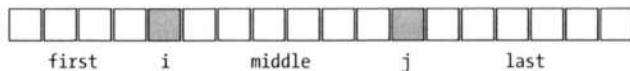
- \*\*\* **E6.8.** Traducete in un programma Java la seguente descrizione in pseudocodice, che permuta in modo casuale i caratteri di una stringa.

```

Leggi una parola e memorizzala in word.
Ripeti un numero di volte pari a word.length()
    Scegli a caso una posizione i in word, che non sia l'ultima posizione.
    Scegli a caso una posizione j in word, con j > i.
    Scambia le lettere che si trovano nelle posizioni i e j.
Visualizza word.

```

Per scambiare le lettere, costruite le tre sottostringhe (`first`, `middle` e `last`) indicate nella figura.



Quindi, sostituite la stringa originaria con:

```
first + word.charAt(j) + middle + word.charAt(i) + last
```

- \* **E6.9.** Scrivete un programma che legga una parola e visualizzi ciascuno dei suoi caratteri da solo su una riga diversa, ordinatamente. Se, ad esempio, l'utente digita la stringa "Harry", il programma deve visualizzare:

```
H
a
r
r
y
```

- \*\* **E6.10.** Scrivete un programma che legga una parola e la visualizzi al contrario. Se, ad esempio, l'utente digita la stringa "Harry", il programma deve visualizzare:

```
yrrah
```

- \* **E6.11.** Scrivete un programma che legga una parola e visualizzi il numero di vocali presenti in essa (per questo esercizio assumete che le vocali siano a e i o u y). Se, ad esempio, l'utente digita la stringa "Harry", il programma deve visualizzare il messaggio: 2 vowels.

- \*\*\* **E6.12.** Scrivete un programma che legga una parola e visualizzi tutte le sue sottostringhe, ordinate per lunghezza crescente. Se, ad esempio, l'utente digita la stringa "rum", il programma deve visualizzare:

```
r
u
m
ru
um
rum
```

- ★ E6.13. Scrivete un programma che visualizzi tutte le potenze di 2, da  $2^0$  a  $2^{20}$ .
- ★★ E6.14. Scrivete un programma che legga un numero intero (`number`) e visualizzi le cifre della sua rappresentazione binaria, in ordine inverso; visualizzate per prima cosa il resto della divisione per due, `number % 2`, poi sostituite il numero con `number / 2`; continuate fino a quando il numero diventa 0. Se, ad esempio, l'utente digita il numero 13, il programma deve visualizzare:

```
1
0
1
1
```

- ★ E6.15. Usando la classe `Picture` presentata nella sezione Esempi completi 6.2, applicate l'effetto "tramonto" (*sunset*) a un'immagine, aumentando in ciascun pixel il valore del rosso del 30 per cento (senza superare il valore massimo, 255).
- ★★ E6.16. Usando la classe `Picture` presentata nella sezione Esempi completi 6.2, applicate l'effetto "telescopio" (*telescope*) a un'immagine, facendo diventare neri tutti i pixel che si trovano all'esterno di un cerchio, il cui centro deve coincidere con il centro dell'immagine e il cui raggio deve essere il 40 per cento della dimensione minore dell'immagine (larghezza o altezza).



- ★ E6.17. Scrivete un programma che visualizzi una tavola pitagorica, come questa:

|     |    |    |    |    |    |    |    |    |     |
|-----|----|----|----|----|----|----|----|----|-----|
| 1   | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10  |
| 2   | 4  | 6  | 8  | 10 | 12 | 14 | 16 | 18 | 20  |
| 3   | 6  | 9  | 12 | 15 | 18 | 21 | 24 | 27 | 30  |
| ... |    |    |    |    |    |    |    |    |     |
| 10  | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |

- ★★ E6.18. Scrivete un programma che legga un numero intero,  $n$ , e visualizzi, usando asterischi, un quadrato pieno e uno con il solo contorno, uno di fianco all'altro, con il lato che misura  $n$ . Se, ad esempio, l'utente fornisce il numero 5, il programma deve visualizzare:

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

- \*\* E6.19. Scrivete un programma che legga un numero intero,  $n$ , e visualizzi, usando asterischi, un rombo pieno il cui lato abbia lunghezza  $n$ . Se, ad esempio, l'utente fornisce il numero 4, il programma deve visualizzare:

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

- \*\* E6.20 (**economia**). *Conversione di valuta.* Scrivete un programma che per prima cosa chieda all'utente di fornire il prezzo odierno di un dollaro statunitense in yen giapponesi, per poi leggere ripetutamente valori in dollari statunitensi, convertendoli in yen, fino all'arrivo del valore sentinella, che è uno zero.
- \*\* E6.21 (**economia**). Scrivete un programma che per prima cosa chieda all'utente di fornire il prezzo odierno di un dollaro statunitense in yen giapponesi, per poi leggere ripetutamente valori in dollari statunitensi, convertendoli in yen, fino all'arrivo del valore sentinella, che è uno zero. Successivamente, il programma legge una sequenza di valori in yen e li converte i dollari: anche questa seconda sequenza viene terminata da uno zero.
- \*\* E6.22. *Il dilemma di Monty Hall.* In una rivista molto popolare, Marylin vos Savant descrisse il problema seguente (ispirato a un gioco a premi televisivo presentato da Monty Hall).

Immaginate di essere i concorrenti di un gioco a premi televisivo nel quale dovete scegliere una porta tra le tre che vi vengono proposte: dietro a una porta c'è, in premio, un'automobile, dietro le altre ci sono delle capre. Voi scegliete una porta, ad esempio la numero 1, e il presentatore, che sa cosa c'è dietro alle porte, apre una delle altre due, ad esempio la numero 3, dove c'è una capra. A quel punto vi chiede: "Vuoi cambiare idea e scegliere la porta numero 2?". Il problema è: vi conviene cambiare idea?

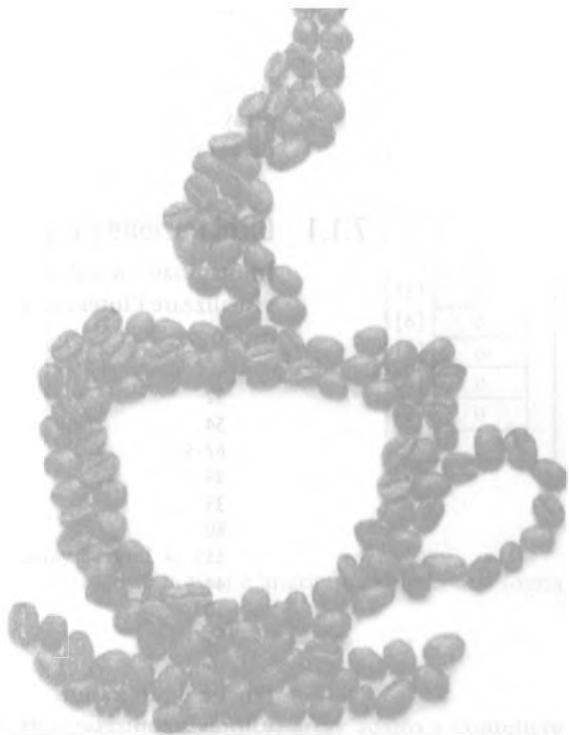
La giornalista dimostrò che conviene, ma molti dei suoi lettori, tra i quali alcuni docenti di matematica, non furono d'accordo con lei, sostenendo che la probabilità di vittoria non cambia per il solo fatto che sia stata aperta una porta.

Il vostro compito è quello di simulare questo gioco. In ciascuna iterazione della simulazione scegliete a caso una porta (cioè un numero da 1 a 3) dove posizionare l'automobile, poi simulate il fatto che il concorrente scelga, di nuovo a caso, una porta. Successivamente, il presentatore del gioco sceglie a caso una porta che nasconde una capra (ma che non sia la porta scelta dal concorrente). Incrementate il contatore della "strategia 1" se il giocatore vince cambiando la porta scelta e il contatore della "strategia 2" se il giocatore vince senza cambiare idea. Eseguite 1000 simulazioni e visualizzate entrambi i contatori.

Sul sito web dedicato al libro si trova una raccolta di progetti di programmazione più complessi.

# 7

## Array e vettori



### Obiettivi del capitolo

- Costruire raccolte di elementi usando array o vettori
- Usare il ciclo `for` esteso per scandire array e vettori
- Apprendere gli algoritmi più comuni per elaborare array e vettori
- Capire come usare array bidimensionali
- Apprendere il concetto di collaudo regressivo

In molti programmi c'è bisogno di raccogliere grandi quantità di dati, da memorizzare in strutture apposite: quelle più comunemente utilizzate in Java sono gli array e i vettori (o “liste ad accesso casuale”, *array list*). Gli array sono caratterizzati da una sintassi più sintetica, mentre i vettori possono crescere automaticamente fino alla dimensione richiesta. In questo capitolo imparerete a utilizzare array e vettori, apprendendo anche alcuni algoritmi di utilizzo frequente.

## 7.1 Array

Iniziamo questo capitolo presentando il tipo di dato *array* (“schiera” o “sequenza”). Gli array, in Java, sono il meccanismo fondamentale per realizzare raccolte di più valori e, nei paragrafi che seguono, imparerete a creare array e ad accedere ai loro elementi.

### 7.1.1 Dichiarazione e utilizzo di array

Immaginate di voler scrivere un programma che legge una sequenza di valori, per poi visualizzare l’intera sequenza con il valore massimo appositamente evidenziato, in questo modo:

```
32
54
67.5
29
35
80
115 <= largest value
44.5
100
65
```

Finché il programma non ha acquisito tutti i valori, non c’è modo di sapere quale sarà il valore da contrassegnare come valore massimo: dopo tutto, anche l’ultimo valore potrebbe essere il massimo. Il programma deve, quindi, memorizzare tutti i valori acquisiti, prima di poterli visualizzare.

Si può semplicemente memorizzare ciascun valore in una diversa variabile? Se sapete che i valori saranno dieci, li potete memorizzare in dieci variabili: `value1`, `value2`, `value3`, ..., `value10`. In ogni caso, però, l’utilizzo di una tale sequenza di variabili è veramente scomoda: dovrete ripetere per dieci volte parecchie righe di codice, una volta per ciascuna variabile. In Java, per memorizzare una sequenza di valori dello stesso tipo è decisamente preferibile usare un **array**.

Un array contiene una sequenza  
di valori del medesimo tipo,  
cioè omogenei.

Ecco come si crea un array che può contenere dieci valori di tipo `double`:

```
new double[10]
```

Il numero di elementi (10, in questo caso) costituisce la *lunghezza* dell’array.

L’operatore `new` costruisce semplicemente l’array: memorizzerete poi l’array in una variabile, in modo da potervi accedere successivamente.

Il tipo di una “variabile array” (cioè di una “variabile che consente di accedere a un oggetto di tipo array”) è uguale al tipo degli elementi che saranno memorizzati nell’array, seguito da una coppia di parentesi quadre, `[]`: in questo esempio il tipo è `double[]`, perché gli elementi sono di tipo `double`.

Ecco la dichiarazione di una variabile array di tipo `double[]` (come si può vedere nella Figura 1):

```
double[] values; ①
```

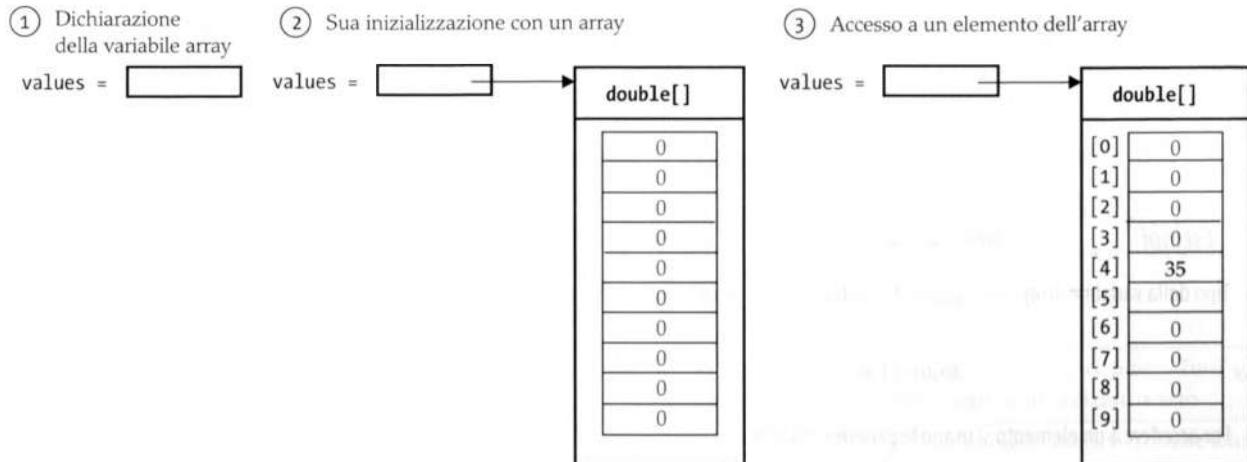


Figura 1 Un array avente dimensione 10

Quando si dichiara una variabile di tipo array, questa non è ancora inizializzata: bisogna inizializzarla assegnandole un array.

```
double[] values = new double[10];    ②
```

A questo punto la variabile `values` è stata inizializzata con un array adatto a contenere 10 numeri in virgola mobile, ciascuno dei quali vale, per impostazione predefinita, zero.

Nel momento in cui si dichiara un array, se ne può anche specificare il contenuto iniziale, come in questo esempio:

```
double[] moreValues = { 32, 54, 67.5, 29, 35, 80, 115, 44.5, 100, 65 };
```

Quando si specificano i valori iniziali, non si usa l'operatore `new`. Il compilatore determina la lunghezza dell'array contando i valori iniziali specificati.

Per accedere a un elemento contenuto in un array, bisogna specificare la “posizione” che si vuole utilizzare, usando l'operatore `[]`:

```
values[4] = 35;    ③
```

Ora la posizione numero 4 dell'array `values` contiene il numero 35 (come si può vedere nella Figura 1). Questo “numero della posizione” viene solitamente chiamato **indice** (*index*). In un array, ciascuna posizione può contenere un **elemento**.

Dato che `values` è un array di valori di tipo `double`, ciascun elemento, `values[i]`, può essere utilizzato esattamente come se fosse una variabile di tipo `double`. Ad esempio, con l'enunciato seguente si può visualizzare l'elemento avente indice 4:

```
System.out.println(values[4]);
```

Si accede a un singolo elemento  
di un array mediante un indice intero *i*,  
usando la notazione `array[i]`.

Un elemento di un array può essere  
usato esattamente come  
una singola variabile dello stesso tipo.

## Sintassi di Java

### 7.1 Array

#### Sintassi

Per costruire un array: `new nomeTipo[lunghezza]`

Per accedere a un elemento: `riferimentoAdArray[indice]`

#### Esempi

Nome della variabile array      /  
 Lunghezza

Tipo della variabile array —— `double[] values = new double[10];`

`double[] moreValues = { 32, 54, 67.5, 29, 35 };`

Per accedere a un elemento si usano le parentesi quadre.

`values[i] = 29.95;`

Elementi inizializzati con  
questi valori.

L'indice deve essere  $\geq 0$   
e < della lunghezza dell'array.

Prima di proseguire, dobbiamo occuparci di un dettaglio molto importante che riguarda gli array in Java. Se osservate attentamente la parte destra della Figura 1, noterete che, modificando `values[4]`, è cambiato il valore del *quinto* elemento dell'array: questo accade perché, in Java, gli elementi degli array sono numerati a partire da zero. Di conseguenza, gli elementi validi per l'array `values` sono:

- `values[0]`, il primo elemento
- `values[1]`, il secondo elemento
- `values[2]`, il terzo elemento
- `values[3]`, il quarto elemento
- `values[4]`, il quinto elemento
- ...
- `values[9]`, il decimo elemento

In altre parole, la dichiarazione

`double[] values = new double[10];`

crea un array con dieci elementi e un indice valido in tale array può essere qualsiasi numero intero compreso tra 0 e 9, estremi inclusi.

Bisogna fare attenzione al fatto che gli indici che si utilizzano appartengano all'intervallo degli indici validi: il tentativo di accedere a un elemento inesistente nell'array è un errore grave. Ad esempio, se `values` ha dieci elementi, non si può accedere al valore `values[20]`. Il tentativo di accesso a un elemento il cui indice non appartenga all'intervallo degli indici validi costituisce un **errore di limiti** (*bounds error*), un tipo di errore che non viene rilevato dal compilatore. Quando succede un errore di limiti durante l'esecuzione di un programma, si verifica il lancio di un'eccezione.

L'indice in un array deve essere almeno zero e minore della lunghezza dell'array.

Un errore di limiti, che si verifica se si usa un indice non valido in un array, può far terminare il programma.

Per conoscere il numero di elementi presenti in un array si usa l'espressione `array.length`.

Ecco un errore di limiti molto frequente:

```
double[] values = new double[10];
values[10] = value;
```

In un array avente dieci elementi, l'elemento `values[10]` non esiste: l'indice può andare da 0 a 9.

Per evitare questo tipo di errori, vi servirà conoscere il numero di elementi presenti in un array. L'espressione `values.length` ha sempre un valore uguale al numero di elementi dell'array `values`. Notate che `length` non è seguita da parentesi.

**Tabella 1**

Dichiarazione di array

|                                                                                                                               |                                                                                                                         |
|-------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------|
| <code>int[] numbers = new int[10];</code>                                                                                     | Un array di dieci numeri interi. Tutti gli elementi vengono inizializzati a zero.                                       |
| <code>final int LENGTH = 10; int[] numbers = new int[LENGTH];</code>                                                          | Invece di un "numero magico", è bene usare una costante.                                                                |
| <code>int length = in.nextInt(); double[] data = new double[length];</code>                                                   | La lunghezza può essere specificata con una variabile.                                                                  |
| <code>int[] squares = { 0, 1, 4, 9, 16 };</code>                                                                              | Un array di cinque numeri interi, con valori iniziali assegnati.                                                        |
| <code>String[] friends = { "Emily", "Bob", "Cindy"};</code>                                                                   | Un array di tre stringhe.                                                                                               |
|  <code>double[] values = new int[10];</code> | <b>Errore:</b> non si può assegnare un array di tipo <code>int[]</code> a una variabile di tipo <code>double[]</code> . |

Il codice seguente garantisce l'accesso all'array soltanto quando la variabile `i`, usata come indice, ha un valore che si trova nell'intervallo lecito:

```
if (0 <= i && i < values.length) { values[i] = value; }
```

Gli array hanno un limite pesante: *la loro lunghezza è fissa*. Se iniziate usando un array di 10 elementi e più tardi vi accorgete che vi servono ulteriori elementi, siete costretti a creare un nuovo array e a copiare tutti i valori dal vecchio al nuovo array. Questo procedimento verrà illustrato in dettaglio nel Paragrafo 7.3.9.

Per ispezionare tutti gli elementi contenuti in un array si usa una variabile che agisca da indice. Supponiamo che `values` abbia dieci elementi e che la variabile numerica intera `i` assuma i valori 0, 1, 2, e così via, fino a 9: di conseguenza, l'espressione `values[i]` rappresenta, uno dopo l'altro, tutti gli elementi. Questo ciclo, ad esempio, visualizza tutti gli elementi dell'array `values`:

```
for (int i = 0; i < 10; i++)
{
    System.out.println(values[i]);
}
```

Si noti che la condizione di controllo del ciclo prevede che l'indice sia *minore di 10*, perché non esiste alcun elemento corrispondente all'espressione `values[10]`.

### 7.1.2 Riferimenti ad array

Se osservate con attenzione la Figura 1, noterete che la variabile `values` non memorizza alcun numero. In effetti, l'array che contiene i numeri si trova in un'altra zona di memoria e la variabile `values` contiene un **riferimento** (*reference*) all'array (il riferimento rappresenta la posizione dell'array all'interno della memoria). Abbiamo già visto questo comportamento nel Paragrafo 2.8, parlando di oggetti. In ogni caso, quando si compie l'accesso a un oggetto o a un array, non occorre preoccuparsi del fatto che Java faccia uso di riferimenti: questo dettaglio diventa importante soltanto quando si copia un riferimento.

Un riferimento a un array specifica la posizione dell'array all'interno della memoria. Copiando un riferimento si ottiene un secondo riferimento allo stesso array.

Dopo aver copiato una variabile array in un'altra, entrambe fanno riferimento allo stesso array, come si può vedere nella Figura 2.

```
int[] scores = { 10, 9, 7, 4, 5 };
int[] values = scores; // si copia il riferimento all'array
```

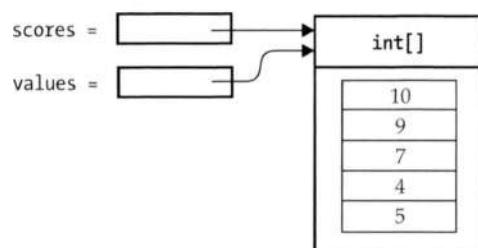
Il contenuto dell'array può essere modificato usando una qualsiasi delle due variabili:

```
scores[3] = 10;
System.out.println(values[3]); // visualizza 10
```

Nel Paragrafo 7.3.9 vedrete come fare, invece, una copia *dei contenuti* di un array.

**Figura 2**

Due variabili array che fanno riferimento al medesimo array



### 7.1.3 Array e metodi

Gli array, come qualsiasi altro valore, possono essere argomenti di metodi o valori restituiti da metodi.

Quando si definisce un metodo che ha un argomento di tipo array, in pratica si definisce una variabile parametro che farà riferimento a tale array. Ad esempio, il metodo seguente aggiunge voti a un oggetto di tipo `Student`:

```
public void addScores(int[] values)
{
    for (int i = 0; i < values.length; i++)
    {
        totalScore = totalScore + values[i];
    }
}
```

Gli array possono essere argomenti di metodi o valori restituiti da metodi.

Per invocare questo metodo è necessario fornire un array come parametro:

```
int[] scores = { 10, 9, 7, 10 };
fred.addScores(scores);
```

Analogamente, un metodo può restituire un array. Ad esempio, la classe `Student` può avere un metodo come questo:

```
public int[] getScores()
```

che restituisce un array contenente tutti i voti dello studente.

### 7.1.4 Array riempiti solo in parte

La dimensione di un array non può essere modificata durante l'esecuzione di un programma. Questo vincolo costituisce un problema tutte le volte in cui non si sa a priori il numero di elementi che saranno necessari all'interno di un array: bisogna fare una previsione su tale numero massimo. Per esempio, possiamo decidere che qualche volta vorremo memorizzare più di dieci elementi, ma, in ogni caso, mai più di 100:

```
final int LENGTH = 100;
double[] values = new double[LENGTH];
```

Insieme a un array riempito parzialmente, usate una variabile ausiliaria che tenga traccia del numero di elementi realmente utilizzati.

Durante una normale esecuzione del programma, soltanto una porzione di questo array risulterà effettivamente occupata da elementi: un array usato in questo modo viene chiamato **array riempito solo in parte** (*partially filled array*). Insieme all'array, bisogna utilizzare una *variabile ausiliaria* associata ad esso, che conti il numero di elementi effettivamente in uso, variabile che nella Figura 3 è stata chiamata `currentSize`.

Il ciclo seguente acquisisce valori in ingresso e li memorizza nell'array `values`:

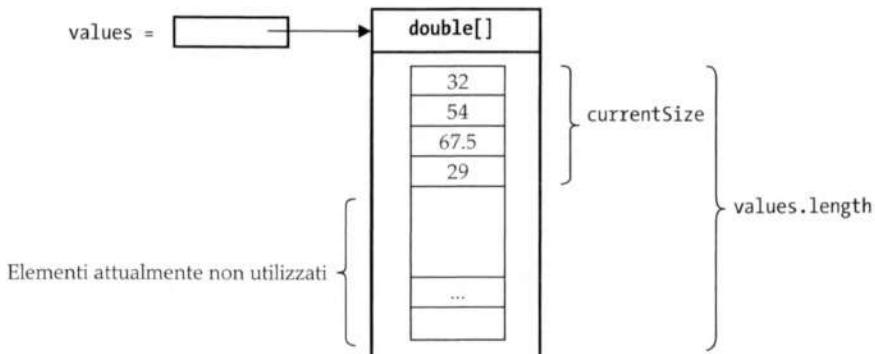
```
int currentSize = 0;
Scanner in = new Scanner(System.in);
while (in.hasNextDouble())
{
    if (currentSize < values.length)
    {
        values[currentSize] = in.nextDouble();
        currentSize++;
    }
}
```

Al termine di questo ciclo la variabile `currentSize` contiene il numero effettivo di valori memorizzati nell'array. Si osservi che quando la variabile `currentSize` raggiunge il valore `values.length` dobbiamo smettere di accettare nuovi valori in ingresso.

Per elaborare gli elementi contenuti nell'array, non basta conoscere la lunghezza dell'array, serve nuovamente la variabile ausiliaria. Questo ciclo visualizza il contenuto dell'array riempito solo in parte:

```
for (int i = 0; i < currentSize; i++)
{
    System.out.println(values[i]);
}
```

**Figura 3**  
Un array riempito solo in parte



### Auto-valutazione

- Dichiarate un array di numeri interi contenente i primi cinque numeri primi.
- Nell'ipotesi che l'array `primes` sia stato inizializzato secondo quanto descritto nella domanda precedente, cosa conterrà dopo l'esecuzione del ciclo seguente?
 

```
for (int i = 0; i < 2; i++)
    {
        primes[4 - i] = primes[i];
    }
```
- Nell'ipotesi che l'array `primes` sia stato inizializzato secondo quanto descritto nella domanda 1, cosa conterrà dopo l'esecuzione del ciclo seguente?
 

```
for (int i = 0; i < 5; i++)
    {
        primes[i]++;
    }
```
- Data la dichiarazione:
 

```
int[] values = new int[10];
```

 scrivete enunciati che memorizzino il numero 10 negli elementi dell'array `values` che corrispondono al minimo e massimo valore valido dell'indice.
- Dichiarate l'array `words` che possa contenere dieci elementi di tipo `String`.
- Dichiarate un array contenente due stringhe, "Yes" e "No".
- Siete in grado di risolvere il problema presentato all'inizio del Paragrafo 7.1.1 senza memorizzare i valori in un array, usando un algoritmo simile a quello visto nel Paragrafo 6.7.5 per trovare il valore massimo in un insieme di dati?
- Dichiarate, in una classe `Lottery`, un metodo che restituisca una combinazione di `n` numeri (senza implementarlo).

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R7.5, R7.6, R7.10 e E7.1, al termine del capitolo.



### Errori comuni 7.1

#### Errori di limiti

Forse l'errore più frequente nell'uso di array è l'accesso a un elemento che non esiste.

```
double[] values = new double[10];
values[10] = 5.4;
// Errore: values ha 10 elementi, con indice compreso tra 0 e 9
```

Se un programma accede a un array usando un indice fuori dai limiti, non c'è nessun messaggio d'errore in fase di compilazione: l'errore si verificherà durante l'esecuzione del programma.

## Errori comuni 7.2

### Array non inizializzati o non riempiti

Un errore frequente consiste nel dichiarare una variabile array, senza creare un array.

```
double[] values;
values[0] = 29.95; // Errore: la variabile values non è inizializzata
```

Le variabili array funzionano esattamente come le variabili oggetto: sono soltanto riferimenti all'array reale. Per costruire l'array vero e proprio, si deve usare l'operatore `new`:

```
double[] values = new double[10];
```

Forse ancora più comune è questo errore: creare un array di oggetti e pensare che vengano automaticamente creati gli oggetti necessari a riempirlo:

```
BankAccount[] accounts = new BankAccount[10];
// L'array contiene dieci riferimenti null
```

Questo array contiene dieci riferimenti `null`, non dieci conti bancari. Dovete ricordarvi di riempire l'array, ad esempio in questo modo:

```
for (int i = 0; i < 10; i++)
{
    accounts[i] = new BankAccount();
}
```

## Suggerimenti per la programmazione 7.1

### Usate array per sequenze di valori correlati

Gli array sono pensati per contenere sequenze di valori aventi il medesimo significato. Ad esempio, è perfettamente sensato creare un array di voti scolastici:

```
int[] scores = new int[NUMBER_OF_SCORES];
```

Ma usare un array come questo è una pessima idea

```
int[] personalData = new int[3];
```

se si pensa di memorizzare, come `personalData[0]`, `personalData[1]` e `personalData[2]`, rispettivamente, l'età di una persona, il saldo del suo conto bancario e il suo numero di

scarpe. Sarebbe davvero complicato e noioso, per un programmatore, ricordarsi quale di questi valori è stato memorizzato in un determinato elemento dell'array: molto meglio usare tre variabili.



## Suggerimenti per la programmazione 7.2

### Trasformare array paralleli in array di oggetti

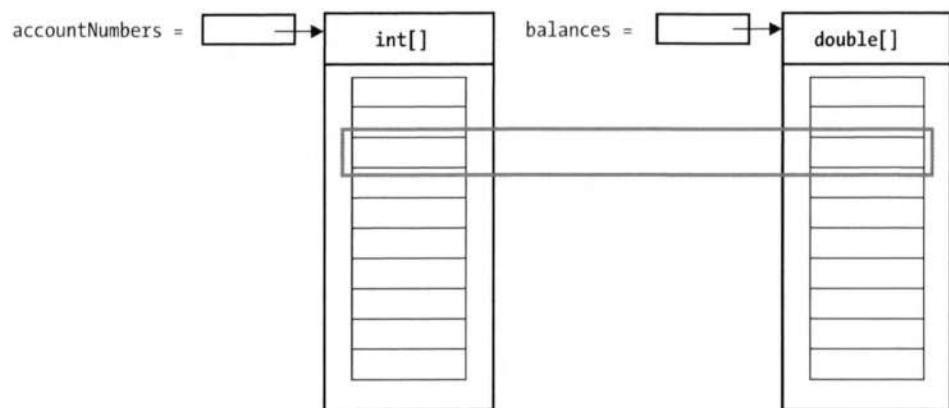
Ai programmatori che sono soliti usare array, ma non hanno familiarità con la programmazione orientata agli oggetti, capita di distribuire le informazioni su array separati. Ecco un esempio tipico: un programma deve gestire dati bancari, che consistono in numeri di conti bancari e nei relativi saldi. Non memorizzate numeri e saldi in array separati:

```
// non fate così
int[] accountNumbers; // numeri di conto
double[] balances; // saldi
```

Array come questi vengono detti *array paralleli* (osservate la Figura 4), perché la *sezione* *i*-esima della struttura contiene dati che devono essere elaborati congiuntamente (`accountNumbers[i]` e `balances[i]`).

**Figura 4**

Evitate gli array paralleli



Evitate di usare array paralleli:  
trasformateli in array di oggetti.

Se vi accorgete che state usando due array che hanno la stessa lunghezza, chiedetevi se potete sostituirli con un unico array che contenga oggetti di una classe. Osservate la “sezione” di dati e identificate il concetto che rappresenta: quindi, inserite tale concetto in una classe. Nel nostro esempio ciascuna sezione contiene un numero di conto bancario e il suo saldo, che sono proprietà di un conto bancario. Quindi, è facile usare un unico array di oggetti:

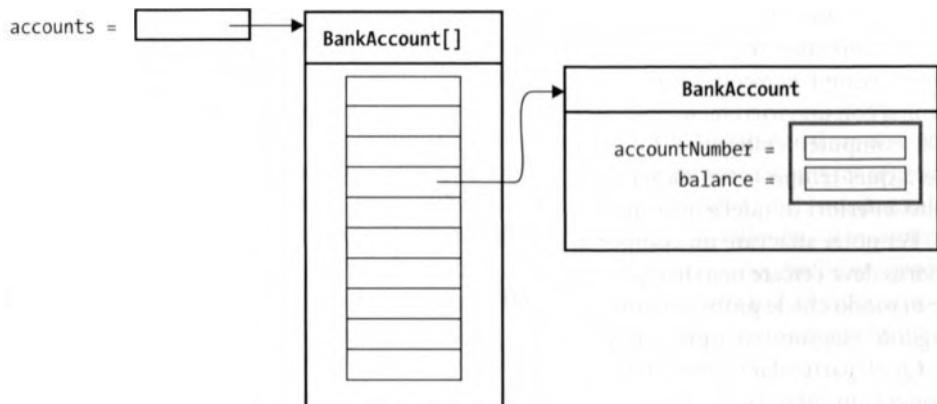
```
BankAccount[] accounts;
```

Osservate la Figura 5: quali sono i vantaggi? Pensate al futuro. Forse il vostro programma verrà modificato e avrete bisogno di memorizzare anche i proprietari dei conti bancari.

Aggiornare la classe `BankAccount` è piuttosto semplice, mentre potrebbe essere abbastanza complicato aggiungere un nuovo array ed essere sicuri che tutti i metodi che accedono ai due array originari accedano ora correttamente anche al terzo.

**Figura 5**

Riorganizzate array paralleli in array di oggetti



Argomenti avanzati 7.1



### Metodi con un numero di parametri variabile

In Java è possibile dichiarare metodi che ricevono un numero di parametri variabile. Ad esempio, possiamo scrivere un metodo che può aggiungere a uno studente un numero di valutazioni arbitrario:

```
fred.addScores(10, 7); // invocazione con due parametri  
fred.addScores(1, 7, 2, 9); // invocazione dello stesso metodo, con 4 parametri
```

Tale metodo va dichiarato in questo modo:

```
public void addScores(int... values)
```

Il simbolo `int...` specifica che il metodo può ricevere un numero qualsiasi di argomenti di tipo `int`. Il parametro `values` è, in realtà, un array di tipo `int[]` contenente tutti i valori che vengono effettivamente passati al metodo. Il codice che realizza il metodo può scandire l'array ricevuto come parametro ed elaborare i valori:

```
public void addScores(int... values)
{
    for (int i = 0; i < values.length; i++) // values è di tipo int[]
    {
        totalScore = totalScore + values[i];
    }
}
```



## Computer e società 7.1

### I virus per computer

Nel novembre del 1988, Robert Morris, uno studente della Cornell University, mise in esecuzione un programma con un virus che infettò circa 6000 computer collegati a Internet (che a quel tempo aveva dimensioni molto inferiori di quelle attuali).

Per poter attaccare un computer, un virus deve cercare una strategia per fare in modo che le proprie istruzioni vengano eseguite su quel computer. Quel particolare virus metteva in opera un attacco di tipo "buffer overrun", cioè un utilizzo di una zona di memoria (*buffer*) al di là dei suoi confini corretti, fornendo una quantità inaspettatamente grande di dati in ingresso a un programma in esecuzione su un computer diverso, da infettare. Quel programma usava un array di 512 caratteri, nell'ipotesi che nessuno avrebbe mai fornito un quantità di dati in ingresso di dimensione maggiore. Sfortunatamente quel programma era stato scritto usando il linguaggio C, che, diversamente da Java, non verifica che l'indice usato in un array sia minore della lunghezza dell'array: se scrivete in un array usando un indice troppo grande, andate semplicemente a sovrascrivere posizioni di memoria relative ad altri dati. Chi programma in C dovrebbe preoccuparsi di fare controlli relativamente al corretto utilizzo degli indici in array, ma questo non venne fatto per il programma sotto attacco. Il programma del virus riempiva di proposito con 536 byte l'array lungo 512 caratteri: i 24 byte in eccesso sovrascrivono un indirizzo che l'attaccante sapeva essere memorizzato subito dopo lo spazio destinato ai

dati in ingresso e che rappresentava il punto in cui il programma doveva riprendere la propria esecuzione. Quando il metodo di acquisizione dei dati in ingresso terminava, il controllo dell'esecuzione non ritornava al metodo che l'aveva invocato, ma al codice fornito dal virus (come si può vedere nella figura). In questo modo il virus era in grado di far eseguire il proprio codice al sistema remoto, infettandolo.

In Java, come in C, i programmati devono stare molto attenti a non superare i confini di un array, ma in Java tale errore provoca un'eccezione durante l'esecuzione, senza alterare la memoria al di fuori di quella riservata all'array: è una delle caratteristiche che rendono più sicuri i programmi scritti in Java.

Ci si potrebbe chiedere per quale motivo un programmatore esperto dedichi settimane a progettare un programma che sia in grado di rendere inutilizzabili migliaia di computer. Sembra che l'intenzione dell'autore fosse la semplice invasione, mentre la messa fuori servizio dei computer sia stato un effetto secondario dovuto a un errore nella progettazione del virus stesso e a ripetute re-infezioni. Morris venne condannato a 3 anni di affidamento in prova, 400 ore di servizi sociali e diecimila dollari di multa.

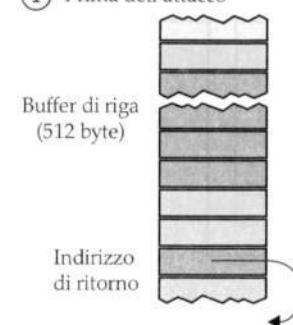
Di recente gli attacchi ai computer si sono intensificati e le motivazioni sono diventate più criminali: invece di mettere fuori uso i computer, spesso i virus si installano in modo permanente nei computer attaccati. I criminali, poi, possono sfruttare la potenza di elaborazione di milioni di computer infettati per, ad esempio, inviare messaggi di posta

elettronica con pubblicità indesiderata (*spam*). Altri virus sono in grado di tenere sotto controllo la tastiera e di inviare ai criminali una copia di tutti i caratteri che vi vengono digitati, consentendo loro di carpire numeri di carte di credito o credenziali di accesso a servizi bancari.

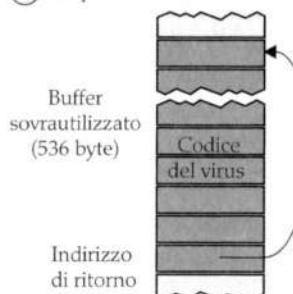
Solitamente l'infezione di un computer avviene perché l'utente esegue codice scaricato da Internet, selezionando un'icona con il mouse o seguendo un collegamento che sembra portare a un videogioco o alla visualizzazione di un filmato. I programmi anti-virus hanno il compito specifico di controllare tutti i programmi scaricati, confrontandoli con un elenco di virus noti che si allunga di continuo.

### Un attacco di tipo "Buffer Overrun"

#### ① Prima dell'attacco



#### ② Dopo l'attacco



Quando usate un calcolatore per gestire i vostri conti bancari, dovete essere ben consapevoli dei rischi derivanti da infezioni virali. Se un virus legge le vostre credenziali bancarie e svuota il vostro conto corrente, non sarà facile convincere l'istituto di credito che l'azione non è stata opera vostra e probabilmente perderete i vostri soldi. Tenete aggiornato il vostro sistema operativo e il relativo programma anti-virus, e non seguite collegamenti sospetti presenti in pagine web o nei messaggi di posta elettronica che ricevete. Usate sistemi

bancari online che, per le operazioni più rilevanti, richiedono l'uso di una "autenticazione in due fasi", ad esempio effettuando una chiamata al vostro telefono cellulare.

I virus sono usati anche per scopi militari. Nel 2010, un virus chiamato Stuxnet si diffuse attraverso il sistema operativo Microsoft Windows e infettò le chiavette USB, facendo in modo che queste cercassero calcolatori industriali Siemens per riprogrammarli in modo quasi impercettibile. Questo comportamento aveva lo scopo di danneggiare

le centrifughe usate in Iran per le operazioni di arricchimento nucleare: i computer che controllavano le centrifughe non erano collegati a Internet, ma venivano configurati tramite chiavette USB, alcune delle quali furono infettate. I ricercatori che si occupano di sicurezza informatica ritengono che il virus sia stato messo a punto dai servizi segreti di Israele e Stati Uniti, raggiungendo l'obiettivo di rallentare il programma nucleare iraniano, anche se nessuno dei due Paesi ha mai riconosciuto (né negato) un proprio coinvolgimento.

## 7.2 Il ciclo for esteso

Il ciclo for esteso scandisce tutti gli elementi di un array.

C'è spesso la necessità di scandire tutti gli elementi di un array: il ciclo `for` esteso rende particolarmente semplice la programmazione di questa procedura.

Immaginate di voler sommare tutti gli elementi presenti nell'array `values`.

Usando il ciclo `for` esteso potete fare in questo modo:

```
double[] values = . . .;
double total = 0;
for (double element : values)
{
    total = total + element;
}
```

Il corpo del ciclo viene eseguito per ciascun elemento presente nell'array `values`. All'inizio di ogni iterazione del ciclo, alla variabile `element` viene assegnato l'elemento successivo dell'array, poi viene eseguito il corpo del ciclo: in pratica, potete immaginare che il ciclo si legga come "per ogni `element` appartenente a `values`".

Il ciclo precedente è equivalente a quello seguente, un normale ciclo `for` con una variabile indice esplicita:

```
for (int i = 0; i < values.length; i++)
{
    double element = values[i];
    total = total + element;
}
```

Noteate una differenza importante che esiste tra il ciclo `for` esteso e quello normale: nel primo caso, alla "variabile elemento" `element` vengono via via assegnati i valori degli elementi, `values[0], values[1], ...`, mentre nel secondo caso alla "variabile indice" `i` vengono assegnati i valori degli indici: `0, 1, ...`

Si può usare un ciclo **for** esteso soltanto se nel corpo del ciclo non c'è bisogno del valore dell'indice.

Ricordate che il ciclo **for** esteso viene utilizzato per uno scopo ben preciso: la scansione, dall'inizio alla fine, di tutti gli elementi di una raccolta. Quindi, non è necessariamente adatto per tutti gli algoritmi che operano su array e, in particolare, non consente di modificare i contenuti dell'array. Il ciclo seguente, infatti, *non* riempie l'array di zeri:

```
for (double element : values)
{
    element = 0;
    // ERRORE: questa assegnazione non modifica gli elementi dell'array
}
```

Quando questo ciclo viene eseguito, durante la sua prima iterazione alla variabile `element` viene assegnato il valore `values[0]`, dopodiché si assegna zero a `element` e si passa all'iterazione successiva, senza, ovviamente, aver modificato il contenuto dell'array `values`. Il rimedio è semplice: usate un ciclo ordinario.

```
for (int i = 0; i < values.length; i++)
{
    values[i] = 0; // così va bene
}
```



### Auto-valutazione

9. Cosa fa questo ciclo?

```
int counter = 0;
for (double element : values)
{
    if (element == 0) { counter++; }
```

## Sintassi di Java

### 7.2 Il ciclo **for** esteso

#### Sintassi

```
for (nomeTipo variabile : raccolta)
{
    enunciati
}
```

#### Esempio

Questa variabile assume un nuovo valore a ogni iterazione del ciclo ed è definita soltanto all'interno del ciclo stesso.

Questi enunciati vengono eseguiti per ciascun elemento.

```
{   for (double element : values)
    {
        sum = sum + element;
    }}
```

Un array

La variabile contiene un elemento, non un indice.

10. Scrivete un ciclo `for` esteso che visualizzi tutti gli elementi dell'array `values`.
11. Scrivete un ciclo `for` esteso che moltiplicherà tra loro tutti gli elementi dell'array `factors` di tipo `double[]`, accumulando il risultato nella variabile `product`.
12. Perché questo ciclo non può essere trasformato in un ciclo `for` esteso?  
`for (int i = 0; i < values.length; i++) { values[i] = i * i; }`

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R7.11, R7.12 e R7.13, al termine del capitolo.

## 7.3 Algoritmi fondamentali per l'elaborazione di array

In questo paragrafo vedrete alcuni degli algoritmi più utilizzati per l'elaborazione di array. Se li usate con array riempiti solo in parte, ricordatevi di sostituire `values.length` con la variabile ausiliaria che memorizza la dimensione effettiva dell'array.

### 7.3.1 Riempimento

Questo ciclo riempie un array con i quadrati dei primi numeri interi: l'elemento di indice 0 conterrà  $0^2$ , l'elemento di indice 1 conterrà  $1^2$ , e così via.

```
for (int i = 0; i < values.length; i++)
{
    values[i] = i * i;
}
```

### 7.3.2 Somma e valore medio

Avete già visto questo algoritmo nel Paragrafo 6.7.1. Quando i valori sono memorizzati in un array il codice è ancora più semplice:

```
double total = 0;
for (double element : values)
{
    total = total + element;
}
double average = 0;
if (values.length > 0) { average = total / values.length; }
```

### 7.3.3 Valore massimo e minimo

Ricordiamo l'algoritmo visto nel Paragrafo 6.7.5, che usa una variabile (`largest`) per memorizzare l'elemento massimo incontrato durante l'esecuzione, e implementiamolo per un array:

```
double largest = values[0],
for (int i = 1; i < values.length; i++)
{
```

```

if (values[i] > largest)
{
    largest = values[i];
}
}

```

Si osservi che il ciclo inizia da 1, perché abbiamo usato `values[0]` per inizializzare `largest`.

Ovviamente, per calcolare il valore minimo basta invertire il senso dell'operatore relazionale di confronto.

Entrambi questi algoritmi richiedono che l'array contenga almeno un elemento.

### 7.3.4 Elementi con separatori

**Quando si separano elementi,  
non bisogna inserire un separatore  
prima del primo elemento.**

Quando si visualizzano gli elementi di un array, solitamente li si vuole separare, spesso con virgole o barrette verticali, in questo modo:

```
32 | 54 | 67.5 | 29 | 35
```

Osservate che il numero di separatori è inferiore di un'unità rispetto al numero di elementi. Si visualizza un separatore prima di ogni elemento dell'array, *tranne il primo* (che è quello di indice 0), in questo modo:

```

for (int i = 0; i < values.length; i++)
{
    if (i > 0)
    {
        System.out.print(" | ");
    }
    System.out.print(values[i]);
}

```

Se come elemento separatore si vuole usare una virgola, si può utilizzare il metodo `Arrays.toString` (importando `java.util.Arrays`). L'invocazione

```
Arrays.toString(values)
```

restituisce una stringa che descrive i valori contenuti nell'array `values`, con il formato:

```
[32, 54, 67.5, 29, 35]
```

Gli elementi, quindi, vengono complessivamente racchiusi da una coppia di parentesi quadre, separati da virgole. Tale metodo può essere comodo per il debugging:

```
System.out.println("values=" + Arrays.toString(values));
```

### 7.3.5 Ricerca lineare

Spesso occorre trovare la posizione di uno specifico elemento all'interno di un array, per poterlo sostituire o eliminare. Si ispezionano tutti gli elementi, uno dopo l'altro, finché

non si trova quello cercato oppure si giunge alla fine dell'array. Ecco come trovare, in un array, la posizione del primo elemento con valore uguale a 100:

```

int searchedValue = 100;
int pos = 0;
boolean found = false;
while (pos < values.length && !found)
{
    if (values[pos] == searchedValue)
    {
        found = true;
    }
    else
    {
        pos++;
    }
}
if (found) { System.out.println("Found at position: " + pos); } trovato
else { System.out.println("Not found"); }
```

Una ricerca lineare ispeziona gli elementi in sequenza finché non trova quello cercato.

Questo algoritmo si chiama **ricerca lineare** (*linear search*) o ricerca sequenziale, perché gli elementi dell'array vengono ispezionati in sequenza. Se l'array è ordinato, si può usare il più efficiente algoritmo di **ricerca binaria** (*binary search*) o ricerca per bisezione, di cui parleremo nel Capitolo 13.

### 7.3.6 Eliminazione di un elemento

Supponiamo di voler rimuovere l'elemento di indice *pos* dall'array *values*. Innanzitutto ci serve una variabile ausiliaria per tenere traccia del numero di elementi presenti nell'array, come visto nel Paragrafo 7.1.4. In questo esempio useremo la variabile *currentSize*.

Se gli elementi sono memorizzati nell'array senza rispettare alcuna particolare proprietà di ordinamento, basta semplicemente sovrascrivere l'elemento da eliminare con l'ultimo elemento presente nell'array, decrementando la variabile *currentSize*: la procedura è illustrata nella Figura 6.

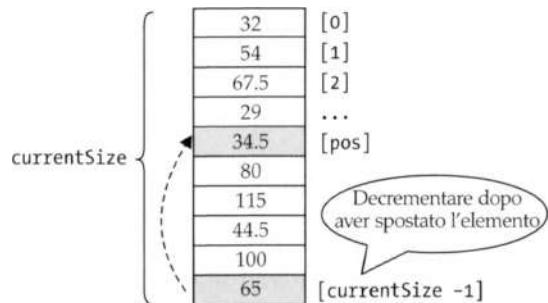
```
values[pos] = values[currentSize - 1];
currentSize--;
```

Se, invece, l'ordine tra gli elementi è importante e va mantenuto, la situazione si complica: tutti gli elementi che hanno indice maggiore dell'elemento da rimuovere vanno spostati di un posto, in una posizione avente indice inferiore di uno rispetto all'attuale, per poi, al termine della procedura, decrementare la variabile che memorizza la dimensione dell'array: tutto ciò è visibile nella Figura 7.

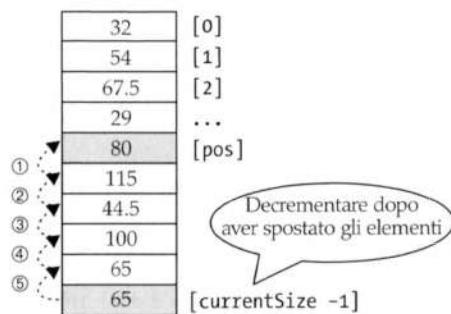
```

for (int i = pos + 1; i < currentSize; i++)
{
    values[i - 1] = values[i];
}
currentSize--;
```

**Figura 6**  
Eliminazione  
di un elemento da un array  
non ordinato



**Figura 7**  
Eliminazione  
di un elemento da un array  
ordinato



### 7.3.7 Inserimento di un elemento

In questo paragrafo vedrete come inserire un elemento in un array, osservando subito che abbiamo bisogno di una variabile ausiliaria per tenere traccia della dimensione dell'array, come visto nel Paragrafo 7.1.4.

Se l'ordine tra gli elementi non è importante, potete semplicemente inserire i nuovi elementi alla fine, incrementando la variabile che memorizza la dimensione dell'array, come si può vedere nella Figura 8.

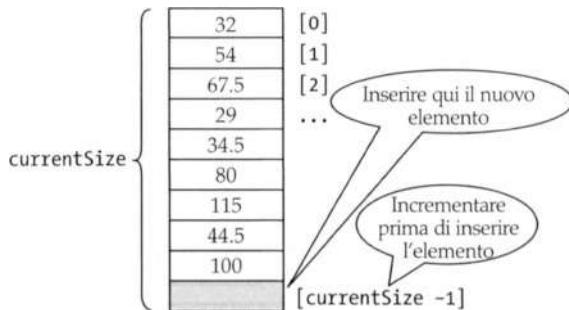
```
if (currentSize < values.length)
{
    currentSize++;
    values[currentSize - 1] = newElement;
}
```

Se, invece, volete inserire un elemento in una specifica posizione intermedia nell'array, per prima cosa tutti gli elementi che hanno indice maggiore o uguale a quello della posizione che interessa vanno spostati di un posto, in una posizione avente indice maggiore di uno rispetto all'attuale; poi, al termine della procedura, si inserisce il nuovo elemento, come si vede nella Figura 9.

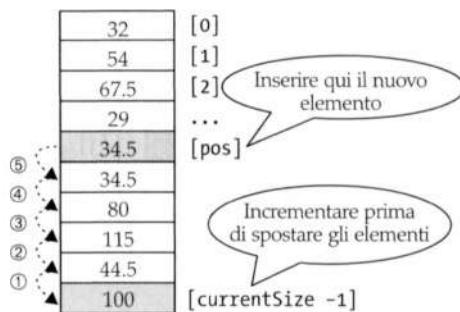
Prima di inserire un elemento, bisogna spostare gli elementi verso la fine dell'array, a partire dall'ultimo.

Osservate con attenzione l'ordine in cui vengono effettuati gli spostamenti. Quando si elimina un elemento, prima si spostano gli elementi di indice minore, uno dopo l'altro, fino a raggiungere la fine dell'array. Quando si inserisce un elemento, invece, si iniziano gli spostamenti dalla fine dell'array e si procede a ritroso fin quando si raggiunge la posizione di inserimento desiderata.

**Figura 8**  
Inserimento  
di un elemento in un array  
non ordinato



**Figura 9**  
Inserimento  
di un elemento in un array  
ordinato



```
if (currentSize < values.length)
{
    currentSize++;
    for (int i = currentSize - 1; i > pos; i--)
    {
        values[i] = values[i - 1];
    }
    values[pos] = newElement;
}
```

### 7.3.8 Scambio di elementi

Spesso si ha la necessità di scambiare tra loro due elementi di un array. Ad esempio, si può ordinare il contenuto di un array effettuando ripetutamente scambi opportuni tra elementi che non rispettano il criterio di ordinamento.

Prendiamo in esame il problema dello scambio tra gli elementi in posizione  $i$  e in posizione  $j$  all'interno dell'array `values`. Sarebbe bello poter assegnare `values[j]` a `values[i]`, ma questa operazione sovrascriverebbe il valore attualmente memorizzato in `values[i]`, per cui, prima di fare questo, lo dobbiamo copiare in un posto sicuro:

```
double temp = values[i];      ②
values[i] = values[j];      ③
```

A questo punto possiamo assegnare a `values[j]` il valore che avevamo preservato in `temp`:

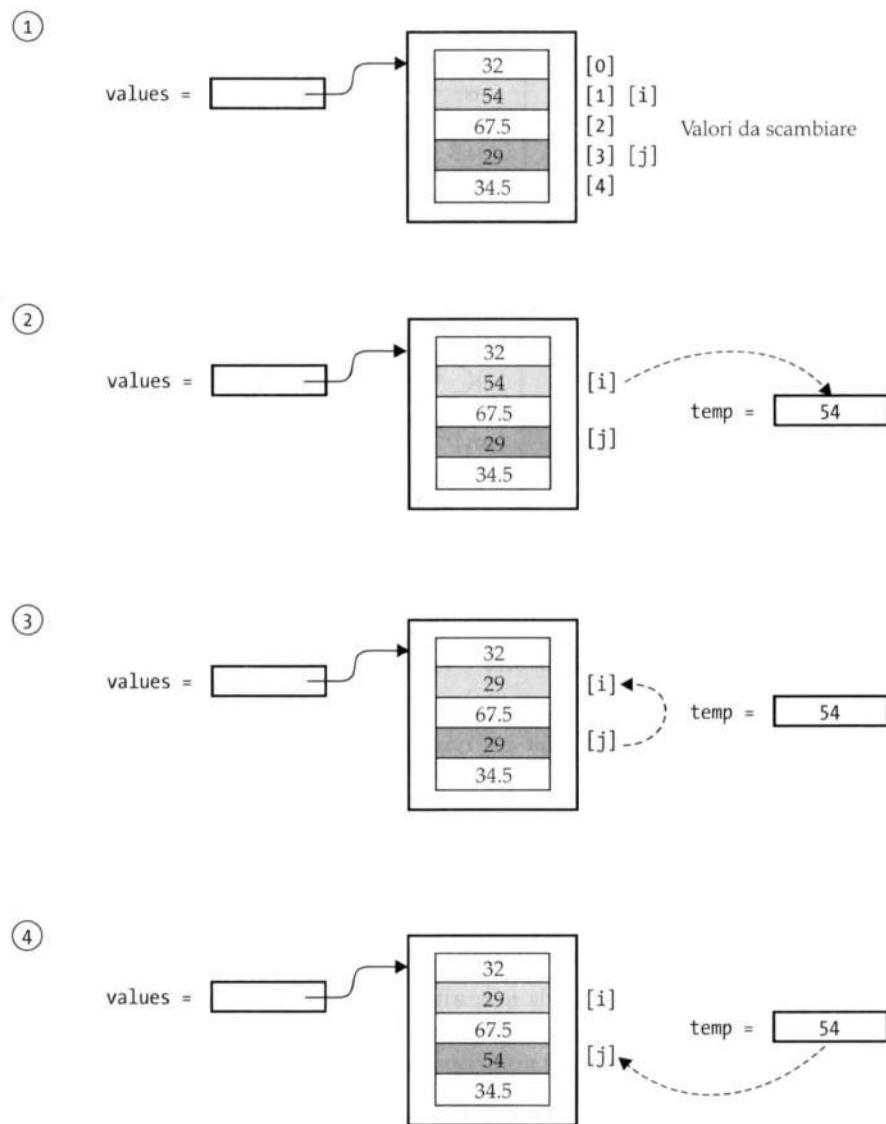
```
values[j] = temp;          ④
```

Per scambiare tra loro due elementi  
di un array bisogna usare  
una variabile temporanea.

La Figura 10 illustra l'intera procedura.

**Figura 10**

Scambio di due elementi  
in un array



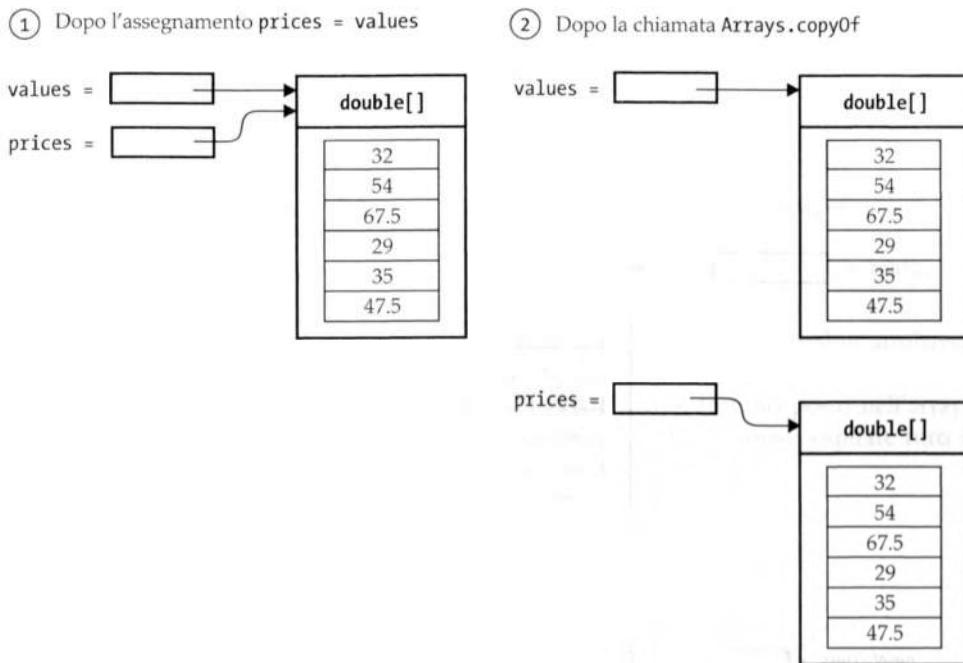
### 7.3.9 Copiatura di array

Le variabili array non memorizzano al proprio interno gli elementi dell'array, ma un riferimento all'array effettivo. Se copiate tale riferimento, ottenete un secondo riferimento al medesimo array, come si vede nella Figura 11.

```
double[] values = new double[6];
... // riempimento dell'array
double[] prices = values; ①
```

**Figura 11**

Differenza tra copiatura di un riferimento ad array e copiatura di un array



Per copiare gli elementi di un array,  
usate il metodo `Arrays.copyOf`.

Se volete fare una vera copia di un array, invocate il metodo `Arrays.copyOf`.

```
double[] prices = Arrays.copyOf(values, values.length); ②
```

L'invocazione `Arrays.copyOf(values, n)` crea un array di lunghezza `n`, vi copia i primi `n` elementi di `values` (o l'intero array se `n > values.length`) e restituisce il nuovo array.

Per poter usare la classe `Arrays` bisogna inserire all'inizio del programma il seguente enunciato:

```
import java.util.Arrays;
```

Il metodo `Arrays.copyOf` si può anche usare per ingrandire un array in cui serva più spazio. Gli enunciati seguenti hanno l'effetto di raddoppiare la dimensione dell'array `values`, come si vede nella Figura 12:

```
double[] newValues = Arrays.copyOf(values, 2 * values.length); ①
values = newValues; ②
```

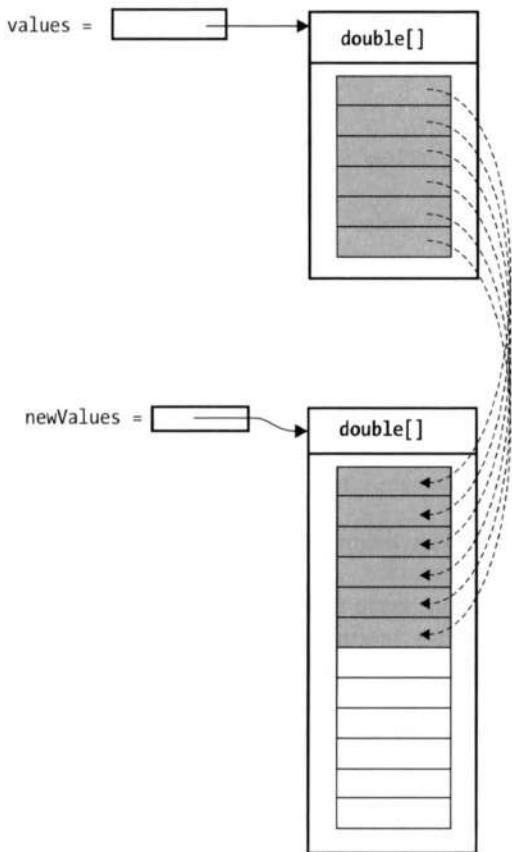
Il metodo `Arrays.copyOf` è stato aggiunto alla libreria di Java nella versione 6. Se usate Java 5, sostituite

```
double[] newValues = Arrays.copyOf(values, n);
```

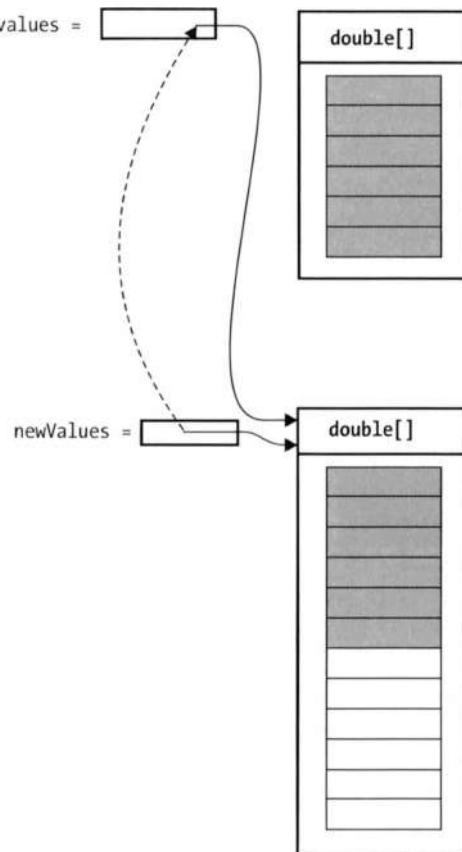
con

```
double[] newValues = new double[n];
for (int i = 0; i < n && i < values.length; i++)
{
    newValues[i] = values[i];
}
```

- ① Si copiano gli elementi in un array più grande



- ② Si memorizza in **values** il riferimento all'array più grande



**Figura 12** Aumento delle dimensioni di un array

### 7.3.10 Acquisizione di valori

Se sapete quanti valori verranno forniti dall'utente, è davvero semplice memorizzarli in un array:

```
double[] inputs = new double[NUMBER_OF_INPUTS];
for (int i = 0; i < inputs.length; i++)
{
    inputs[i] = in.nextDouble();
}
```

Questa tecnica, però, non funziona in quei casi in cui si ha la necessità di acquisire una sequenza di dati di lunghezza arbitraria. In tal caso si aggiungono i nuovi dati all'array fin quando non sono terminati:

```
int currentSize = 0;
while (in.hasNextDouble() && currentSize < inputs.length)
{
    inputs[currentSize] = in.nextDouble();
    currentSize++;
}
```

In questo modo `inputs` funziona come array riempito solo in parte e la variabile ausiliaria `currentSize` memorizza il numero di dati acquisiti.

Questo ciclo, però, ignora silenziosamente i dati che non trovano posto nell'array: un approccio migliore prevede di far crescere l'array in modo che possa ospitare tutti i dati che vengono acquisiti.

```
double[] inputs = new double[INITIAL_SIZE];
int currentSize = 0;
while (in.hasNextDouble())
{
    // aumenta la dimensione dell'array se è pieno
    if (currentSize >= inputs.length)
    {
        inputs = Arrays.copyOf(inputs, 2 * inputs.length);
    }
    inputs[currentSize] = in.nextDouble();
    currentSize++;
}
```

Al termine della fase di acquisizione, si possono eliminare gli elementi in eccesso (che non contengono dati):

```
inputs = Arrays.copyOf(inputs, currentSize);
```

L'esempio che segue è un programma che usa alcuni di questi algoritmi, risolvendo il problema da cui eravamo partiti all'inizio di questo capitolo: individuare e contrassegnare il valore massimo in una sequenza.

### File LargestInArray.java

```
1 import java.util.Scanner;
2
3 /**
4  * Legge una sequenza di valori e li visualizza, segnalando il massimo.
5 */
6 public class LargestInArray
7 {
8     public static void main(String[] args)
9     {
10         final int LENGTH = 100;
11         double[] values = new double[LENGTH];
```

```

12     int currentSize = 0;
13
14     // legge i valori
15
16     System.out.println("Please enter values, Q to quit:");
17     Scanner in = new Scanner(System.in);
18     while (in.hasNextDouble() && currentSize < values.length)
19     {
20         values[currentSize] = in.nextDouble();
21         currentSize++;
22     }
23
24     // trova il valore massimo
25
26     double largest = values[0];
27     for (int i = 1; i < currentSize; i++)
28     {
29         if (values[i] > largest)
30         {
31             largest = values[i];
32         }
33     }
34
35     // visualizza tutti i valori, segnalando il massimo
36
37     for (int i = 0; i < currentSize; i++)
38     {
39         System.out.print(values[i]);
40         if (values[i] == largest)
41         {
42             System.out.print(" == largest value");
43         }
44         System.out.println();
45     }
46 }
47 }
```

### Esecuzione del programma:

```

Please enter values, Q to quit:
34.5 80 115 44.5 Q
34.5
80
115 == largest value
44.5
```



### Auto-valutazione

13. Cosa visualizza il programma `LargestInArray` se vengono forniti in ingresso i dati seguenti?  
20 10 20 Q
14. Scrivete un ciclo che conti quanti elementi di un array sono uguali a zero.
15. Esaminate l'algoritmo che trova l'elemento di valore massimo in un array. Perché non abbiamo inizializzato `largest` e `i` a zero, come nel codice seguente?  
`double largest = 0;`

```
for (int i = 0; i < values.length; i++)
{
    if (values[i] > largest)
    {
        largest = values[i];
    }
}
```

16. Visualizzando separatori tra gli elementi di un array, abbiamo evitato di inserire un separatore prima del primo elemento. Riscrivete quel ciclo in modo che venga visualizzato un separatore *dopo* ogni elemento, tranne l'ultimo.

17. Per la visualizzazione di un array con separatori tra gli elementi, cosa c'è di sbagliato negli enunciati seguenti?

```
System.out.print(values[0]);
for (int i = 1; i < values.length; i++)
{
    System.out.print(", " + values[i]);
}
```

18. Per cercare la posizione di una corrispondenza abbiamo usato un ciclo while, non un ciclo for. Cosa c'è di sbagliato nell'utilizzo del ciclo seguente?

```
for (pos = 0; pos < values.length && !found; pos++)
{
    if (values[pos] > 100)
    {
        found = true;
    }
}
```

19. Per inserire un elemento in un array abbiamo spostato gli elementi aventi indice maggiore, a partire dalla fine dell'array. Cosa c'è di sbagliato nell'utilizzo del ciclo seguente, che parte dalla posizione di inserimento?

```
for (int i = pos; i < currentSize; i++)
{
    values[i + 1] = values[i];
}
```

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R7.16, R7.19 e E7.7, al termine del capitolo.



---

## Errori comuni 7.3

### Sottovalutare la dimensione di un insieme di dati

Sottovalutare la quantità di dati che l'utente inserirà in un programma è un errore di programmazione che si fa spesso. Supponete di scrivere un programma che cerchi una porzione di testo in un file. Inserite ciascuna riga del file in una stringa e create un array di stringhe. Quale dimensione deve avere? Sicuramente nessuno userà il programma con un testo avente più di 100 righe. Davvero? Uno scaltro collaudatore del programma può facilmente inserire l'intero testo di *Alice nel paese delle meraviglie* o di *Guerra e pace* (che sono disponibili in Internet). All'improvviso il vostro programma si trova a dover gestire decine o centinaia di migliaia di righe: dovete fare in modo che accetti dati in ingresso di dimensioni maggiori, oppure potete decidere di rifiutare educatamente i dati in eccesso.



## Argomenti avanzati 7.2

### Ordinare usando la libreria di Java

Ordinare un array in modo efficiente non è un compito facile. Nel Capitolo 13 vedrete come implementare algoritmi di ordinamento efficienti, ma, nel frattempo, potete usare la libreria di Java, che mette a disposizione un metodo adeguato, `sort`.

Potete ordinare l'array `values` con questa invocazione:

```
Arrays.sort(values);
```

Se l'array è riempito solo in parte, usate questa variante:

```
Arrays.sort(values, 0, currentSize);
```

## 7.4 Problem Solving: adattamento di algoritmi

**Combinando algoritmi fondamentali si possono risolvere problemi di programmazione complessi.**

Nel Paragrafo 7.3 avete visto alcuni algoritmi fondamentali per l'elaborazione di array: si tratta dei blocchi elementari con cui costruire molti programmi che elaborano array. In generale, avere a disposizione un buon repertorio di algoritmi di base, da poter combinare e adattare, è una valida strategia per risolvere i problemi.

Consideriamo questo esempio: avete i punteggi ottenuti da uno studente in alcuni questionari di valutazione e dovete eliminare il voto più basso, per poi calcolare la valutazione finale, costituita dalla somma di tutti gli altri voti. Se, ad esempio, i voti ottenuti sono:

8 7 8.5 9.5 7 4 10

allora la valutazione finale è  $8 + 7 + 8.5 + 9.5 + 7 + 10 = 50$ .

Non disponiamo di un algoritmo “pronto all’uso” per gestire questa situazione, però possiamo cercare, tra gli algoritmi che conosciamo, quali risolvono problemi simili. Ad esempio:

- calcolo della somma (Paragrafo 7.3.2);
- individuazione dell’elemento minimo (Paragrafo 7.3.3);
- eliminazione di un elemento (Paragrafo 7.3.6).

Possiamo, ora, decidere un piano d’azione che combini questi algoritmi:

Trova il minimo.

Eliminalo dall’array.

Calcola la somma.

Proviamo a farlo con il nostro esempio. Il valore minimo tra i voti seguenti:

|     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
| 8   | 7   | 8.5 | 9.5 | 7   | 4   | 10  |

è 4. Come lo rimuoviamo?

Abbiamo un problema. L'algoritmo di eliminazione di un elemento, visto nel Paragrafo 7.3.6, individua l'elemento da rimuovere mediante la sua *posizione*, non il suo valore.

Ci serve un altro degli algoritmi che conosciamo:

- ricerca lineare (Paragrafo 7.3.5).

Conseguentemente, dobbiamo aggiustare il nostro piano d'azione:

Trova il valore minimo.

Trova la sua posizione.

Elimina tale posizione dall'array.

Calcola la somma.

Funzionerà? Proseguiamo con il nostro esempio.

Dopo aver scoperto che il valore minimo è 4, la ricerca lineare ci dice che il valore 4 si trova nella posizione 5.

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|-----|-----|-----|-----|-----|-----|-----|
| 8   | 7   | 8.5 | 9.5 | 7   | 4   | 10  |

Eliminiamo l'elemento che si trova in posizione 5:

| [0] | [1] | [2] | [3] | [4] | [5] |
|-----|-----|-----|-----|-----|-----|
| 8   | 7   | 8.5 | 9.5 | 7   | 10  |

Infine, calcoliamo la somma degli elementi rimasti:  $8 + 7 + 8.5 + 9.5 + 7 + 10 = 50$ .

L'analisi del funzionamento dell'algoritmo con questo esempio, compiuta passo dopo passo, può essere sufficiente per dimostrare che la strategia funziona.

Possiamo, però, trovare una soluzione migliore? Non sembra molto efficiente trovare il valore minimo e, poi, scandire di nuovo l'array per trovare la sua posizione.

Possiamo adattare l'algoritmo che trova l'elemento minimo in modo che tenga traccia della sua posizione. Questo è l'algoritmo originale:

```
double smallest = values[0];
for (int i = 1; i < values.length; i++)
{
    if (values[i] < smallest)
    {
        smallest = values[i];
    }
}
```

Bisogna conoscere bene gli algoritmi fondamentali, per poterli adattare a situazioni simili.

Quando troviamo un elemento che diventa il nuovo minimo, aggiorniamo anche la posizione del minimo trovato fino a quel momento:

```
if (values[i] < smallest)
{
    smallest = values[i];
```

```
    smallestPosition = i;
}
```

A questo punto, però, non c'è più bisogno di memorizzare il valore minimo corrente: è semplicemente `values[smallestPosition]`. Con questa osservazione, possiamo modificare nuovamente l'algoritmo, in questo modo:

```
int smallestPosition = 0;
for (int i = 1; i < values.length; i++)
{
    if (values[i] < values[smallestPosition])
    {
        smallestPosition = i;
    }
}
```

Con questa modifica, il nostro problema viene risolto dalla seguente strategia:

- Trova la posizione del valore minimo.
- Elimina tale posizione dall'array.
- Calcola la somma.

Il paragrafo successivo illustrerà una tecnica per individuare un algoritmo quando nessuno degli algoritmi fondamentali che abbiamo visto può essere adattato per risolvere il problema.



## Auto-valutazione

20. Il Paragrafo 7.3.6 ha presentato due diversi algoritmi per l'eliminazione di un elemento da un array. Quale dei due è il più adatto alla soluzione del problema descritto in questo paragrafo?
21. Nel problema appena discusso, per calcolare la somma finale non è necessario *eliminare* effettivamente il valore minimo dall'array: descrivete un algoritmo alternativo.
22. Usando uno o più algoritmi tra quelli visti nel Paragrafo 6.7, come si può visualizzare il numero di valori negativi e il numero di valori positivi presenti in un array di numeri?
23. Come si possono visualizzare tutti i valori positivi presenti in un array di numeri, separati da una virgola?
24. Analizzate il seguente algoritmo, che costruisce un array contenente tutti i valori presenti in un altro array che soddisfano una determinata condizione:

```
int matchesSize = 0;
for (int i = 0; i < values.length; i++)
{
    if (values[i] soddisfa la condizione)
    {
        matches[matchesSize] = values[i];
        matchesSize++;
    }
}
```

Come vi può essere d'aiuto per risolvere il problema posto nella domanda precedente?

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi R7.25 e R7.26, al termine del capitolo.



## Consigli pratici 7.1

### Lavorare con array

In molti problemi di elaborazione di dati c'è bisogno di utilizzare sequenze di valori. Questa sezione descrive le fasi utili per memorizzare in un array una sequenza di valori acquisiti come dati in ingresso, per poi elaborare i suoi elementi.

**Problema.** Prendiamo nuovamente in esame il problema visto nel Paragrafo 7.4: il punteggio complessivo di una serie di questionari si ottiene sommando tutti i singoli punteggi, escludendo il minimo. Se, ad esempio, i punteggi nei singoli questionari sono 8, 4, 8.5, 9.5, 7, 5 e 10, il punteggio complessivo è 50.

#### Fase 1 Scomponete il problema in sotto-problemi

Di solito il problema da risolvere viene scomposto in più sotto-problemi, ad esempio così

- Acquisire i dati, scrivendoli in un array.
- Elaborare i dati (in uno o più passi).
- Visualizzare i risultati.

Per decidere come elaborare i dati dovete tenere ben presenti gli algoritmi visti nel Paragrafo 7.3: la maggior parte dei problemi può essere risolta usando uno o più di tali algoritmi.

Nel nostro esempio, leggeremo i dati, quindi elimineremo il voto minore e calcoleremo il totale. Ad esempio, se i voti sono 8 7 8.5 9.5 7 5 10, eliminiamo il voto minimo, che è 5, ottenendo la sequenza 8 7 8.5 9.5 7 10: la somma di questi valori è il voto finale, 50.

Abbiamo così individuato tre passi:

- Acquisisci i dati.
- Elimina il valore minimo.
- Calcola la somma.

#### Fase 2 Determinate gli algoritmi che servono

A volte capita che uno dei passi individuati corrisponda esattamente a uno degli algoritmi elementari del Paragrafo 7.3, come avviene, nel nostro caso, con il calcolo della somma finale (Paragrafo 7.3.2) e l'acquisizione dei dati (Paragrafo 7.3.10). Altre volte occorrerà combinare più algoritmi: per eliminare il voto minimo dobbiamo trovare il valore minimo (Paragrafo 7.3.3), individuare la sua posizione (Paragrafo 7.3.5) e, infine, eliminare l'elemento che si trova in quella posizione (Paragrafo 7.3.6).

Aumentiamo, quindi, il livello di dettaglio dello pseudocodice, in questo modo:

- Acquisisci i dati.
- Trova il valore minimo.

Trova la sua posizione.  
 Elimina l'elemento in tale posizione.  
 Calcola la somma.

Questo progetto funziona, l'abbiamo già visto nel Paragrafo 7.4, ma c'è una strada alternativa: è semplice calcolare la somma di tutti i valori e, poi, sottrarre il valore minimo al totale ottenuto. In questo modo possiamo evitare di determinare la posizione del minimo e di eliminarlo:

Acquisisci i dati.  
 Trova il valore minimo.  
 Calcola la somma.  
 Sottrai il valore minimo.

### Fase 3 Strutturate il programma usando classi e metodi

Anche se sarebbe possibile realizzare tutti i passi dell'algoritmo all'interno del metodo `main`, raramente questa è una buona soluzione: è molto meglio realizzare ciascun passo in un metodo diverso, così come è bene progettare una classe che abbia il compito di memorizzare ed elaborare i dati.

Nel nostro esempio memorizziamo i voti all'interno della classe `Student`: uno studente ha un array di voti.

```
public class Student
{
    private double[] scores;
    private double scoresSize;

    ...
    public Student(int capacity) { . . . }
    public boolean addScore(double score) { . . . }
    public double finalScore() { . . . }
}
```

Un'altra classe, `ScoreAnalyzer`, avrà il compito di acquisire i dati introdotti dall'utente e di visualizzare il risultato. Il suo metodo `main` invoca i metodi della classe `Student`:

```
Student fred = new Student(100);
System.out.println("Please enter values, Q to quit:");
while (in.hasNextDouble())
{
    if (!fred.addScore(in.nextDouble()))
    {
        System.out.println("Too many scores.");
        return;
    }
}
System.out.println("Final score: " + fred.finalScore());
```

Gran parte del lavoro deve ora essere svolto dal metodo `finalScore`, ma non è giusto che tutto sia compito suo: definiamo, invece, alcuni metodi ausiliari

```
public double sum()
public double minimum()
```

che realizzano, rispettivamente, gli algoritmi visti nel Paragrafo 7.3.2 e 7.3.3. Possiamo finalmente scrivere il metodo `finalScore`:

```
public double finalScore()
{
    if (scoresSize == 0)
    {
        return 0;
    }
    else if (scoresSize == 1)
    {
        return scores[0];
    }
    else
    {
        return sum() - minimum();
    }
}
```

#### Fase 4 Completate il programma e collaudatelo

Inserite i metodi nella classe. Riguardate il codice che avete scritto e verificate di aver gestito sia i casi normali sia le condizioni eccezionali. Cosa accade con un array vuoto? E con uno che contiene un solo elemento? E quando non si trova alcuna corrispondenza con il valore cercato? E quando le corrispondenze sono multiple? Esamineate tali condizioni limite e assicuratevi che il vostro programma funzioni correttamente.

Nel nostro esempio, se il vettore è vuoto non è possibile trovare il valore minimo al suo interno: in tal caso, dobbiamo terminare l'esecuzione del programma *prima* di invocare il metodo `minimum`.

Cosa succede se il valore minimo è ripetuto più volte? Significa che lo studente ha più prove d'esame con lo stesso voto, pari al minimo dei suoi voti. Il nostro codice sottrae al totale di tutti i voti una sola delle occorrenze di voto minimo e questo è proprio il comportamento voluto.

La tabella che segue elenca alcuni casi di prova e i risultati previsti.

| Caso di prova      | Risultato previsto | Commento                                             |
|--------------------|--------------------|------------------------------------------------------|
| 8 7 8.5 9.5 7 5 10 | 50                 | Si veda la Fase 1.                                   |
| 8 7 7 9            | 24                 | Viene eliminata una sola occorrenza del voto minimo. |
| 8                  | 8                  | Se c'è un solo voto, il voto minimo non va rimosso.  |
| (Nessun ingresso)  | <b>Errore</b>      | Situazione non valida.                               |

Nel pacchetto dei file scaricabili per questo libro, la cartella `how_to_1` del Capitolo 7 contiene il codice sorgente completo della classe.



## Esempi completi 7.1

### Lancio di dadi

**Problema.** Dovete decidere se un dado è equo, contando la frequenza con cui, lanciandolo, compaiono le diverse facce: 1, 2, ..., 6. I dati in ingresso sono una sequenza di valori ottenuti con ripetuti lanci del dado. Dovete visualizzare una tabella che riporti la frequenza rilevata per ciascuna faccia del dado.

#### Fase 1 Scomponete il problema in sotto-problemi

Il nostro primo tentativo di scomposizione è una semplice copiatura della descrizione del problema:

- Acquisisci i valori dei lanci del dado.
- Conta le occorrenze dei singoli valori: 1, 2, ..., 6.
- Visualizza i conteggi.

Ma possiamo analizzare il problema un po' meglio. Questa scomposizione suggerisce che, mentre leggiamo i valori in ingresso, li memorizziamo tutti in una lista, per poi elaborarli nella seconda fase. Ma è davvero necessario memorizzarli? In fin dei conti vogliamo soltanto sapere quante volte compare ciascuna faccia. Se utilizziamo un array di contatori, uno per ogni faccia, possiamo ignorare ciascun dato in ingresso dopo aver incrementato il relativo contatore.

Questo miglioramento ci porta allo pseudocodice seguente:

- Per ciascun valore acquisito in ingresso
- Incrementa il contatore corrispondente.
- Visualizza i contatori.

#### Fase 2 Determinate gli algoritmi che servono

Non disponiamo di un algoritmo “pronto all’uso” per acquisire dati in ingresso e incrementare un contatore, ma progettarne uno è davvero semplice. Supponiamo di aver acquisito un valore e di averlo memorizzato nella variabile `value`: è un numero intero compreso tra 1 e 6. Avendo predisposto una lista di contatori, `counters`, di lunghezza 6, eseguiamo semplicemente:

```
counters[value - 1]++;
```

In alternativa, possiamo usare un array con sette valori, “sprecando” l’elemento `counters[0]`. Questo trucco rende più semplice l’aggiornamento dei contatori, perché, dopo aver acquisito un valore, eseguiamo:

```
counters[value]++; // value tra 1 e 6
```

All’inizio, ovviamente, bisogna creare l’array in questo modo:

```
counters = new int[sides + 1];
```

Perché abbiamo introdotto la variabile `sides` per rappresentare il numero di facce di un dado? Per considerare la possibilità che si voglia usare il programma anche per analizzare, ad esempio, dadi a 12 facce: in tal caso, basterà modificare il valore di `sides`, portandolo a 12.

L'unica cosa che rimane da fare è la visualizzazione dei conteggi, che, ad esempio, si potrebbe presentare così:

```
1: 3  
2: 3  
3: 2  
4: 2  
5: 2  
6: 0
```

Non conosciamo un algoritmo che visualizzi le informazioni proprio in questa forma, ma il compito è simile a quello risolto dal ciclo elementare che visualizza tutti gli elementi di una lista:

```
for (int element : counters)  
{  
    System.out.println(element);  
}
```

Tuttavia, questo ciclo non è adatto allo scopo, per due motivi. Innanzitutto visualizza anche l'elemento inutilizzato, associato all'indice 0. Se vogliamo evitare tale elemento, non possiamo usare un ciclo `for` esteso. Ci serve, invece, un normale ciclo `for` controllato da contatore, come questo:

```
for (int i = 1; i < counters.length; i++)  
{  
    System.out.println(counters[i]);  
}
```

Questo ciclo visualizza i valori dei contatori, ma non rispetta appieno il formato che abbiamo deciso. Vogliamo che ci siano anche i corrispondenti valori delle facce:

```
for (int i = 1; i < counters.length; i++)  
{  
    System.out.printf("%2d: %4d\n", i, counters[i]);  
}
```

### Fase 3 Strutturate il programma usando classi e metodi

Scrivremo un metodo per ciascuna fase:

- `void countInputs()`
- `void printCounters()`

Il metodo `main`, poi, invocherà tali metodi:

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

```

    {
        final int SIDES = 6;
        Dice dice = new Dice(SIDES);
        dice.countInputs();
        dice.printCounters();
    }
}

```

Il metodo `countInputs` acquisisce tutti i valori in ingresso e, per ciascun valore letto, incrementa il contatore corrispondente. Il metodo `printCounters` visualizza il valore di ciascuna faccia e il relativo contatore, come già descritto.

#### Fase 4 Completate il programma e collaudatelo

Al termine del paragrafo trovate il codice del programma completo e c'è un solo dettaglio che non è stato discusso in precedenza. Quando aggiorniamo un contatore, in questo modo:

```
counters[value]++;
```

vogliamo essere certi che l'utente non abbia fornito un dato errato in ingresso, che provocherebbe un errore di limiti nell'accesso alla lista. Per questo motivo rifiutiamo i dati che siano minori di uno o maggiori di `sides`.

La tabella seguente mostra alcuni casi di prova e il corrispondente risultato previsto. Per risparmiare spazio, nelle informazioni visualizzate dal programma riportiamo soltanto i valori dei contatori.

| Caso di prova     | Risultato previsto | Commento                                                                      |
|-------------------|--------------------|-------------------------------------------------------------------------------|
| 1 2 3 4 5 6       | 1 1 1 1 1 1        | Ogni valore ricorre una volta sola.                                           |
| 1 2 3             | 1 1 1 0 0 0        | I valori che non compaiono devono avere conteggio zero.                       |
| 1 2 3 1 2 3 4     | 2 2 2 1 0 0        | I contatori devono essere coerenti con le occorrenze dei valori in ingresso.  |
| (Nessun ingresso) | 0 0 0 0 0 0        | Si tratta di una sequenza di ingresso lecita: tutti i contatori valgono zero. |
| 1 2 3 4 5 6 7     | <b>Errore</b>      | Tutti i valori d'ingresso devono essere compresi tra 1 e 6.                   |

Ecco il programma completo.

#### File Dice.java

```

1 import java.util.Scanner;
2
3 /**
4  * Questa classe legge una sequenza di risultati di lanci di un dado
5  * e conta le occorrenze di ciascuna faccia.
6 */
7 public class Dice
8 {

```

```
9  private int[] counters;
10
11 public Dice(int sides)
12 {
13     counters = new int[sides + 1]; // counters[0] non viene usato
14 }
15
16 public void countInputs()
17 {
18     System.out.println("Please enter values, Q to quit:");
19     Scanner in = new Scanner(System.in);
20     while (in.hasNextInt())
21     {
22         int value = in.nextInt();
23
24         // incrementa il contatore corrispondente al valore acquisito
25
26         if (1 <= value && value < counters.length)
27         {
28             counters[value]++;
29         }
30         else
31         {
32             System.out.println(value + " is not a valid input.");
33         }
34     }
35 }
36
37 public void printCounters()
38 {
39     for (int i = 1; i < counters.length; i++)
40     {
41         System.out.printf("%2d: %4d\n", i, counters[i]);
42     }
43 }
44 }
```

### File DiceAnalyzer.java

```
1  public class DiceAnalyzer
2  {
3      public static void main(String[] args)
4      {
5          final int SIDES = 6;
6          Dice dice = new Dice(SIDES);
7          dice.countInputs();
8          dice.printCounters();
9      }
10 }
```

### Esecuzione del programma

```
Please enter values, Q to quit:
1 2 3 1 2 3 4 Q
1: 2
2: 2
```

3: 2  
4: 1  
5: 0  
6: 0

## 7.5 Problem Solving: progettare algoritmi facendo esperimenti

Nel Paragrafo 7.4 avete visto come risolvere un problema combinando insieme e adattando algoritmi noti, ma cosa facciamo se, per il problema che dobbiamo risolvere, non ci sono algoritmi standard da utilizzare? In questo paragrafo apprenderete una tecnica che consente di individuare algoritmi facendo esperimenti concreti e manipolando oggetti reali.

Consideriamo questo problema: dato un array la cui dimensione è un numero pari, dobbiamo scambiare la sua prima metà con la sua seconda metà. Se, ad esempio, l'array contiene questi otto numeri:

|   |    |    |   |    |   |   |   |
|---|----|----|---|----|---|---|---|
| 9 | 13 | 21 | 4 | 11 | 7 | 1 | 3 |
|---|----|----|---|----|---|---|---|

deve diventare:

|    |   |   |   |   |    |    |   |
|----|---|---|---|---|----|----|---|
| 11 | 7 | 1 | 3 | 9 | 13 | 21 | 4 |
|----|---|---|---|---|----|----|---|

Per molti studenti risulta abbastanza complicato individuare un algoritmo che risolva questo problema. Probabilmente intuiranno che serve un ciclo e che alcuni elementi andranno inseriti nell'array (Paragrafo 7.3.7) o scambiati tra loro (Paragrafo 7.3.8), ma tali intuizioni potrebbero non essere sufficienti per disegnare un diagramma di flusso o per descrivere un algoritmo mediante pseudocodice.

Una tecnica che si è dimostrata utile per la progettazione di algoritmi è la manipolazione di oggetti reali e concreti. Iniziate allineando oggetti che rappresentino gli elementi della lista: una buona idea è utilizzare monete, carte da gioco o piccoli giocattoli.

In questo caso usiamo otto monete:

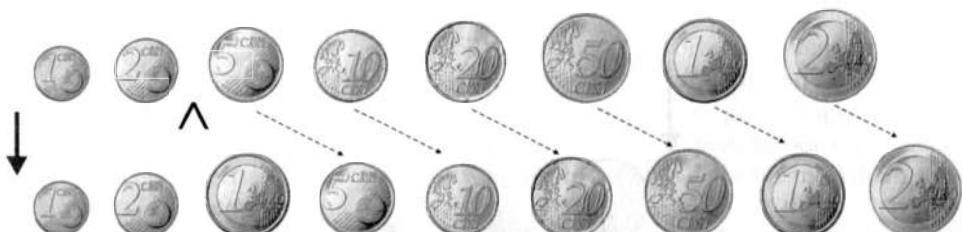


Ora fermiamoci un attimo e cerchiamo di capire quali mosse possiamo compiere, in generale, per modificare l'ordinamento delle monete. Ad esempio, possiamo eliminare una moneta (Paragrafo 7.3.6):

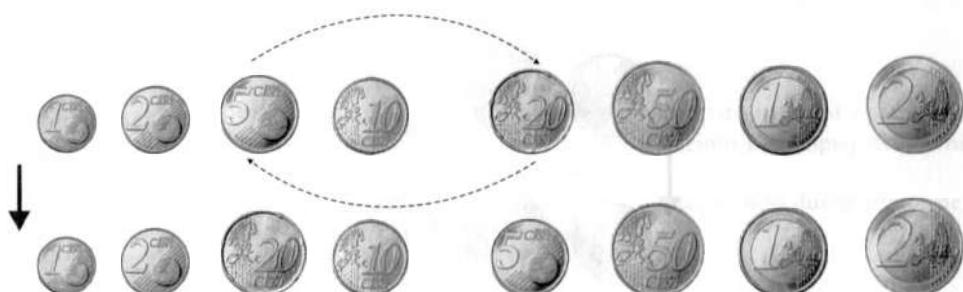


Per visualizzare un array di valori usate una sequenza di monete, di carte da gioco o di piccoli giocattoli.

Oppure possiamo inserire una moneta (Paragrafo 7.3.7):



O, ancora, possiamo scambiare due monete (Paragrafo 7.3.8):



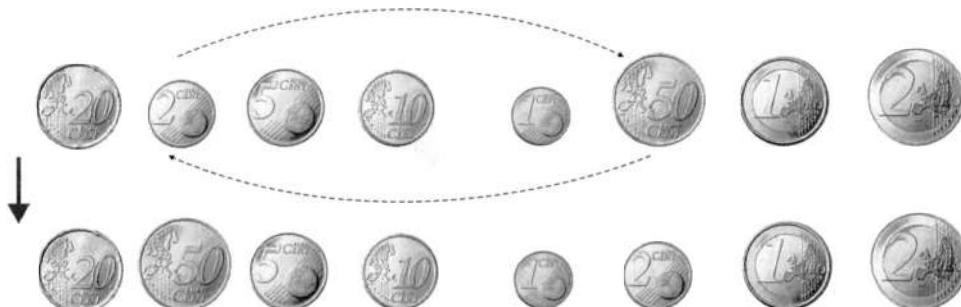
Proseguite gli esperimenti: allineate alcune monete ed eseguite queste tre operazioni fino a quando non le sentite vostre.

Ora chiediamoci come questo possa aiutarci nella soluzione del nostro problema, che, ricordiamo, è quello di scambiare la prima metà dell'array con la sua seconda metà.

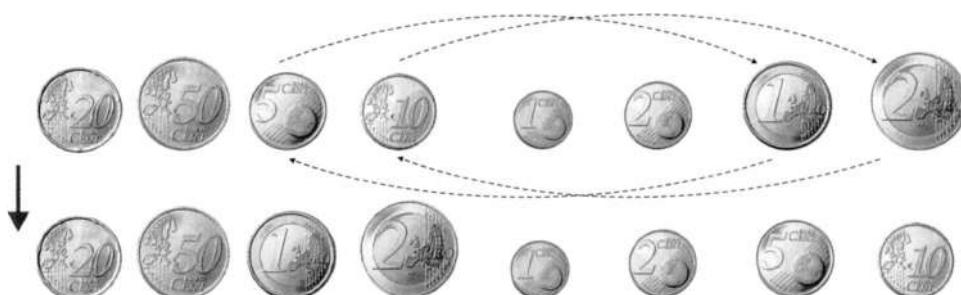
Iniziamo sistemando la prima moneta, obiettivo che si raggiunge scambiandola con la quinta. Come programmatore Java, però, diremo che lo scambio avviene tra le monete che si trovano nelle posizioni 0 e 4:



Successivamente, scambiamo tra loro le monete che si trovano nelle posizioni 1 e 5:



Ancora due scambi e abbiamo finito:



A questo punto l'algoritmo diventa evidente:

```
i = 0
j = ... (lo vedremo tra un attimo)
Finché ... (non conosciamo ancora la condizione)
    Scambia gli elementi nelle posizioni i e j
    i++
    j++
```

Quale valore iniziale dobbiamo assegnare alla variabile  $j$ ? Quando le monete sono otto, la moneta in posizione 0 viene scambiata con quella in posizione 4; in generale, la moneta in posizione 0 viene scambiata con quella che si trova al centro dell'array, o, più precisamente, in posizione  $\text{lunghezza} / 2$ .

E quante iterazioni bisogna eseguire? Dobbiamo scambiare tutte le monete che si trovano nella prima metà dell'array, cioè dobbiamo scambiare un numero di monete uguale a  $\text{lunghezza} / 2$ . Lo pseudocodice, quindi, si completa così:

```
i = 0
j =  $\text{lunghezza} / 2$ 
Finché  $i < \text{lunghezza} / 2$ 
    Scambia gli elementi nelle posizioni i e j
    i++
    j++
```

Per tenere traccia degli spostamenti delle variabili che indicano le posizioni nell'array si possono usare graffette o altri contrassegni.

A questo punto è bene eseguire lo pseudocodice passo dopo passo, tenendo traccia dei valori delle variabili e del contenuto dell'array (come detto nel Paragrafo 6.2), magari usando graffette per indicare le posizioni rappresentate dai valori delle variabili *i* e *j* all'interno dell'array. Se questa analisi si dimostrerà corretta, sapremo che nello pseudocodice non sono presenti gli insidiosi errori "per scarto di uno". In effetti, la domanda di auto-valutazione numero 25 vi chiederà di svolgere proprio questa attività, mentre nell'Esercizio E7.8 vi sarà proposto di tradurre in Java lo pseudocodice. L'Esercizio R7.27, invece, suggerisce un diverso algoritmo per lo scambio delle due metà di un array, eliminando e inserendo ripetutamente elementi.

Molte persone ritengono che la manipolazione concreta di oggetti metta meno in soggezione del disegno di diagrammi o della progettazione di algoritmi in astratto. Provateci anche voi, quando vi troverete a progettare nuovi algoritmi!



## Auto-valutazione

25. Eseguite, passo dopo passo, l'algoritmo che abbiamo sviluppato in questo paragrafo, usando due graffette per tenere traccia, sulla carta, delle posizioni *i* e *j*. Spiegate perché nello pseudocodice non sono presenti errori di limiti.
26. Prendete delle monete e simulate lo pseudocodice seguente, usando due graffette per indicare le posizioni *i* e *j*. Cosa fa questo algoritmo?

```
i = 0.  
j = lunghezza - 1.  
Finché i < j  
    Scambia gli elementi nelle posizioni i e j.  
    i++.  
    j--.
```

27. Prendete in esame il problema di riordinare gli elementi di un array in modo che i numeri pari vengano prima di quelli dispari, mentre, per tutte le altre condizioni, l'ordine non è importante. Ad esempio, l'array:

**1 4 14 2 1 3 5 6 23**

potrebbe essere riordinato in questo modo:

**4 2 14 6 1 5 3 23 1**

Usando monete e graffette, progettate un algoritmo che risolva questo problema scambiando elementi nell'array, poi descrivetelo mediante pseudocodice.

28. Progettate un algoritmo che risolva il problema descritto nella domanda precedente usando eliminazioni e inserimenti invece di usare scambi.
29. Prendete in esame l'algoritmo, visto nel Paragrafo 6.7.5, che trova l'elemento di valore massimo presente in una sequenza di dati in ingresso (*non* in un array). Perché questo algoritmo si visualizza meglio scegliendo carte da gioco da un mazzo piuttosto che disponendo soldatini allineati?

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi R7.27, R7.28 e E7.8, al termine del capitolo.

## 7.6 Array bidimensionali

Spesso capita di voler memorizzare raccolte di valori che, per loro natura, hanno una disposizione bidimensionale: insiemi di dati di questo tipo sono frequenti in applicazioni di carattere economico-finanziario e scientifico. In informatica, chiamiamo **array bidimensionale** o *matrice* una disposizione di valori costituita da righe e colonne.

Vediamo come si possano memorizzare i dati riportati nella Figura 13: il numero di medaglie vinte nel pattinaggio artistico alle Olimpiadi Invernali del 2014.

**Figura 13**  
Medaglie vinte  
nel pattinaggio artistico



|                       | Oro | Argento | Bronzo |
|-----------------------|-----|---------|--------|
| Canada                | 0   | 3       | 0      |
| Italia                | 0   | 0       | 1      |
| Germania              | 0   | 0       | 1      |
| Giappone              | 1   | 0       | 0      |
| Kazakistan            | 0   | 0       | 1      |
| Russia                | 3   | 1       | 1      |
| Corea del Sud         | 0   | 1       | 0      |
| Stati Uniti d'America | 1   | 0       | 1      |

### 7.6.1 Dichiarazione di array bidimensionali

Per memorizzare i dati di una tabella  
si usa un array bidimensionale.

Per creare, in Java, un array bidimensionale, bisogna definirne il numero di righe e il numero di colonne. Ad esempio, `new int[8][3]` costruisce un array con otto righe e tre colonne e il riferimento a un tale array va memorizzato in una variabile di tipo `int[][]`. Ecco la dichiarazione completa di un array bidimensionale adatto a memorizzare i nostri dati sul numero di medaglie:

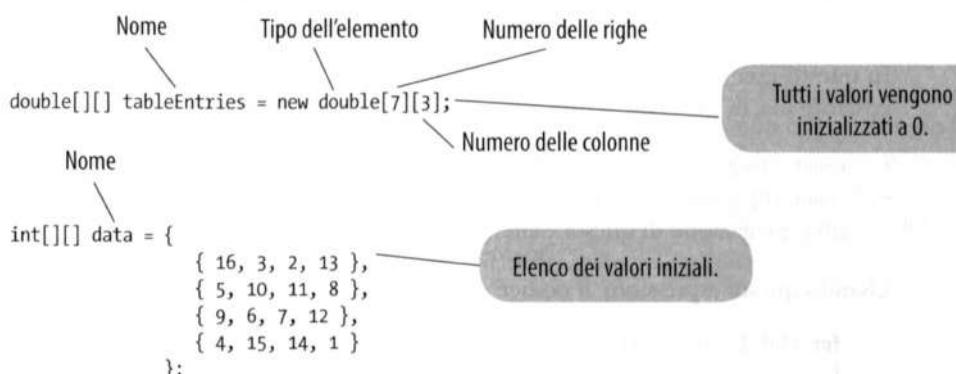
```
final int COUNTRIES = 8;
final int MEDALS = 3;
int[][] counts = new int[COUNTRIES][MEDALS];
```

In alternativa, si può inizializzare l'array contestualmente alla sua dichiarazione, raggruppando opportunamente i dati relativi a ciascuna riga:

```
int[][] counts =
{
    { 0, 3, 0 },
    { 0, 0, 1 },
    { 0, 0, 1 },
    { 1, 0, 0 },
    { 0, 0, 1 },
    { 3, 1, 1 },
    { 0, 1, 0 },
    { 1, 0, 1 }
};
```

## Sintassi di Java

### 7.3 Dichiarazione di array bidimensionale



Come per gli array monodimensionali, nessuna delle due dimensioni di un array bidimensionale può essere modificata dopo la sua creazione.

#### 7.6.2 Accesso agli elementi

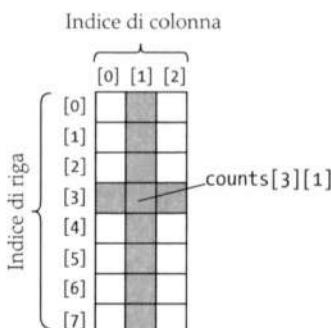
Ai singoli elementi di un array bidimensionale si accede usando due indici: `array[i][j]`.

Per accedere a un determinato elemento di un array bidimensionale bisogna specificare, in due coppie di parentesi quadre separate, il valore di due indici, che selezionano, rispettivamente, la riga e la colonna (si veda la Figura 14):

```
int medalCount = counts[3][1];
```

**Figura 14**

Accesso agli elementi di un array bidimensionale



Per accedere ordinatamente a tutti gli elementi di un array bidimensionale si usano due cicli annidati. Ad esempio, questo ciclo visualizza tutti gli elementi di `counts`:

```
for (int i = 0; i < COUNTRIES; i++)
{
    // Elabora la i-esima riga
    for (int j = 0; j < MEDALS; j++)
    {
        // Elabora la j-esima colonna della i-esima riga
    }
}
```

```

        System.out.printf("%8d", counts[i][j]);
    }
    System.out.println() // va a capo al termine della riga visualizzata
}

```

In questo esempio il numero di righe e il numero di colonne sono dati sotto forma di costanti, ma, in generale, si possono usare le espressioni seguenti:

- `counts.length` è il numero di righe
- `counts[0].length` è il numero di colonne (la sezione Argomenti avanzati 7.3 fornisce una spiegazione di questa espressione)

Usando queste espressioni, il codice dei cicli annidati visti in precedenza diventa:

```

for (int i = 0; i < counts.length; i++)
{
    for (int j = 0; j < counts[0].length; j++)
    {
        System.out.printf("%8d", counts[i][j]);
    }
    System.out.println()
}

```

### 7.6.3 Individuazione degli elementi adiacenti

Alcuni programmi che elaborano array bidimensionali hanno bisogno di individuare gli elementi adiacenti a un determinato elemento: un'attività molto frequente nei giochi. La Figura 15 mostra come si calcolano i valori degli indici degli elementi adiacenti all'elemento individuato da `[i][j]`.

**Figura 15**

Le posizioni adiacenti a un elemento in un array bidimensionale

| $[i - 1][j - 1]$ | $[i - 1][j]$ | $[i - 1][j + 1]$ |
|------------------|--------------|------------------|
| $[i][j - 1]$     | $[i][j]$     | $[i][j + 1]$     |
| $[i + 1][j - 1]$ | $[i + 1][j]$ | $[i + 1][j + 1]$ |

Ad esempio, gli elementi adiacenti a `counts[3][1]` a sinistra e a destra sono, rispettivamente, `counts[3][0]` e `counts[3][2]`, mentre gli elementi adiacenti in alto e in basso sono `counts[2][1]` e `counts[4][1]`.

Quando si individuano gli elementi adiacenti bisogna fare attenzione ai confini della tabella. Ad esempio, `counts[0][1]` non ha alcun elemento adiacente “verso l’alto”. Consideriamo il problema di calcolare la somma dei valori degli elementi adiacenti a `counts[i][j]` verso l’alto e verso il basso. Bisogna controllare se l’elemento in esame si trova in cima o in fondo all’array:

```

int total = 0;
if (i > 0) { total = total + counts[i - 1][j]; }

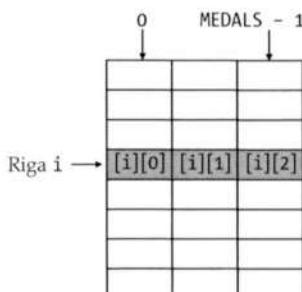
```

```
if (i < ROWS - 1) { total = total + counts[i + 1][j]; }
```

#### 7.6.4 Accedere a righe e colonne

Un problema frequente è quello di accedere ai valori contenuti in una riga o in una colonna, ad esempio per sommarli o per trovare l'elemento massimo o minimo in una riga o in una colonna. Nell'array che stiamo usando come esempio la somma di una riga ha come risultato il numero di medaglie conquistate da una nazione.

Capire quali siano i valori corretti per gli indici richiede qualche ragionamento ed è bene fare uno schizzo della tabella. Per calcolare la somma dei valori della riga  $i$  dobbiamo ispezionare i seguenti elementi:

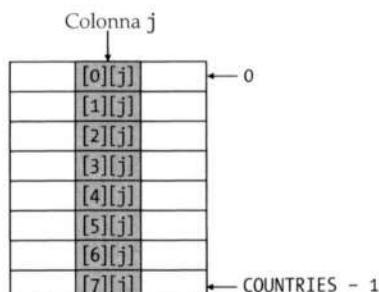


Come potete vedere, dobbiamo calcolare la somma di caselle del tipo `counts[i][j]`, con  $j$  che varia da 0 a  $\text{MEDALS} - 1$ , estremi inclusi. Questo ciclo calcola la somma richiesta:

```
int total = 0;
for (int j = 0; j < MEDALS; j++)
{
    total = total + counts[i][j];
}
```

Il calcolo della somma di una colonna è analogo: si sommano caselle del tipo `counts[i][j]`, con  $i$  che varia da 0 a  $\text{COUNTRIES} - 1$ .

```
int total = 0;
for (int i = 0; i < COUNTRIES; i++)
{
    total = total + counts[i][j];
}
```



Il programma che segue illustra l'utilizzo di array bidimensionali: visualizza il medagliere usato in questo paragrafo come esempio, aggiungendo una colonna con i valori totali per nazione.

### File Medals.java

```

1  /**
2   * Visualizza un medagliere con i totali per nazione.
3  */
4 public class Medals
5 {
6     public static void main(String[] args)
7     {
8         final int COUNTRIES = 8;
9         final int MEDALS = 3;
10
11     String[] countries =
12     {
13         "Canada",
14         "Italy",
15         "Germany",
16         "Japan",
17         "Kazakhstan",
18         "Russia",
19         "South Korea",
20         "United States"
21     };
22
23     int[][] counts =
24     {
25         { 0, 3, 0 },
26         { 0, 0, 1 },
27         { 0, 0, 1 },
28         { 1, 0, 0 },
29         { 0, 0, 1 },
30         { 3, 1, 1 },
31         { 0, 1, 0 },
32         { 1, 0, 1 }
33     };
34
35     System.out.println("          Country    Gold  Silver  Bronze  Total");
36
37     // Visualizza righe con paese, medaglie e totale
38     for (int i = 0; i < COUNTRIES; i++)
39     {
40         // elabora la riga i-esima
41         System.out.printf("%15s", countries[i]);
42
43         int total = 0;
44
45         // visualizza gli elementi di una riga e aggiorna il totale
46         for (int j = 0; j < MEDALS; j++)
47         {
48             System.out.printf("%8d", counts[i][j]);
49             total = total + counts[i][j];

```

```

50 }
51
52     // visualizza il totale della riga e va a capo
53     System.out.printf("%8d\n", total);
54 }
55 }
56 }

```

### Esecuzione del programma

| Country       | Gold | Silver | Bronze | Total |
|---------------|------|--------|--------|-------|
| Canada        | 0    | 3      | 0      | 3     |
| Italy         | 0    | 0      | 1      | 1     |
| Germany       | 0    | 0      | 1      | 1     |
| Japan         | 1    | 0      | 0      | 1     |
| Kazakhstan    | 0    | 0      | 1      | 1     |
| Russia        | 3    | 1      | 1      | 5     |
| South Korea   | 0    | 1      | 0      | 1     |
| United States | 1    | 0      | 1      | 2     |



### Auto-valutazione

30. Che risultati si ottengono sommando i valori delle singole colonne nei dati usati come esempio in questo paragrafo?
31. Considerate un array  $8 \times 8$  da usare come scacchiera in un gioco da tavolo, così dichiarata:  
`int[][] board = new int[8][8];`  
Usando due cicli annidati, inizializzate la scacchiera in modo che contenga valori 0 e 1 alternati, come quella per il gioco degli scacchi:

```

0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
...
1 0 1 0 1 0 1 0

```

*Suggerimento:* Controllate se  $i + j$  è un numero pari.

32. Create un array bidimensionale che rappresenti la scacchiera del “gioco del tris” (nel mondo anglosassone, *tic-tac-toe*): deve avere tre righe e tre colonne e ciascuna casella può contenere le stringhe “x”, “o” oppure “ ”.
33. Scrivere un enunciato di assegnazione che inserisca una “x” nell’angolo superiore destro della scacchiera definita nella domanda precedente.
34. Quali sono gli elementi che si trovano sulla diagonale che collega la casella superiore sinistra e la casella inferiore destra nella scacchiera definita nella domanda 32?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R7.29, E7.16 e E7.17, al termine del capitolo.



## Esempi completi 7.2

### Una tabella con la popolazione mondiale

**Problema.** Data la seguente tabella con la popolazione mondiale suddivisa per continenti, vogliamo scrivere un programma che la visualizzi in un formato tabulare analogo, aggiungendo una riga in basso che riporti il totale colonna per colonna, cioè la popolazione mondiale negli anni indicati.

| Popolazione per continente (in milioni) |      |      |      |      |      |      |      |
|-----------------------------------------|------|------|------|------|------|------|------|
| Anno                                    | 1750 | 1800 | 1850 | 1900 | 1950 | 2000 | 2050 |
| Africa                                  | 106  | 107  | 111  | 133  | 221  | 767  | 1766 |
| Asia                                    | 502  | 635  | 809  | 947  | 1402 | 3634 | 5268 |
| Australia                               | 2    | 2    | 2    | 6    | 13   | 30   | 46   |
| Europa                                  | 163  | 203  | 276  | 408  | 547  | 729  | 628  |
| Nord America                            | 2    | 7    | 26   | 82   | 172  | 307  | 392  |
| Sud America                             | 16   | 24   | 38   | 74   | 167  | 511  | 809  |

**Fase 1** Per prima cosa, scomponete il problema in sotto-problemi

Inizializza i dati della tabella.

Visualizza la tabella.

Calcola e visualizza i totali per colonna.

**Fase 2** Iniziate la tabella come sequenza di righe

```
int[][] populations =
{
    { 106, 107, 111, 133, 221, 767, 1766 },
    { 502, 635, 809, 947, 1402, 3634, 5268 },
    { 2, 2, 2, 6, 13, 30, 46 },
    { 163, 203, 276, 408, 547, 729, 628 },
    { 2, 7, 26, 82, 172, 307, 392 },
    { 16, 24, 38, 74, 167, 511, 809 }
};
```

**Fase 3** Create un array con i nomi dei continenti

Per visualizzare le intestazioni delle righe della tabella serve un array monodimensionale con i nomi dei continenti; avrà tanti elementi quante sono le righe della tabella.

```
String[] continents =
{
    "Africa",
    "Asia",
    "Australia",
    "Europe",
    "North America",
    "South America"
};
```

Per visualizzare una riga della tabella dobbiamo per prima cosa visualizzare il nome del relativo continente, poi i valori di tutte le colonne: lo si fa con due cicli annidati. Il ciclo più esterno si occupa di visualizzare ciascuna riga:

```
// visualizza i dati relativi alla popolazione
for (int i = 0; i < ROWS; i++)
{
    // visualizza la riga i-esima
    ...
    System.out.println(); // la riga è finita, va a capo
}
```

Per visualizzare i dati contenuti in una riga della tabella partiamo dall'intestazione della riga (cioè il nome del continente), poi proseguiamo con i dati di tutte le colonne:

```
System.out.printf("%20s", continents[i]);
for (int j = 0; j < COLUMNS; j++)
{
    System.out.printf("%5d", populations[i][j]);
```

#### Fase 4 Calcolate il totale per ciascuna colonna

Per visualizzare le somme delle singole colonne usiamo l'algoritmo descritto nel Paragrafo 7.6.4, effettuando il calcolo una volta per ciascuna colonna.

```
for (int j = 0; j < COLUMNS; j++)
{
    int total = 0;
    for (int i = 0; i < ROWS; i++)
    {
        total = total + populations[i][j];
    }
    System.out.printf("%5d", total);
}
```

Ecco, infine, il programma completo.

#### File **WorldPopulation.java**

```
1 /**
2  * Visualizza una tabella con la popolazione mondiale negli ultimi 300 anni.
3 */
4 public class WorldPopulation
5 {
6     public static void main(String[] args)
7     {
8         final int ROWS = 6;
9         final int COLUMNS = 7;
10
11         int[][] populations =
12         {
13             { 106, 107, 111, 133, 221, 767, 1766 },
14             { 502, 635, 809, 947, 1402, 3634, 5268 },
15             { 2, 2, 2, 6, 13, 30, 46 },
```

```

16         { 163, 203, 276, 408, 547, 729, 628 },
17         { 2, 7, 26, 82, 172, 307, 392 },
18         { 16, 24, 38, 74, 167, 511, 809 }
19     };
20
21     String[] continents =
22     {
23         "Africa",
24         "Asia",
25         "Australia",
26         "Europe",
27         "North America",
28         "South America"
29     };
30
31     System.out.println("           Year 1750 1800 1850 1900 1950 2000 2050");
32
33     // visualizza i dati relativi alla popolazione
34
35     for (int i = 0; i < ROWS; i++)
36     {
37         // visualizza la riga i-esima
38         System.out.printf("%20s", continents[i]);
39         for (int j = 0; j < COLUMNS; j++)
40         {
41             System.out.printf("%5d", populations[i][j]);
42         }
43         System.out.println(); // la riga è finita, va a capo
44     }
45
46     // visualizza i totali sulle singole colonne
47
48     System.out.print("           World");
49     for (int j = 0; j < COLUMNS; j++)
50     {
51         int total = 0;
52         for (int i = 0; i < ROWS; i++)
53         {
54             total = total + populations[i][j];
55         }
56         System.out.printf("%5d", total);
57     }
58     System.out.println();
59 }
60 }
```

### Esecuzione del programma

|               | Year | 1750 | 1800 | 1850 | 1900 | 1950 | 2000 | 2050 |
|---------------|------|------|------|------|------|------|------|------|
| Africa        | 106  | 107  | 111  | 133  | 221  | 767  | 1766 |      |
| Asia          | 502  | 635  | 809  | 947  | 1402 | 3634 | 5268 |      |
| Australia     | 2    | 2    | 2    | 6    | 13   | 30   | 46   |      |
| Europe        | 163  | 203  | 276  | 408  | 547  | 729  | 628  |      |
| North America | 2    | 7    | 26   | 82   | 172  | 307  | 392  |      |
| South America | 16   | 24   | 38   | 74   | 167  | 511  | 809  |      |
| World         | 791  | 978  | 1262 | 1650 | 2522 | 5978 | 8909 |      |



## Argomenti avanzati 7.3

### Array bidimensionali con righe di lunghezze variabili

Quando si dichiara un array bidimensionale con l'enunciato

```
int[][] a = new int[3][3];
```

si ottiene una matrice 3 per 3 che può contenere 9 elementi:

|         |         |         |
|---------|---------|---------|
| a[0][0] | a[0][1] | a[0][2] |
| a[1][0] | a[1][1] | a[1][2] |
| a[2][0] | a[2][1] | a[2][2] |

In questa matrice tutte le righe hanno la stessa lunghezza, ma in Java è possibile creare anche array nei quali la lunghezza delle righe è variabile. Per esempio, si può creare un array che ha la forma di un triangolo, come questo:

|         |         |         |
|---------|---------|---------|
| b[0][0] |         |         |
| b[1][0] | b[1][1] |         |
| b[2][0] | b[2][1] | b[2][2] |

Per creare un array come questo bisogna faticare un po' di più. Per prima cosa si crea uno spazio adatto a contenere tre righe, lasciando vuoto il secondo indice dell'array per indicare che ciascuna riga verrà creata manualmente:

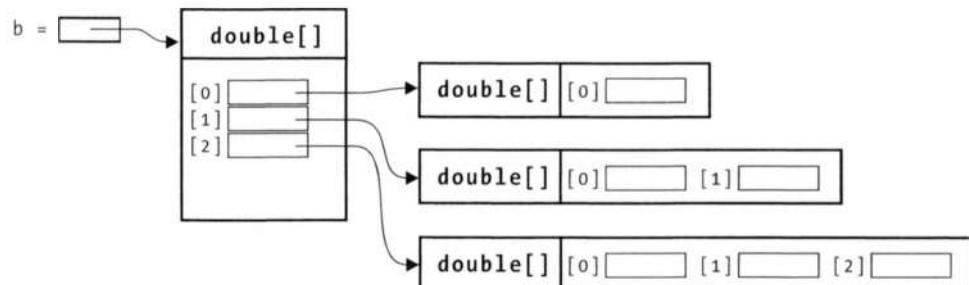
```
double[][] b = new double[3][];
```

Poi si crea ogni riga separatamente (Figura 16):

```
for (int i = 0; i < b.length; i++)
{
    b[i] = new double[i + 1];
}
```

Si può accedere a ciascun elemento dell'array usano la consueta notazione `b[i][j]`: l'espressione `b[i]` seleziona la riga di indice `i`, mentre l'operatore `[j]` seleziona l'elemento di indice `j` in tale riga.

**Figura 16**  
Un array triangolare



Osservate che il numero di righe è `b.length`, mentre la lunghezza della riga di indice `i` è `b[i].length`. Ad esempio, questa coppia di cicli visualizza un array “frastagliato”:

```
for (int i = 0; i < b.length; i++)
{
    for (int j = 0; j < b[i].length; j++)
    {
        System.out.print(b[i][j]);
    }
    System.out.println();
}
```

In alternativa si possono usare due cicli `for` estesi:

```
for (double[] row : b)
{
    for (double element : row)
    {
        System.out.print(element);
    }
    System.out.println();
}
```

Array come questi sono ovviamente piuttosto rari.

In realtà, Java realizza tutti gli array bidimensionali in questo modo, anche quelli “normali”: li tratta come array di array unidimensionali. L'espressione `new int[3][3]` crea automaticamente un array di tre righe e i tre array che memorizzeranno il contenuto delle tre righe.



## Argomenti avanzati 7.4

### Array multidimensionali

Si possono anche dichiarare array con più di due dimensioni. Ecco, ad esempio, un array tridimensionale:

```
int[][][] rubiksCube = new int[3][3][3];
```

Ciascun elemento dell'array viene specificato mediante il valore di tre indici:

```
rubiksCube[i][j][k]
```

## 7.7 Vettori

Un vettore memorizza una sequenza  
di valori e la lunghezza  
della sequenza è variabile.

Quando scrivete un programma che acquisisce dati in ingresso, non sempre sapete quanti dati arriveranno. In tale situazione, un **vettore** (*array list*) offre due significativi vantaggi:

- La dimensione di un vettore può aumentare o diminuire, in base alle necessità.
- La classe `ArrayList` fornisce metodi per svolgere le operazioni più comuni, come l'inserimento e la rimozione di elementi.

Nei paragrafi che seguono imparerete a utilizzare i vettori.

## Sintassi di Java

### 7.4 Vettori

|                 |                                                                                                                                                                                                                                                              |  |
|-----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| <b>Sintassi</b> | Per costruire un vettore:<br>new ArrayList<nomeTipo>()                                                                                                                                                                                                       |  |
|                 | Per accedere a un elemento:<br>riferimentoAVettore.get(indice)<br>riferimentoAVettore.set(indice, valore)                                                                                                                                                    |  |
| <b>Esempio</b>  | <p>Tipo della variabile      Nome della variabile      Un vettore di dimensione zero</p> <pre>ArrayList&lt;String&gt; friends = new ArrayList&lt;String&gt;();</pre> <p>friends.add("Cindy");   String name = friends.get(i);   friends.set(i, "Harry");</p> |  |

Per accedere a un elemento si usano i metodi `get` e `set`.

Il metodo `add` aggiunge un elemento in fondo al vettore, aumentandone la dimensione.

L'indice deve essere  $\geq 0$  e  $< \text{friends.size}()$ .

#### 7.7.1 Dichiarazione e utilizzo di vettori

L'enunciato seguente dichiara un vettore di stringhe:

```
ArrayList<String> names = new ArrayList<String>();
```

La classe `ArrayList` è una classe generica:

`ArrayList<NomeTipo>` contiene elementi di tipo `NomeTipo`.

La classe `ArrayList` si trova nel pacchetto `java.util`, quindi, per poter usare i vettori in un vostro programma, dovete aggiungere all'inizio l'enunciato `import java.util.ArrayList`.

Il tipo `ArrayList<String>` specifica un vettore con elementi di tipo `String`. Le parentesi angolari attorno al tipo `String` indicano che `String` è un **tipo (che funge da) parametro** o, come anche si dice, “parametro di tipo”: si può sostituire `String` con qualsiasi altra classe, ottenendo un vettore di tipo diverso, e per questo motivo `ArrayList` viene detta **classe generica (generic class)**. Ricordate, però, che i tipi primitivi non possono essere utilizzati come parametro di tipo: non esistono tipi di vettori come `ArrayList<int>` né `ArrayList<double>`. Il Paragrafo 7.7.4 illustrerà l'utilizzo di vettori di numeri.

Dimenticarsi di inizializzare il vettore è un errore decisamente frequente:

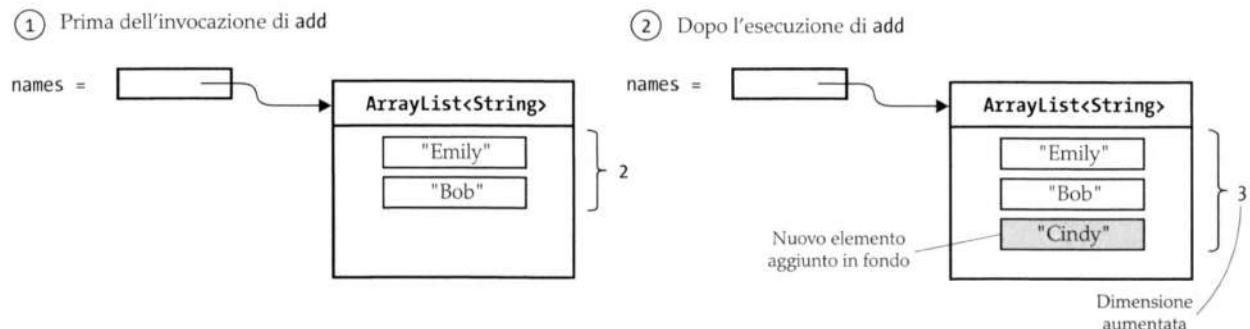
```
ArrayList<String> names;
names.add("Harry"); // ERRORE: la variabile names non è stata inizializzata
```

La variabile `names` va adeguatamente inizializzata, in questo modo:

```
ArrayList<String> names = new ArrayList<String>();
```

Si noti la coppia di parentesi tonde che si trova dopo `new ArrayList<String>`, nella parte destra dell'enunciato di dichiarazione con inizializzazione: segnala l'invocazione del **costruttore** della classe `ArrayList<String>`.

La dimensione iniziale di un vettore, dopo la sua costruzione, è zero: si può usare il metodo `add` per aggiungere un oggetto alla fine della sequenza di oggetti contenuti nel vettore, la cui dimensione aumenta di un'unità a ogni invocazione di `add` (come si vede nella Figura 17):



**Figura 17** Aggiunta di un elemento a un vettore mediante `add`

```
names.add("Emily"); // names ha lunghezza 1 e ha "Emily" come unico elemento
names.add("Bob"); // names ha lunghezza 2 e contiene: "Emily", "Bob"
names.add("Cindy"); // names ha lunghezza 3 e contiene: "Emily", "Bob", "Cindy"
```

Per conoscere la dimensione di un vettore si usa il metodo `size`.

Il metodo `size` restituisce la dimensione attuale del vettore.

Per ispezionare un elemento di un vettore si usa il metodo `get`, non l'operatore `[ ]`; come con gli array, i valori degli indici iniziano da 0. Ad esempio, `names.get(2)` restituisce l'elemento avente indice 2, cioè il terzo elemento del vettore:

```
String name = names.get(2);
```

Per accedere a un elemento di un vettore usando il suo indice si invocano i metodi `get` e `set`.

Accedere a un elemento non esistente è un errore, esattamente come accade negli array. L'errore di limiti più frequente è questo:

```
int i = names.size();
name = names.get(i); // Errore
```

L'indice valido di valore massimo è `names.size() - 1`.

Per assegnare un nuovo valore a un elemento di un vettore si usa il metodo `set`:

```
names.set(2, "Carolyn");
```

Questa invocazione assegna la stringa "Carolyn" alla posizione 2 del vettore `names`, sovrascrivendo qualunque valore memorizzato precedentemente in quella posizione.

Il metodo `set` può sovrascrivere solamente elementi che già esistono nel vettore ed è diverso dal metodo `add`, che aggiunge un nuovo elemento alla fine del vettore.

Per aggiungere o rimuovere elementi in un vettore si invocano i metodi `add` e `remove`.

È anche possibile inserire un elemento in una posizione intermedia all'interno di un vettore. L'invocazione `names.add(1, "Ann")`, ad esempio, sposta tutti gli elementi di una posizione in avanti, a partire dall'elemento attualmente in posizione 1 fino all'ultimo elemento presente nel vettore, poi aggiunge la stringa "Ann" nella posizione 1 (Figura 18).

Durante ogni invocazione del metodo `add` la dimensione del vettore viene aumentata di uno.

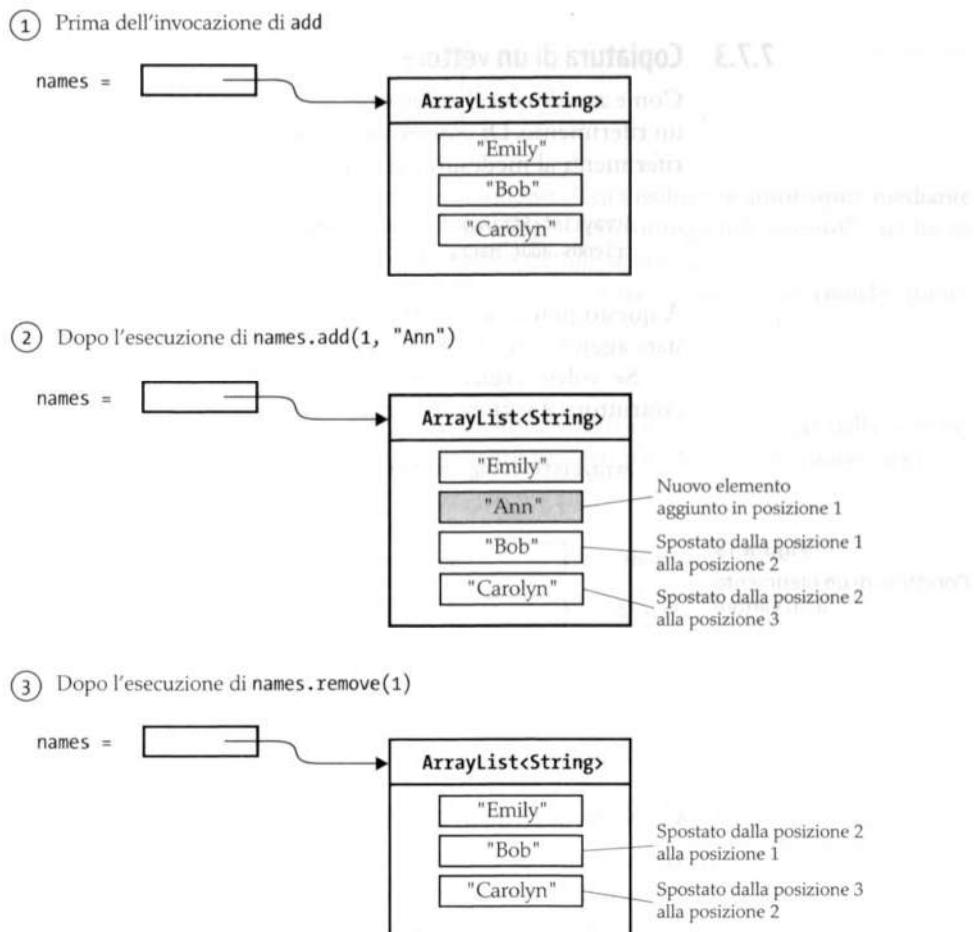
Al contrario, il metodo `remove` elimina l'elemento che si trova in una determinata posizione, poi sposta di una posizione all'indietro tutti gli elementi che si trovano dopo l'elemento rimosso e diminuisce di uno la dimensione del vettore, come si può vedere nell'ultima parte della Figura 18.

È molto semplice visualizzare rapidamente gli elementi contenuti in un vettore: basta fornirlo come parametro al metodo `println`.

```
System.out.println(names); // visualizza [Emily, Bob, Carolyn]
```

**Figura 18**

Aggiunta e rimozione  
di un elemento  
in una posizione  
intermedia di un vettore



### 7.7.2 Il ciclo for esteso usato con vettori

Per accedere ordinatamente a tutti gli elementi di un vettore si può anche usare il ciclo `for esteso`. Ad esempio, il ciclo seguente visualizza tutti i nomi:

```
ArrayList<String> names = . . . ;
for (String name : names)
{
    System.out.println(name);
}
```

Il ciclo precedente è equivalente al seguente ciclo `for` elementare:

```
for (int i = 0; i < names.size(); i++)
{
    String name = names.get(i);
    System.out.println(name);
}
```

### 7.7.3 Copiatura di un vettore

Come accade con gli array, è necessario ricordare che una variabile di tipo vettore contiene un riferimento. Di conseguenza, copiando un riferimento a un vettore si ottengono due riferimenti al medesimo vettore (come si può vedere nella Figura 19):

```
ArrayList<String> friends = names;
friends.add("Harry");
```

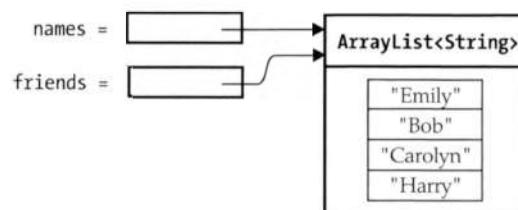
A questo punto `names` e `friends` fanno riferimento allo stesso vettore, in fondo al quale è stata aggiunta la stringa "Harry".

Se volete creare una copia di un vettore, costruitela passando come parametro al costruttore il vettore originale.

```
ArrayList<String> newNames = new ArrayList<String>(names);
```

**Figura 19**

Copiatura di un riferimento a un vettore



### 7.7.4 Classi involucro (*wrapper*) e *auto-boxing*

In Java non si possono inserire valori di un tipo primitivo (cioè numeri, caratteri e valori booleani) direttamente all'interno di vettori: non si può, ad esempio, creare un oggetto di tipo `ArrayList<double>`. Bisogna, invece, usare una delle **classi involucro** (*wrapper class*) elencate nella tabella riportata nella pagina seguente.

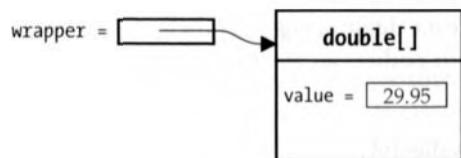
Per inserire valori di un tipo primitivo in un vettore si devono usare le classi involucro.

Ad esempio, per memorizzare valori di tipo `double` in un vettore, si usa un vettore di tipo `ArrayList<Double>`. Notate che i nomi delle classi involucro iniziano con lettere maiuscole e che due di essi differiscono dal nome del corrispondente tipo primitivo: `Integer` e `Character`.

| Tipo primitivo | Classe involucro |
|----------------|------------------|
| byte           | Byte             |
| boolean        | Boolean          |
| char           | Character        |
| double         | Double           |
| float          | Float            |
| int            | Integer          |
| long           | Long             |
| short          | Short            |

**Figura 20**

Una variabile che fa riferimento a un involucro



La conversione tra tipi primitivi e le corrispondenti classi involucro è automatica, mediante un processo che viene chiamato **auto-boxing** (“auto-impacchettamento”, anche se sarebbe stato più coerente l'utilizzo, in inglese, di *auto-wrapping*).

Ad esempio, se assegnate un valore di tipo `double` a una variabile di tipo `Double`, questo viene automaticamente “inserito in una scatola” (come nella Figura 20):

```
Double wrapper = 29.95;
```

Viceversa, i valori di tipo involucro vengono automaticamente “estratti dalla scatola” per generare valori di tipo primitivo, mediante un processo denominato *auto-unboxing*:

```
double x = wrapper;
```

**Tabella 2** Vettori al lavoro

```

ArrayList<String> names = new ArrayList<String>();
names.add("Ann"); names.add("Cindy");
System.out.print(names);
names.add(1, "Bob");

names.remove(0);

names.set(0, "Bill");

String name = names.get(i);
String last = names.get(names.size() - 1);

ArrayList<Integer> squares = new ArrayList<Integer>();
for (int i = 0; i < 10; i++)
{
    squares.add(i * i);
}
  
```

Costruisce un vettore vuoto, adatto a contenere stringhe.  
Aggiunge elementi alla fine del vettore.  
Visualizza [Ann, Cindy].  
Inserisce un elemento in posizione 1, dopodiché `names` contiene [Ann, Bob, Cindy].  
Elimina l'elemento in posizione 0, dopodiché `names` contiene [Bob, Cindy].  
Sostituisce un elemento, dopodiché `names` contiene [Bill, Cindy].  
Ispeziona un elemento.  
Ispeziona l'ultimo elemento.  
Costruisce un vettore che contiene i quadrati dei primi dieci numeri interi.

Dato che le procedure di *boxing* e *unboxing* sono automatiche, non serve pensarci. Per memorizzare numeri in un vettore, basta ricordarsi semplicemente di usare il tipo involucro quando si dichiara il vettore, affidandosi poi alle conversioni automatiche.

```
ArrayList<Double> values = new ArrayList<Double>();
values.add(29.95);
double x = values.get(0);
```

### 7.7.5 Algoritmi per array usati con vettori

Gli algoritmi che nel Paragrafo 7.3 sono stati applicati ad array possono essere convertiti per essere utilizzati con vettori, semplicemente usando i metodi dei vettori al posto della sintassi specificata degli array (la Tabella 3 riporta le conversioni necessarie). Questo frammento di codice, ad esempio, cerca l'elemento di valore massimo in un array:

```
double largest = values[0];
for (int i = 1; i < values.length; i++)
{
    if (values[i] > largest)
    {
        largest = values[i];
    }
}
```

e questo è il medesimo algoritmo, convertito per l'utilizzo di un vettore:

```
double largest = values.get(0);
for (int i = 1; i < values.size(); i++)
{
    if (values.get(i) > largest)
    {
        largest = values.get(i);
    }
}
```

**Tabella 3** Confronto tra operazioni con array e con vettori

| Operazione                                        | Array                                                        | Vettori                                          |
|---------------------------------------------------|--------------------------------------------------------------|--------------------------------------------------|
| Ispeziona un elemento                             | x = values[4];                                               | x = values.get(4);                               |
| Sostituisce un elemento                           | values[4] = 35;.                                             | values.set(4, 35);                               |
| Numero di elementi                                | values.length                                                | values.size()                                    |
| Numero di elementi in uso                         | currentSize (variabile ausiliaria, come nel Paragrafo 7.1.4) | values.size()                                    |
| Eliminazione di un elemento                       | Vedere Paragrafo 7.3.6                                       | values.remove(4);                                |
| Aggiunta di un elemento, aumentando la dimensione | Vedere Paragrafo 7.3.7                                       | values.add(35);                                  |
| Inizializzazione del contenuto                    | int[] values = { 1, 4, 9 };                                  | Operazione non prevista, si invoca più volte add |

### 7.7.6 Acquisizione di valori in un vettore

Durante l'acquisizione di un numero impreciso di valori in ingresso, l'utilizzo di un vettore è molto più semplice dell'uso di un array. È sufficiente acquisire i valori e, uno dopo l'altro, aggiungerli in fondo al vettore:

```
ArrayList<Double> inputs = new ArrayList<Double>();
while (in.hasNextDouble())
{
    inputs.add(in.nextDouble());
}
```

### 7.7.7 Eliminazione di specifici valori

L'eliminazione di elementi da un vettore si effettua in modo molto semplice, invocando il metodo `remove`. Capita abbastanza spesso di dover eliminare da un vettore tutti gli elementi che soddisfano una specifica condizione: supponiamo, ad esempio, di voler eliminare tutte le stringhe aventi lunghezza inferiore a 4.

Ovviamente si potrebbero analizzare uno dopo l'altro gli elementi del vettore, cercando quelli che soddisfano la condizione, in questo modo:

```
ArrayList<String> words = . . . ;
for (int i = 0; i < words.size(); i++)
{
    String word = words.get(i);
    if (word.length() < 4)
    {
        . . . // elimina l'elemento avente indice i
    }
}
```

ma c'è un problema subdolo: dopo aver eliminato un elemento, il ciclo `for` incrementa la variabile `i`, passando all'elemento *successivo* di quello da analizzare, cioè saltandone uno.

Prendiamo in esame questo esempio concreto, dove `words` contiene le stringhe "welcome", "to", "the" e "island". Quando `i` vale 1 eliminiamo la parola "to", che si trova nella posizione 1. Dopodiché `i` viene incrementata e vale 2, per cui la parola "the", che si trova ora nella posizione 1, non verrà mai esaminata.

Quando eliminiamo una parola, non dobbiamo incrementare l'indice. Lo pseudocodice corretto è, quindi, questo:

```
If l'elemento di indice i soddisfa la condizione
    Elimina l'elemento.
Else
    Incrementa i.
```

Dato che non sempre incrementiamo l'indice, in questo algoritmo il ciclo `for` non è adatto, è meglio usare un ciclo `while`:

```
int i = 0;
while(i < words.size())
```

```

    {
        String word = words.get(i);
        if (word.length() < 4)
        {
            words.remove(i);
        }
        else
        {
            i++;
        }
    }
}

```

### 7.7.8 Array o vettore?

Per la maggior parte dei programmati è più semplice utilizzare un vettore, perché i vettori aumentano e diminuiscono la propria dimensione in modo automatico. Di contro, gli array mettono a disposizione una sintassi molto più elegante e comoda per accedere agli elementi e per inizializzarli.

Quale contenitore è meglio utilizzare? Ecco alcuni consigli.

- Se la dimensione del contenitore non cambia mai, usate un array.
- Se dovete memorizzare una lunga sequenza di valori di tipo primitivo e dovete fare particolarmente attenzione all'efficienza, usate un array.
- In tutti gli altri casi usate un vettore.

Il programma presentato nel seguito mostra come si possa individuare e segnalare il valore massimo all'interno di una sequenza di valori memorizzata in un vettore. È una versione migliorata del programma analogo già presentato nel Paragrafo 7.3.10, che usava array: questo è in grado di elaborare sequenze di ingresso di lunghezza indeterminata.

#### File LargestInArrayList.java

```

1 import java.util.ArrayList;
2 import java.util.Scanner;
3
4 /**
5     Legge una sequenza di valori e li visualizza, segnalando il massimo.
6 */
7 public class LargestInArrayList
8 {
9     public static void main(String[] args)
10    {
11        ArrayList<Double> values = new ArrayList<Double>();
12
13        // legge i valori
14
15        System.out.println("Please enter values, Q to quit:");
16        Scanner in = new Scanner(System.in);
17        while (in.hasNextDouble())
18        {
19            values.add(in.nextDouble());
}

```

```

20 }
21
22 // trova il valore massimo
23
24 double largest = values.get(0);
25 for (int i = 1; i < values.size(); i++)
26 {
27     if (values.get(i) > largest)
28     {
29         largest = values.get(i);
30     }
31 }
32
33 // visualizza tutti i valori, segnalando il massimo
34
35 for (double element : values)
36 {
37     System.out.print(element);
38     if (element == largest)
39     {
40         System.out.print(" <= largest value");
41     }
42     System.out.println();
43 }
44 }
45 }

```

### Esecuzione del programma

```

Please enter values, Q to quit:
35 80 115 44.5 Q
35
80
115 <= largest value
44.5

```



### Auto-valutazione

35. Dichiarate un vettore di numeri interi, `primes`, che contenga i primi cinque numeri primi (2, 3, 5, 7 e 11).
36. Dato il vettore `primes` descritto nella domanda precedente, scrivete un ciclo che ne visualizzi gli elementi in ordine inverso, partendo dall'ultimo.
37. Cosa contiene `names` dopo l'esecuzione degli enunciati seguenti?  

```
ArrayList<String> names = new ArrayList<String>();
names.add("Bob");
names.add(0, "Ann");
names.remove(1);
names.add("Cal");
```
38. Cosa c'è di sbagliato in questo frammento di codice?  

```
ArrayList<String> names;
names.add("Bob");
```
39. Considerate questo metodo, che aggiunge gli elementi di un vettore (`source`) alla fine di un altro vettore (`target`).  

```
public void append(ArrayList<String> target, ArrayList<String> source)
```

```

{
    for (int i = 0; i < source.size(); i++)
    {
        target.add(source.get(i));
    }
}

```

Cosa contengono i vettori `names1` e `names2` dopo l'esecuzione degli enunciati seguenti?

```

ArrayList<String> names1 = new ArrayList<String>();
names1.add("Emily");
names1.add("Bob");
names1.add("Cindy");
ArrayList<String> names2 = new ArrayList<String>();
names2.add("Dave");
append(names1, names2);

```

40. Per memorizzare i nomi dei giorni della settimana è meglio usare un vettore o un array?
41. Nel pacchetto dei file scaricabili per questo libro, la cartella `section_7` del Capitolo 7 contiene il codice sorgente di un'implementazione alternativa della soluzione del problema visto nella sezione Consigli pratici 7.1. Confrontate le due implementazioni, una che usa array e l'altra che usa vettori: qual è il vantaggio principale della seconda?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R7.14, R7.33, E7.17 e E7.20, al termine del capitolo.



## Errori comuni 7.4

### Lunghezza e dimensione

Sfortunatamente la sintassi di Java per determinare il numero degli elementi contenuti in un array, in un vettore e in una stringa non è affatto coerente. Spesso si fa confusione: dovete assolutamente ricordarvi la sintassi corretta per ciascun tipo di dato.

| Tipo di dato | Numero di elementi      |
|--------------|-------------------------|
| Array        | <code>a.length</code>   |
| Vettore      | <code>a.size()</code>   |
| Stringa      | <code>a.length()</code> |



## Argomenti avanzati 7.5

### La sintassi a diamante (*diamond syntax*)

Per dichiarare vettori e altre classi generiche esiste anche una sintassi alternativa, decisamente comoda. Quando si dichiara e si costruisce un vettore, non occorre ripetere nel costruttore il tipo usato come parametro nella dichiarazione della variabile, per cui si può scrivere

```
ArrayList<String> names = new ArrayList<>();
```

invece di

```
ArrayList<String> names = new ArrayList<String>();
```

Si parla di questa scorciatoia come di “sintassi a diamante” (*diamond syntax*), perché le due parentesi angolari vuote hanno una forma che ricorda vagamente quella di un diamante, <>.

Finora abbiamo sempre utilizzato la sintassi esplicita, ma nei capitoli successivi useremo la sintassi a diamante.

## 7.8 Collaudo regressivo

Un pacchetto di prove (*test suite*) è un insieme di prove per il collaudo ripetuto.

Di fatto, creare un nuovo caso di prova ogni volta che si riscontra un errore nel programma (*bug*) è una pratica utile e diffusa, usando poi tale prova per verificare che la correzione dell’errore funzioni veramente. Non eliminate il nuovo caso di prova e usatelo per collaudare la versione successiva del programma e tutte le versioni seguenti. Una simile raccolta di casi di prova è detta **test suite** (*pacchetto di prove*).

Vi stupirete della frequenza con cui un errore, già corretto, riappare in una versione successiva: si tratta di un fenomeno chiamato *ciclicità*. Vi capiterà di non comprendere a fondo il motivo di un errore e inserirete una rapida correzione che sembrerà funzionare: più tardi applicherete un’altra rapida correzione per risolvere un secondo problema e questo farà riemergere il primo errore. Naturalmente è sempre meglio ragionare a fondo sulle cause di un errore e risolverlo alla radice, anziché produrre una serie di soluzioni “d’emergenza”. Tuttavia, se non riuscite ad afferrare il problema, conviene almeno poter contare su una schietta valutazione del funzionamento del programma: se si conservano a portata di mano tutti i vecchi casi di prova, e se ciascuna nuova versione viene collaudata con tutti tali casi, si avrà questa risposta. L’operazione di verifica dell’avvenuta correzione di una serie di errori rilevati in passato si chiama **collaudo regressivo** (*regression testing*).

Il collaudo regressivo prevede l’esecuzione ripetuta di prove già eseguite in precedenza, per essere certi che problemi risolti in passato non compaiano nelle nuove versioni del programma.

```
public class ScoreTester1
{
    public static void main(String[] args)
    {
        Student fred = new Student(100);
        fred.addScore(10);
        fred.addScore(20);
        fred.addScore(5);
        System.out.println("Final score: " + fred.finalScore());
        System.out.println("Expected: 30"); // valore previsto
    }
}
```

Un altro approccio efficace prevede di realizzare una classe di collaudo generale e di fornirle dati in ingresso prelevati da file diversi, come in questo esempio:

### File ScoreTester.java

```

1 import java.util.Scanner;
2
3 public class ScoreTester
4 {
5     public static void main(String[] args)
6     {
7         Scanner in = new Scanner(System.in);
8         double expected = in.nextDouble();
9         Student fred = new Student(100);
10        while (in.hasNextDouble())
11        {
12            if (!fred.addScore(in.nextDouble()))
13            {
14                System.out.println("Too many scores.");
15                return;
16            }
17        }
18        System.out.println("Final score: " + fred.finalScore());
19        System.out.println("Expected: " + expected); // valore previsto
20    }
21 }
```

Il programma legge il valore previsto, che dovrebbe essere calcolato dalla classe `Student`, e le valutazioni da elaborare. Eseguendo il programma con diversi insiemi di valori di ingresso si possono collaudare situazioni diverse.

Sarebbe ovviamente molto noioso digitare a mano i valori di ingresso ogni volta che si intende eseguire un collaudo. È molto meglio memorizzare tali valori in un file come questo:

### File input1.txt

```

30
10
20
5
```

Quando si esegue il programma all'interno di una finestra di console si può collegare il flusso di ingresso del programma al file contenente i dati da elaborare, come se i caratteri del file venissero effettivamente digitati sulla tastiera da un utente. Scrivete questo comando in una finestra di console:

```
java ScoreTester < input1.txt
```

Il programma viene eseguito, ma non legge dati dalla tastiera: l'oggetto `System.in` (e l'oggetto di tipo `Scanner` che legge da `System.in`) preleva i dati dal file `input1.txt`. Questo processo viene chiamato “redirezione di input” e ne abbiamo parlato nella sezione Argomenti avanzati 6.2.

Nella finestra viene quindi visualizzato quanto segue:

## Esecuzione del programma

```
Final score: 30
Expected: 30
```

È possibile usare anche la redirezione di output. Per memorizzare in un file ciò che viene visualizzato dal programma, usate questo comando:

```
java ScoreTester < input1.txt > output1.txt
```

Questo modo di operare è utile per archiviare i risultati dell'esecuzione di un pacchetto di prove.



## Auto-valutazione

42. Ipotizzate di aver modificato il codice di un metodo: perché dovreste voler ripetere i collaudi che erano già stati superati con successo dalla precedente versione del codice?
43. Ipotizzate che un acquirente di un vostro programma scopra un errore: cosa dovreste fare prima di eliminare l'errore?
44. Perché il programma ScoreTester non visualizza messaggi che chiedano di inserire dati?

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi R7.35 e R7.36, al termine del capitolo.



## Suggerimenti per la programmazione 7.3

### File batch e script di shell

Se dovete eseguire ripetutamente le stesse attività sulla riga dei comandi, vale la pena di imparare le caratteristiche di automazione offerte dal vostro sistema operativo.

In ambiente Windows potete usare *file batch* per eseguire automaticamente una sequenza di comandi. Per esempio, immaginate di dover collaudare un programma mediante l'esecuzione di tre classi di collaudo:

```
java ScoreTester1
java ScoreTester < input1.txt
java ScoreTester < input2.txt
```

In seguito scoprirete un errore, lo correggete e dovete eseguire nuovamente le medesime prove, con la necessità di digitare ancora una volta i tre comandi: è ragionevole che debba esistere un sistema migliore per farlo. In ambiente Windows, inserite i comandi in un file di testo, che chiamerete, ad esempio, *test.bat*:

### File test.bat

```
java ScoreTester1
java ScoreTester < input1.txt
java ScoreTester < input2.txt
```

Quindi, per eseguire automaticamente i tre comandi contenuti in tale *file batch*, è sufficiente digitare un solo comando:

```
test.bat
```

I file batch sono una caratteristica del sistema operativo, non di Java. Nei sistemi Linux, Mac OS e UNIX, per lo stesso scopo si usano gli *script di shell*. Nel caso di questo semplice esempio, potete eseguire i medesimi comandi scrivendo

```
sh test.bat
```

Ci sono molti interessanti utilizzi per i file batch e gli script di shell ed è bene imparare anche alcune loro caratteristiche avanzate, come i parametri e i cicli.



## Computer e società 7.2

### L'incidente del Therac-25

Il Therac-25 è un dispositivo computerizzato per la terapia a emissione di radiazioni, destinata ai malati di tumore. Fra il mese di giugno 1985 e il mese di gennaio 1987, alcune di queste macchine rilasciarono dosi eccessive di radiazioni ad almeno sei pazienti, uccidendone alcuni e menomando seriamente gli altri.

Le macchine erano controllate da un programma informatico, i cui errori furono direttamente responsabili delle dosi eccessive. Secondo Leveson e Turner ("An Investigation of the Therac-25 Accidents", *IEEE Computer*, July 1993, 18-41), il programma fu scritto da un unico programmatore, che in seguito lasciò la società costruttrice che produceva l'apparecchio e non fu più rintracciabile. Nessuno, fra i dipendenti della società che furono interrogati, fu in grado di dire alcunché circa il livello di studi o le qualifiche di quel programmatore.

L'indagine dell'agenzia federale FDA (Food and Drug Administration) rilevò che il programma era scarsamente documentato e che non esisteva né un elenco di specifiche, né

un piano formale di collaudo (questo dovrebbe farvi riflettere: avete un piano formale per il collaudo dei vostri programmi?).

I sovradosaggi erano da imputare alla progettazione dilettantistica del software, che doveva controllare simultaneamente diversi dispositivi: la tastiera, lo schermo, la stampante e, naturalmente, l'apparecchio per le radiazioni. La sincronizzazione e la condivisione dei dati fra le diverse attività erano realizzate mediante una soluzione *ad hoc*, nonostante all'epoca fossero già conosciute tecniche sicure per il *multitasking*. Se il programmatore avesse beneficiato di studi regolari in merito a queste tecniche o se si fosse preso il disturbo di studiare la letteratura sull'argomento, avrebbe potuto costruire una macchina più sicura, anche se, probabilmente, ciò avrebbe comportato l'adozione di un sistema multitasking, scelto fra quelli in commercio, che forse avrebbe richiesto un computer più costoso.

Gli stessi difetti erano presenti nel software che controllava il modello precedente, il Therac-20, ma

quella macchina conteneva blocchi hardware che impedivano meccanicamente l'eccesso di radiazioni. Nel Therac-25 i dispositivi di sicurezza hardware vennero eliminati per sostituirli con controlli software, presumibilmente per risparmiare.

Frank Houston, della FDA, nel 1985 scrisse: "Una quantità significativa di software, nei sistemi critici per la salute, proviene da piccole aziende, specialmente nell'industria delle apparecchiature mediche. Si tratta di aziende che rientrano nel profilo di quelle refrattarie o disinformatate in merito ai principi della sicurezza dei sistemi e della progettazione del software."

È difficile individuare il colpevole: il programmatore? Il dirigente, che non solo non si è assicurato che il programmatore fosse all'altezza del compito, ma che pure non ha preteso un collaudo completo? Gli ospedali che hanno installato l'apparecchiatura, o la FDA, che non ha controllato il processo di progettazione? Ancora oggi, sfortunatamente, non esistono standard industriali che definiscono un processo sicuro per la progettazione del software.

## Riepilogo degli obiettivi di apprendimento

### Array per memorizzare sequenze di valori

- Un array contiene una sequenza di valori del medesimo tipo, cioè omogenei.
- Si accede a un singolo elemento di un array mediante un indice intero *i*, usando la notazione *array[i]*.
- Un elemento di un array può essere usato esattamente come una singola variabile dello stesso tipo.
- L'indice in un array deve essere almeno zero e minore della lunghezza dell'array.
- Un errore di limiti, che si verifica se si usa un indice non valido in un array, può far terminare il programma.
- Per conoscere il numero di elementi presenti in un array si usa l'espressione *array.length*.
- Un riferimento a un array specifica la posizione dell'array all'interno della memoria. Copiando un riferimento si ottiene un secondo riferimento allo stesso array.
- Gli array possono essere argomenti di metodi o valori restituiti da metodi.
- Insieme a un array riempito parzialmente, usate una variabile ausiliaria che tenga traccia del numero di elementi realmente utilizzati.
- Evitate di usare array paralleli: trasformateli in array di oggetti.

### Il ciclo for esteso

- Il ciclo for esteso scandisce tutti gli elementi di un array.
- Si può usare un ciclo for esteso soltanto se nel corpo del ciclo non c'è bisogno del valore dell'indice.

### Gli algoritmi più comuni per l'elaborazione di array

- Quando si separano elementi, non bisogna inserire un separatore prima del primo elemento.
- Una ricerca lineare ispeziona gli elementi in sequenza finché non trova quello cercato.
- Prima di inserire un elemento, bisogna spostare gli elementi verso la fine dell'array, *a partire dall'ultimo*.
- Per scambiare tra loro due elementi di un array bisogna usare una variabile temporanea.
- Per copiare gli elementi di un array, usate il metodo `Arrays.copyOf`.

### Algoritmi fondamentali e problemi complessi

- Combinando algoritmi fondamentali si possono risolvere problemi di programmazione complessi.
- Bisogna conoscere bene gli algoritmi fondamentali, per poterli adattare a situazioni simili.

### Progettazione di algoritmi e manipolazione concreta di oggetti

- Per visualizzare un array di valori usate una sequenza di monete, di carte da gioco o di piccoli giocattoli.
- Per tenere traccia degli spostamenti delle variabili che indicano le posizioni nell'array si possono usare graffette o altri contrassegni.

### Per dati organizzati in righe e colonne si usano array bidimensionali

- Per memorizzare i dati di una tabella si usa un array bidimensionale.
- Ai singoli elementi di un array bidimensionale si accede usando due indici: *array[i][j]*.

### Per gestire sequenze di dimensione variabile si usano vettori

- Un vettore memorizza una sequenza di valori e la lunghezza della sequenza è variabile.
- La classe `ArrayList` è una classe generica: `ArrayList<NomeTipo>` contiene elementi di tipo `NomeTipo`.
- Per conoscere la dimensione di un vettore si usa il metodo `size`.
- Per accedere a un elemento di un vettore usando il suo indice si invocano i metodi `get` e `set`.
- Per aggiungere o rimuovere elementi in un vettore si invocano i metodi `add` e `remove`.
- Per inserire valori di un tipo primitivo in un vettore si devono usare le classi involucro.

### Il collaudo regressivo

- Un pacchetto di prove (*test suite*) è un insieme di prove per il collaudo ripetuto.
- Il collaudo regressivo prevede l'esecuzione ripetuta di prove già eseguite in precedenza, per essere certi che problemi risolti in passato non compaiano nelle nuove versioni del programma.

## Elementi di libreria presentati in questo capitolo

`java.lang.Boolean`  
`java.lang.Double`  
`java.lang.Integer`  
`java.util.Arrays`  
`copyOf`  
`toString`

`java.util.ArrayList<E>`  
`add`  
`get`  
`remove`  
`set`  
`size`

## Esercizi di riepilogo e approfondimento

- ★★ **R7.1.** Usando un unico array, risolvete i seguenti problemi, seguendo l'ordine in cui sono proposti:
- Dichiarate un array predisposto per contenere dieci numeri interi.
  - Inserite il numero 17 come elemento iniziale dell'array.
  - Inserite il numero 29 come elemento finale dell'array.
  - Assegnate -1 a tutti gli altri elementi.
  - Aggiungete 1 a tutti gli elementi dell'array.
  - Visualizzate tutti gli elementi dell'array, uno per riga.
  - Visualizzate tutti gli elementi dell'array in un'unica riga, separati da virgolette.
- \* **R7.2.** In un array, cos'è un indice? Quali sono i valori validi per un indice in un array? Cos'è un errore di limiti?
- \* **R7.3.** Scrivete un programma che contiene un errore di limiti ed eseguitelo. Cosa accade al vostro computer?
- \* **R7.4.** Scrivete un ciclo che legge dieci numeri e un secondo ciclo che li visualizza in ordine inverso rispetto a come sono stati acquisiti.
- ★★ **R7.5.** Scrivete un frammento di codice che riempia l'array `values` con ciascuno dei seguenti insiemi di numeri:
- 1 2 3 4 5 6 7 8 9 10
  - 0 2 4 6 8 10 12 14 16 18 20
  - 1 4 9 16 25 36 49 64 81 100
  - 0 0 0 0 0 0 0 0 0

- e. 1 4 9 16 9 7 4 9 11
- f. 0 1 0 1 0 1 0 1 0 1
- g. 0 1 2 3 4 0 1 2 3 4

★★ **R7.6.** Dato il seguente array:

```
int[] a = { 1, 2, 3, 4, 5, 4, 3, 2, 1, 0 };
```

qual è il valore di total al termine dell'esecuzione di ciascuno dei cicli seguenti?

- a. int total = 0;  
for (int i = 0; i < 10; i++) { total = total + a[i]; }
- b. int total = 0;  
for (int i = 0; i < 10; i = i + 2) { total = total + a[i]; }
- c. int total = 0;  
for (int i = 1; i < 10; i = i + 2) { total = total + a[i]; }
- d. int total = 0;  
for (int i = 2; i <= 10; i++) { total = total + a[i]; }
- e. int total = 0;  
for (int i = 1; i < 10; i = 2 \* i) { total = total + a[i]; }
- f. int total = 0;  
for (int i = 9; i >= 0; i--) { total = total + a[i]; }
- g. int total = 0;  
for (int i = 9; i >= 0; i = i - 2) { total = total + a[i]; }
- h. int total = 0;  
for (int i = 0; i < 10; i++) { total = total - a[i]; }

★★ **R7.7.** Dato il seguente array:

```
int[] a = { 1, 2, 3, 4, 5, 4, 3, 2, 1, 0 };
```

qual è il suo contenuto al termine dell'esecuzione di ciascuno dei cicli seguenti?

- a. for (int i = 1; i < 10; i++) { a[i] = a[i - 1]; }
- b. for (int i = 9; i > 0; i--) { a[i] = a[i - 1]; }
- c. for (int i = 0; i < 9; i++) { a[i] = a[i + 1]; }
- d. for (int i = 8; i >= 0; i--) { a[i] = a[i + 1]; }
- e. for (int i = 1; i < 10; i++) { a[i] = a[i] + a[i - 1]; }
- f. for (int i = 1; i < 10; i = i + 2) { a[i] = 0; }
- g. for (int i = 0; i < 5; i++) { a[i + 5] = a[i]; }
- h. for (int i = 1; i < 5; i++) { a[i] = a[9 - i]; }

★★★ **R7.8.** Scrivete un ciclo che riempia l'array `values` con dieci numeri casuali compresi tra 1 e 100. Poi, scrivete il codice per due cicli annidati che riempiano l'array `values` con dieci numeri casuali compresi tra 1 e 100 ma *tutti diversi tra loro* (cioè, al termine, l'array non deve contenere numeri replicati).

★★ **R7.9.** Scrivete un ciclo che calcoli contemporaneamente il valore massimo e il valore minimo in un array di numeri.

\* **R7.10.** Cosa c'è di sbagliato in questi due frammenti di codice?

- a. int[] values = new int[10];  
for (int i = 1; i <= 10; i++)

```

    {
        values[i] = i * i;
    }
b. int[] values;
for (int i = 0; i < values.length; i++)
{
    values[i] = i * i;
}

```

**\*\* R7.11.** Risolvete i problemi seguenti scrivendo cicli `for` estesi.

- Visualizzare tutti gli elementi di un array in un'unica riga, separati da uno spazio.
- Calcolare il valore massimo tra tutti gli elementi dell'array.
- Contare quanti elementi dell'array sono negativi.

**\*\* R7.12.** Riscrivete i cicli seguenti senza usare il `for` esteso, tenendo presente che `values` è di tipo `double[]`.

- `for (double x : values) { total = total + x; }`
- `for (double x : values) { if (x == target) { return true; } }`
- `int i = 0;`  
`for (double x : values) { values[i] = 2 * x; i++; }`

**\*\* R7.13.** Riscrivete i cicli seguenti usando il `for` esteso, tenendo presente che `values` è di tipo `double[]`.

- `for (int i = 0; i < values.length; i++) { total = total + values[i]; }`
- `for (int i = 1; i < values.length; i++) { total = total + values[i]; }`
- `for (int i = 0; i < values.length; i++)`  
`{`  
 `if (values[i] == target) { return i; }`  
`}`

**\*** **R7.14.** Cosa c'è di sbagliato nei seguenti frammenti di codice?

- `ArrayList<int> values = new ArrayList<int>();`
- `ArrayList<Integer> values = new ArrayList();`
- `ArrayList<Integer> values = new ArrayList<Integer>;`
- `ArrayList<Integer> values = new ArrayList<Integer>();`  
`for (int i = 1; i <= 10; i++)`  
`{`  
 `values.set(i - 1, i * i);`  
`}`
- `ArrayList<Integer> values;`  
`for (int i = 1; i <= 10; i++)`  
`{`  
 `values.add(i * i);`  
`}`

**\*\* R7.15.** Per le operazioni seguenti, che agiscono su array riempiti solo in parte, scrivete le intestazioni dei metodi, senza implementarli:

- a. Disporre gli elementi in ordine decrescente.
  - b. Visualizzare tutti gli elementi, separati da una stringa data.
  - c. Contare quanti elementi sono minori di un valore dato.
  - d. Eliminare tutti gli elementi che sono minori di un valore dato.
  - e. Copiare in un altro array tutti gli elementi che sono minori di un valore dato.
- \* **R7.16.** Eseguite a mano, passo dopo passo, il ciclo visto nel Paragrafo 7.3.4, usando i dati là indicati. Tenete traccia su carta dell'esecuzione, mostrando in due colonne il valore di *i* e quanto viene visualizzato.
- \* **R7.17.** Considerate il ciclo seguente, che raccoglie tutti gli elementi che soddisfano una determinata condizione: in questo caso, che l'elemento sia maggiore di 100.

```
ArrayList<Double> matches = new ArrayList<Double>();
for (double element : values)
{
    if (element > 100)
    {
        matches.add(element);
    }
}
```

Sapendo che *values* contiene gli elementi 110, 90, 100, 120 e 80, eseguite il ciclo a mano, passo dopo passo, tenendo traccia su carta dell'esecuzione, mostrando in due colonne il valore di *element* e il contenuto di *matches*.

- \* **R7.18.** Eseguite a mano, passo dopo passo, il ciclo visto nel Paragrafo 7.3.5, nell'ipotesi che *values* contenga gli elementi 80, 90, 100, 120 e 110. Tenete traccia su carta dell'esecuzione, mostrando in due colonne il valore di *pos* e il valore di *found*. Ripetete l'esercizio nel caso in cui *values* contenga gli elementi 80, 90, 120 e 70.
- \*\* **R7.19.** Eseguite a mano, passo dopo passo, l'algoritmo di eliminazione di un elemento visto nel Paragrafo 7.3.6, nell'ipotesi che *values* contenga gli elementi 110, 90, 100, 120 e 80, eliminando l'elemento corrispondente all'indice 2.
- \*\* **R7.20.** Progettate mediante pseudocodice un algoritmo che ruoti di una posizione gli elementi di un array, spostando alla fine dell'array il suo elemento iniziale, come nella figura.



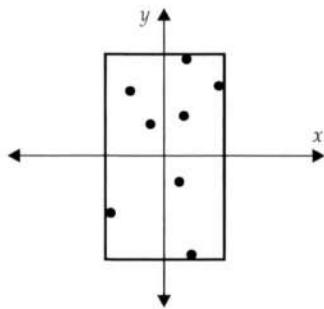
- \*\* **R7.21.** Progettate mediante pseudocodice un algoritmo che elimini da un array tutti i valori negativi, preservando l'ordine relativo tra gli elementi rimanenti.
- \*\* **R7.22.** Nell'ipotesi che *values* sia un array *ordinato* di numeri interi, progettate mediante pseudocodice un algoritmo che vi inserisca un nuovo elemento in modo che, al termine, l'array sia ancora ordinato.
- \*\*\* **R7.23.** In un array, chiamiamo "serie" (*run*) una sequenza di valori adiacenti ripetuti. Progettate mediante pseudocodice un algoritmo che calcoli la lunghezza della più lunga serie presente in un array. Ad esempio, in questo array la serie più lunga ha lunghezza 4:

1 2 5 5 3 1 2 4 3 2 2 2 3 6 5 5 6 3 1

- \*\*\* R7.24. Cosa c'è di sbagliato nel metodo seguente, che vorrebbe riempire di numeri casuali un array?

```
public void makeCombination(int[] values, int n)
{
    Random generator = new Random();
    int[] numbers = new int[values.length];
    for (int i = 0; i < numbers.length; i++)
    {
        numbers[i] = generator.nextInt(n);
    }
    values = numbers;
}
```

- \*\* R7.25. Sono dati due array che riportano, rispettivamente, le coordinate  $x$  e  $y$  di un insieme di punti nel piano cartesiano. Per disegnarli, dobbiamo conoscere le coordinate  $x$  e  $y$  dei vertici del più piccolo rettangolo parallelo agli assi cartesiani che contenga tutti i punti. Come si possono calcolare queste informazioni usando gli algoritmi elementari visti nel Paragrafo 7.3?



- \* R7.26. Risolvete il problema della valutazione dei questionari posto nel Paragrafo 7.4 ordinando prima l'array. Come va modificato l'algoritmo che calcola il punteggio totale?
  - \*\* R7.27. Risolvete il problema descritto nel Paragrafo 7.5 usando un algoritmo che elimini e inserisca gli elementi invece di scambiarli. Descrivete l'algoritmo mediante pseudocodice, nell'ipotesi che esistano metodi per la rimozione e per l'inserimento. Provate a eseguire concretamente l'algoritmo usando delle monete e spiegate perché sia meno efficiente di quello che effettua gli scambi, sviluppato nel Paragrafo 7.5.
  - \*\* R7.28. Progettate un algoritmo che trovi, in un array di numeri, il valore che ricorre più frequentemente. Disponete monete in sequenza e usate delle graffette: vicino a ogni moneta mettete tante graffette quante sono le altre monete uguali presenti nella sequenza. Descrivete l'algoritmo mediante pseudocodice e spiegate in che modo l'utilizzo di monete e graffette vi sia stato utile nel progettarlo.
  - \*\* R7.29. Scrivete enunciati che, in Java, risolvano i problemi indicati, usando questo array:
- ```
int[][] values = new int[ROWS][COLUMNS];
```
- Assegnare 0 a tutti gli elementi.
  - Assegnare alternativamente i valori 0 e 1 agli elementi, in modo da creare una scacchiera.
  - Assegnare 0 soltanto agli elementi che si trovano nella prima e nell'ultima riga.
  - Calcolare la somma di tutti gli elementi.
  - Visualizzare tutti gli elementi in formato tabulare.

- \*\* **R7.30.** Progettate mediante pseudocodice un algoritmo che assegni il valore  $-1$  a tutti gli elementi che si trovano nella prima e nell'ultima colonna e nella prima e ultima riga di un array bidimensionale di numeri interi.
- \* **R7.31.** Nel Paragrafo 7.7.7 avete visto quanta attenzione bisogna porre nell'aggiornamento del valore dell'indice quando si rimuovono elementi da un vettore. Mostrate come si possa evitare questo problema scandendo gli elementi del vettore a ritroso.
- \*\* **R7.32.** Vero o falso?
- Tutti gli elementi di un array sono dello stesso tipo.
  - Gli array non possono avere stringhe come elementi.
  - Gli array bidimensionali hanno sempre lo stesso numero di righe e di colonne.
  - In un array bidimensionale gli elementi di colonne diverse possono avere tipi diversi.
  - Un metodo non può restituire un array bidimensionale.
  - Un metodo non può modificare la lunghezza di un array ricevuto come parametro.
  - Un metodo non può modificare il numero di colonne di un array bidimensionale ricevuto come parametro.
- \*\* **R7.33.** Come si risolvono i problemi seguenti?
- Verificare se due array contengono gli stessi elementi, nello stesso ordine.
  - Copiare il contenuto di un array in un altro.
  - Riempire di zeri un array, sovrascrivendo tutti i suoi elementi.
  - Eliminare da un array tutti i suoi elementi.
- \* **R7.34.** Vero o falso?
- Tutti gli elementi di un vettore sono dello stesso tipo.
  - Gli indici usati in un vettore devono essere numeri interi.
  - I vettori non possono avere stringhe come elementi.
  - I vettori possono modificare la propria dimensione, aumentandola o diminuendola.
  - Un metodo non può restituire un vettore.
  - Un metodo non può modificare la lunghezza di un vettore ricevuto come parametro.
- \* **R7.35 (collaudo).** Definite i termini *collaudo regressivo* (“regression testing”) e *pacchetto di prove* (“test suite”).
- \*\* **R7.36 (collaudo).** Nel collaudo, quale fenomeno viene detto *ciclicità*? Cosa si può fare per evitarlo?

## Esercizi di programmazione

- \*\* **E7.1.** Scrivete un programma che inizializzi un array con dieci numeri interi casuali e, poi, visualizzi quattro righe di informazioni, contenenti:
- Tutti gli elementi di indice pari.
  - Tutti gli elementi di valore pari.
  - Tutti gli elementi in ordine inverso.
  - Soltanto il primo e l'ultimo elemento.
- \* **E7.2.** Modificate il programma `LargestInArray.java`, visto nel Paragrafo 7.3, in modo che evidenzi l'elemento minimo e l'elemento massimo.

- \*\* E7.3. Scrivete il metodo `sumWithoutSmallest` che calcoli, usando un unico ciclo, la somma di tutti i valori di un array, escludendo il valore minimo. Nel ciclo, aggiornate la somma e il valore minimo; al termine, restituire la differenza tra questi due valori.
- \* E7.4. Aggiungete alla classe `Student`, vista nel Paragrafo 7.4, il metodo `removeMin` che elimini il punteggio minimo senza invocare altri metodi
- \*\* E7.5. Scrivete un programma che legga una sequenza di numeri interi, memorizzandola in un array, e ne calcoli la *somma a elementi alterni*. Per esempio, se il programma viene eseguito fornendo questi dati

1 4 9 16 9 7 4 9 11

allora calcola

$$1 - 4 + 9 - 16 + 9 - 7 + 4 - 9 + 11 = -2$$

- \* E7.6. Scrivete un programma che legga una sequenza di numeri interi, memorizzandola in un array, e ne inverta l'ordine. Per esempio, se il programma viene eseguito fornendo questi dati

1 4 9 16 9 7 4 9 11

allora il contenuto dell'array deve diventare il seguente:

11 9 4 7 9 16 9 4 1

- \*\*\* E7.7. Scrivete un programma che produca 10 permutazioni casuali dei numeri da 1 a 10. Per generare una permutazione casuale, riempite un array con i numeri da 1 a 10, facendo in modo che non ve ne siano due uguali. Potreste farlo in modo brutale, generando numeri casuali fino a quando non viene prodotto un valore non ancora presente nell'array, ma si tratta di una soluzione inefficiente. Seguite, invece, questo algoritmo:

Ripeti 10 volte

Crea un secondo array e riempilo con i numeri da 1 a 10.

Scegli a caso un elemento dal secondo array.

Eliminalo dal secondo array e aggiungilo alla permutazione attuale.

- \* E7.8. Scrivete un metodo che implementi l'algoritmo sviluppato nel Paragrafo 7.5.
- \*\* E7.9. Progettate la classe `DataSet` che memorizzi una sequenza di valori di tipo `double`, con un costruttore

```
public DataSet(int maximumNumberOfValues)
```

e un metodo

```
public void add(double value)
```

che aggiunge un valore alla sequenza, se c'è ancora posto.

Aggiungete altri metodi per calcolare la somma, il valore medio, il valore massimo e il valore minimo della sequenza.

- \*\* E7.10. Scrivete metodi che risolvano i problemi seguenti per un array di numeri interi, fornendo un programma di collaudo per ciascun metodo. I metodi vanno inseriti a completamento della classe seguente:

```

public class ArrayMethods
{
    private int[] values;
    public ArrayMethods(int[] initialValues) { values = initialValues; }
    public void swapFirstAndLast() { . . . } // risolve il problema a.
    public void shiftRight() { . . . } // risolve il problema b.
    . . .
}

```

- a. Scambiare tra loro il primo e l'ultimo elemento dell'array.
- b. Far scorrere tutti gli elementi di una posizione “verso destra”, spostando l'ultimo elemento nella prima posizione. Ad esempio, l'array 1 4 9 16 25 deve diventare 25 1 4 9 16.
- c. Sostituire con 0 tutti gli elementi pari.
- d. Sostituire ciascun elemento, tranne il primo e l'ultimo, con il più grande dei due elementi adiacenti.
- e. Eliminare l'elemento centrale dell'array se questo ha dimensione dispari, altrimenti eliminare i due elementi centrali.
- f. Spostare tutti gli elementi pari all'inizio dell'array, preservando però l'ordinamento relativo tra gli elementi, se si esclude la condizione imposta.
- g. Restituire il secondo valore maggiore dell'array, cioè il valore massimo tra quelli inferiori al valore massimo presente nell'array.
- h. Restituire true se e solo se l'array è ordinato in senso crescente.
- i. Restituire true se e solo se l'array contiene due elementi adiacenti duplicati.
- j. Restituire true se e solo se l'array contiene elementi duplicati (non necessariamente adiacenti).

**\*\* E7.11.** Nella classe seguente

```

public class Sequence
{
    private int[] values;
    public Sequence(int size) { values = new int[size]; }
    public void set(int i, int n) { values[i] = n; }
    public int get(int i) { return values[i]; }
    public int size() { return values.length; }
}

```

aggiungete il metodo

```
public boolean equals(Sequence other)
```

che verifichi se due sequenze contengono gli stessi valori, nello stesso ordine.

**\*\* E7.12.** Aggiungete alla classe Sequence dell'esercizio precedente il metodo

```
public boolean sameValues(Sequence other)
```

che verifichi se due sequenze contengono gli stessi valori, indipendentemente dall'ordine e ignorando la presenza di valori duplicati. Ad esempio, le due sequenze

1 4 9 16 9 7 4 9 11

e

11 11 7 9 16 4 1

devono essere considerate uguali. Probabilmente avrete bisogno di qualche metodo ausiliario.

- \*\*\* **E7.13.** Aggiungete alla classe `Sequence` dell'Esercizio E7.11 il metodo

```
public boolean isPermutationOf(Sequence other)
```

che verifichi se due sequenze contengono gli stessi valori con le stesse molteplicità, indipendentemente dall'ordine. Ad esempio, la sequenza

1 4 9 16 9 7 4 9 11

è una permutazione della sequenza

11 1 4 9 16 9 7 4 9

mentre la sequenza

1 4 9 16 9 7 4 9 11

non è una permutazione della sequenza

11 11 7 9 16 4 1 4 9

Probabilmente avrete bisogno di qualche metodo ausiliario.

- \*\*\* **E7.14.** Aggiungete alla classe `Sequence` dell'Esercizio E7.11 il metodo

```
public Sequence sum(Sequence other)
```

che restituisca una sequenza i cui elementi siano la somma degli elementi in posizioni corrispondenti nelle due sequenze elaborate. Ad esempio, la somma tra la sequenza

1 4 9 16 9 7 4 9 11

e la sequenza

11 11 7 9 16 4 1

è la sequenza

12 15 16 25 25 11 5 9 11

- \*\*\* **E7.15.** Scrivete un programma che generi una sequenza di 20 valori casuali compresi tra 0 e 99, memorizzandola in un array, poi visualizzzi la sequenza generata, la ordini e la visualizzi di nuovo, ordinata. Usate il metodo `sort` della libreria standard di Java.

- \*\*\* **E7.16.** Aggiungete alla seguente classe `Table` un metodo che calcoli il valore medio tra gli otto valori adiacenti nelle diverse direzioni (come visualizzato nella Figura 15 del Paragrafo 7.6.3) a un determinato elemento della tabella:

```
public double neighborAverage(int row, int column)
```

Se l'elemento in esame si trova in un bordo della tabella considerate soltanto gli elementi adiacenti che appartengono effettivamente alla tabella stessa: ad esempio, se `row` e `column` valgono 0, esistono soltanto tre elementi adiacenti.

```
public class Table
{
    private int[][] values;
    public Table(int rows, int columns) { values = new int[rows][columns]; }
    public void set(int i, int j, int n) { values[i][j] = n; }
}
```

- \*\* **E7.17.** Aggiungete alla classe `Table` dell'esercizio precedente il metodo ~~`sum(int, boolean)`~~

```
public double sum(int i, boolean horizontal)
```

che restituisca la somma degli elementi della riga  $i$ -esima (se il valore di `horizontal` è `true`) o della colonna  $i$ -esima (se il valore di `horizontal` è `false`).

- \*\* **E7.18.** Scrivete un programma che legga una sequenza di valori in ingresso e visualizzi un diagramma a barre corrispondente ai valori acquisiti, simile a questo:

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

Potete ipotizzare che i valori siano tutti positivi. Per prima cosa individuate il valore massimo: la barra corrispondente a quel valore deve avere 40 asterischi e le altre devono avere un numero di asterischi proporzionale al loro valore rispetto a tale valore massimo.

- \*\*\* **E7.19.** Risolvete nuovamente l'esercizio precedente, disegnando però le barre verticalmente, con la barra più alta costituita da venti asterischi.  
 \*\*\* **E7.20.** Migliorate il programma dell'Esercizio E7.18 in modo che possa gestire correttamente anche valori negativi.  
 \*\* **E7.21.** Migliorate il programma dell'Esercizio E7.18 aggiungendo un'etichetta descrittiva a ciascuna barra del diagramma, dopo aver chiesto all'utente il testo delle etichette e i valori. Il diagramma visualizzato deve essere simile a questo:

Egypt	*****
France	*****
Japan	*****
Uruguay	*****
Switzerland	*****

- \* **E7.22.** Nella classe seguente

```
public class Sequence
{
    private ArrayList<Integer> values;
    public Sequence() { values = new ArrayList<Integer>(); }
    public void add(int n) { values.add(n); }
    public String toString() { return values.toString(); }
}
```

aggiungete il metodo

```
public Sequence append(Sequence other)
```

che crei una nuova sequenza, ottenuta aggiungendo il contenuto della sequenza *other* in fondo a quello della sequenza in esame, senza modificare nessuna delle due. Se, ad esempio, *a* è la sequenza

1 4 9 16

e *b* è la sequenza

9 7 4 9 11

allora l'invocazione *a.append(b)* restituisce la sequenza

1 4 9 16 9 7 4 9 11

senza modificare *a* o *b*.

- \*\* E7.23. Aggiungete alla classe Sequence dell'esercizio precedente il metodo

```
public Sequence merge(Sequence other)
```

che restituisca una sequenza i cui elementi siano presi alternativamente dalle due sequenze elaborate. Se una delle due sequenze è più corta dell'altra, dopo aver terminato l'alternanza bisogna aggiungere al risultato gli elementi rimasti nella sequenza più lunga. Se, ad esempio, *a* è la sequenza

1 4 9 16

e *b* è la sequenza

9 7 4 9 11

allora l'invocazione *a.merge(b)* restituisce la sequenza

1 9 4 7 9 4 16 9 11

senza modificare *a* o *b*.

- \*\* E7.24. Aggiungete alla classe Sequence dell'Esercizio E7.22 il metodo

```
public Sequence mergeSorted(Sequence other)
```

che fonda due sequenze ordinate, generando una sequenza ordinata. Usate un indice per ciascuna sequenza, per tenere traccia della porzione che è già stata elaborata. Ad ogni passo aggiungere alla sequenza in costruzione l'elemento minimo tra quelli non ancora elaborati, prendendolo da una delle due sequenze di partenza, incrementando l'indice di conseguenza. Se, ad esempio, *a* è la sequenza

1 4 9 16

e b è la sequenza

4 7 9 9 11

allora l'invocazione `a.mergeSorted(b)` restituisce la sequenza

1 4 4 7 9 9 9 11 16

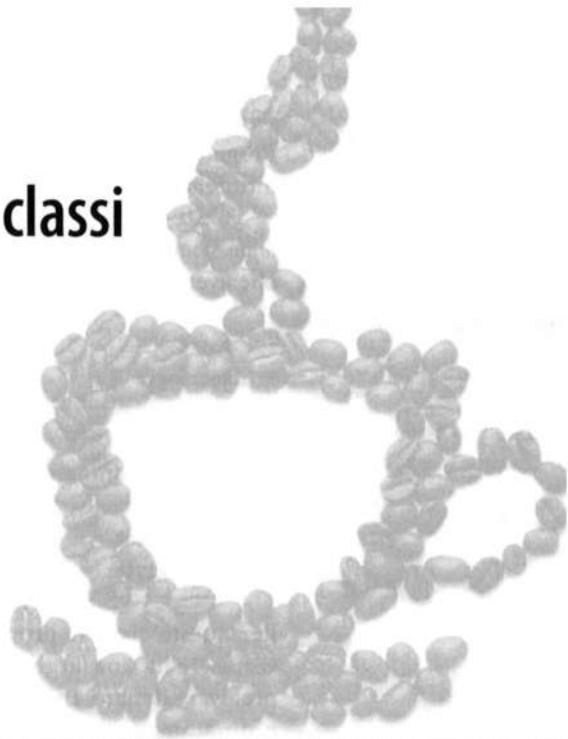
senza modificare a o b. In caso di sequenze non originariamente ordinate fondete i più lunghi prefissi ordinati di a e b.

**Sul sito web dedicato al libro si trova una raccolta di progetti di programmazione più complessi.**



# 8

## Progettazione di classi



### Obiettivi del capitolo

---

- Imparare a identificare le classi più appropriate per risolvere un problema
- Assimilare il concetto di coesione
- Ridurre al minimo le dipendenze e gli effetti collaterali
- Capire come si identifica la rappresentazione dei dati di una classe
- Apprendere l'utilizzo di metodi statici e variabili statiche
- Saper utilizzare i pacchetti
- Imparare a usare ambienti per il collaudo di unità

Un buon progetto deve essere sia funzionale sia elegante: quando si progettano classi, bisogna ricordare che ogni singola classe dovrebbe essere dedicata a uno scopo preciso e definito, e che le classi usate in un programma devono collaborare per il raggiungimento dell'obiettivo finale. In questo capitolo imparerete a identificare le classi, a progettare buoni metodi e a scegliere una rappresentazione dei dati adeguata. Vedrete, inoltre, come usare i pacchetti per organizzare le vostre classi e come progettare funzionalità e caratteristiche che appartengono alla classe nel suo complesso e non a singoli oggetti. Infine, presenteremo l'infrastruttura di collaudo JUnit, utile per verificare le funzionalità delle classi che progetterete.

## 8.1 Individuare le classi

Nei capitoli precedenti avete usato un buon numero di classi e probabilmente, nello svolgimento degli esercizi di programmazione, avete anche progettato alcune vostre classi. Progettare una classe può essere una sfida: non sempre è facile capire come iniziare o intuire se il risultato sarà di buona qualità.

Cosa rende buona una classe? La cosa più importante è che una classe dovrebbe rappresentare un singolo concetto all'interno del dominio del problema. Alcune delle classi che avete visto rappresentano concetti matematici

- Point
- Rectangle
- Ellipse2D

Una classe dovrebbe rappresentare un singolo concetto nel dominio in cui è descritto il problema: le scienze, la matematica o l'economia.

mentre altre sono astrazioni di entità della vita reale

- BankAccount
- CashRegister

Per queste classi le proprietà di un oggetto sono di facile comprensione: un oggetto Rectangle ha una larghezza e un'altezza e un oggetto BankAccount consente di versare e prelevare denaro. In generale, concetti che appartengono all'ambito a cui si riferisce il programma (come, ad esempio, le scienze, l'economia o il gioco) costituiscono buone classi, il cui nome dovrebbe essere un sostantivo che descrive il concetto. In effetti, una delle semplici regole che consentono di iniziare un progetto di classi afferma che occorre esaminare prima di tutto i sostantivi che compaiono nella descrizione del problema.

Un'altra utile categoria di classi può essere descritta come quella degli *attori*: un oggetto di una classe che agisce nel ruolo di attore svolge un lavoro per voi. Esempi di attori sono la classe Scanner vista nel Capitolo 4 e la classe Random del Capitolo 6: un oggetto Scanner identifica numeri e stringhe in un flusso di caratteri, mentre un oggetto Random genera numeri casuali. Solitamente, scegliere per questo genere di classi un nome che termini in "er" oppure "or" è una buona idea (RandomNumberGenerator potrebbe essere un nome migliore per la classe Random).

In alcuni rari casi una classe non serve a creare oggetti, ma contiene una raccolta di metodi statici e di costanti fra loro correlati: la classe Math è un esempio tipico e si parla di classe "di utilità" (*utility class*).

Infine, avete visto classi il cui unico scopo è quello di iniziare l'esecuzione di un programma e, pertanto, hanno soltanto il metodo main. Da un punto di vista progettuale, questi sono esempi di classi piuttosto degeneri.

Quale potrebbe essere una classe poco valida? Se dal nome della classe non siete in grado di capire cosa potrebbe fare un suo oggetto, allora probabilmente non siete sulla strada giusta. Supponiamo che una delle esercitazioni che vi è stata assegnata chieda di scrivere un programma che stampa buste paga (*paycheck*) e che abbiate iniziato a progettare la classe PaycheckProgram: cosa dovrebbe fare un oggetto di tale classe? Dovrebbe fare tutto quello che viene richiesto nell'esercitazione: ciò non semplifica molto le cose. Una classe

migliore potrebbe essere Paycheck: il vostro programma potrebbe, quindi, gestire uno o più oggetti di tipo Paycheck.

Un altro tipico errore consiste nel trasformare un'azione in una classe. A esempio, se dovete calcolare una busta paga, potreste pensare di scrivere una classe ComputePaycheck: ma potete immaginare un oggetto descritto come "Calcola una busta paga"? Il fatto che "Calcola una busta paga" non sia un sostantivo vi suggerisce che non siete sulla strada giusta. D'altra parte, una classe Paycheck ha, intuitivamente, senso: potete immaginare un oggetto "busta paga" e potete, quindi, pensare a metodi utili della classe Paycheck, come computeTaxes, che vi aiutino a risolvere il problema.



## Auto-valutazione

1. Qual è una semplice regola pratica per l'identificazione delle classi?
2. Supponete di dover scrivere un programma che gioca a scacchi. Sarebbe opportuno progettare una classe ChessBoard ("scacchiera")? E una classe MovePiece ("muovi un pezzo")?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R8.3, R8.4 e R8.5, al termine del capitolo.

## 8.2 Progettare buoni metodi

In questo paragrafo imparerete alcuni criteri utili per analizzare e migliorare la qualità dell'interfaccia pubblica di una classe.

### 8.2.1 Un'interfaccia pubblica coesa

L'interfaccia pubblica di una classe ha una buona coesione se tutte le sue caratteristiche sono correlate al concetto rappresentato dalla classe.

Una classe dovrebbe rappresentare un singolo concetto; tutte le caratteristiche della sua interfaccia pubblica dovrebbero avere una buona **coesione**, cioè dovrebbero essere strettamente correlate al singolo concetto rappresentato dalla classe.

Se scoprirete che l'interfaccia pubblica di una classe si riferisce a più concetti, forse è giunto il momento di usare più classi distinte. Considerate, ad esempio, l'interfaccia pubblica della classe CashRegister del Capitolo 4:

```
public class CashRegister
{
    public static final double QUARTER_VALUE = 0.25;
    public static final double DIME_VALUE = 0.1;
    public static final double NICKEL_VALUE = 0.05;
    ...
    public void receivePayment(int dollars, int quarters,
                               int dimes, int nickels, int pennies)
    ...
}
```

Qui, in verità, sono presenti due concetti: i valori delle singole monete e un registratore di cassa che contiene monete e calcola il loro valore totale (per semplicità immaginiamo

che il registratore di cassa contenga soltanto monete e non banconote, ma l'Esercizio E8.3 analizza una soluzione più generale).

Potrebbe essere più sensato avere una classe `Coin` (*moneta*) separata, in modo che ciascuna moneta abbia la responsabilità di conoscere il proprio valore.

```
public class Coin
{
    . .
    public Coin(double aValue, String aName) { ... }
    public double getValue() { ... }
    . .
}
```

La classe `CashRegister` si potrebbe, quindi, semplificare così:

```
public class CashRegister
{
    . .
    public void receivePayment(int coinCount, Coin coinType)
    {
        payment = payment + coinCount * coinType.getValue();
    }
    . .
}
```

A questo punto la classe `CashRegister` non ha più bisogno di contenere informazioni relative ai valori delle monete: la stessa classe è in grado di gestire anche euro o altre valute!

Questa è, evidentemente, una soluzione migliore, perché separa le responsabilità del registratore di cassa da quelle delle monete. L'unico motivo per cui non abbiamo seguito questo approccio nel Capitolo 4 è stato quello di proporre un esempio semplice.

## 8.2.2 Minimizzare le dipendenze

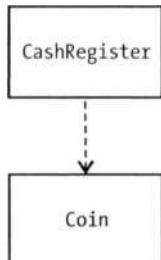
Una classe dipende da un'altra se i suoi metodi la usano in qualche modo.

Molti metodi hanno bisogno di altre classi per svolgere il proprio compito. Ad esempio, il metodo `receivePayment` della classe `CashRegister` appena ristrutturata usa ora la classe `Coin` per determinare il totale da pagare: diciamo che la classe `CashRegister` *dipende* dalla classe `Coin`.

Per visualizzare relazioni come la dipendenza tra classi, i programmati usano diagrammi di classi. In questo libro usiamo la notazione UML (“Unified Modeling Language”) per oggetti e classi: usata per l’analisi e la progettazione orientate agli oggetti, fu inventata da Grady Booch, Ivar Jacobson e James Rumbaugh, tre ricercatori di spicco nello sviluppo di software orientato agli oggetti. La notazione UML distingue fra *diagrammi di oggetti* e *diagrammi di classi*. In un diagramma di classi si annota una dipendenza mediante una linea tratteggiata che termina con una freccia aperta e punta alla classe nei confronti della quale esiste la dipendenza. La Figura 1 mostra un diagramma di classi che indica che la classe `CashRegister` dipende dalla classe `Coin`.

**Figura 1**

Relazione di dipendenza:  
la classe CashRegister  
dipende dalla classe Coin



Notate che la classe `Coin` *non* dipende dalla classe `CashRegister`: tutti i metodi della classe `Coin` svolgono i propri compiti senza mai invocare alcun metodo della classe `CashRegister`. In teoria, le monete non sanno nemmeno di poter essere contenute in un registratore di cassa.

Vediamo un esempio di come si possano minimizzare le dipendenze, analizzando come abbiamo sempre visualizzato il saldo di un conto bancario:

```
System.out.println("The balance is now $" + mom'sSavings.getBalance());
```

Perché non abbiamo semplicemente inserito nella classe `BankAccount` un metodo che visualizzi il saldo?

```
public void printBalance() // sconsigliato
{
    System.out.println("The balance is now $" + balance);
}
```

Questo metodo dipende da `System.out`, ma non tutti gli ambienti di elaborazione dispongono dell'oggetto `System.out`: ad esempio, uno sportello bancario automatico non visualizza i messaggi diretti alla finestra di console. In altre parole, questa scelta di progettazione viola la regola che impone di minimizzare le dipendenze tra classi, perché il metodo `printBalance` rende la classe `BankAccount` dipendente dalle classi `System` e `PrintStream`.

È preferibile che il codice che visualizza informazioni in uscita o acquisisce informazioni in ingresso si trovi in una classe separata: in questo modo, l'attività di input e di output viene completamente disaccoppiata dal lavoro di elaborazione effettivamente svolto dalle classi che progettate.

### 8.2.3 Tenere distinti accessi e modifiche

Un **metodo modificatore** (*mutator method*) modifica lo stato dell'oggetto con cui viene invocato, mentre un **metodo d'accesso** (*accessor method*) chiede soltanto all'oggetto di calcolare un risultato, senza modificarne lo stato.

Alcune classi, dette **immutabili**, sono progettate in modo da avere solamente metodi d'accesso, senza alcun metodo modificatore. Un esempio è la classe `String`: una volta costruita una stringa, il suo contenuto non cambia mai e nessun metodo della classe `String` può modificare il contenuto della stringa con cui viene invocato. Ad esempio, il metodo `toUpperCase` non modifica i caratteri di una stringa, ma costruisce una *nuova* stringa, che contiene caratteri maiuscoli:

Una classe immutabile  
non ha metodi modificatori.

```
String name = "John Q. Public";
String uppercased = name.toUpperCase(); // la stringa name non viene modificata
```

I riferimenti a oggetti di una classe immutabile possono essere condivisi in sicurezza.

Una classe immutabile offre un vantaggio molto importante: i riferimenti ai suoi oggetti si possono distribuire liberamente, in sicurezza. Se nessun metodo può cambiare lo stato degli oggetti, nessun codice può modificare un oggetto in un momento inaspettato.

Naturalmente non tutte le classi devono essere immutabili: l'immutabilità ha senso principalmente per classi che rappresentano valori, come stringhe, date, quantità di denaro, colori, e così via.

Nelle classi mutabili è comunque preferibile separare in modo chiaro i metodi modificatori dai metodi d'accesso, per evitare modifiche accidentali. Come regola pratica, un metodo che restituisce un valore non dovrebbe essere anche un modificatore. Ad esempio, nessuno immagina che l'invocazione di `getBalance` con un oggetto `BankAccount` possa modificarne il saldo (sareste certamente meravigliati se la vostra banca applicasse una commissione anche per la semplice ispezione del saldo). Se seguite questa regola, tutti i metodi modificatori delle vostre classi saranno “di tipo `void`”.

Qualche volta la regola viene seguita in modo un po' flessibile e anche i metodi modificatori restituiscono un valore. Ad esempio, la classe `ArrayList` mette a disposizione il metodo `remove`, che elimina un elemento:

```
ArrayList<String> names = . . . ;
boolean success = names.remove("Romeo");
```

Quel metodo restituisce `true` se la rimozione è andata a buon fine, cioè se il vettore conteneva l'oggetto da eliminare. La restituzione di tale valore non sarebbe una buona scelta di progetto se non ci fossero altri modi per sapere se l'oggetto da rimuovere è presente nel vettore, ma esiste un metodo (`contains`) che svolge proprio quel compito: si ritiene accettabile che un metodo modificatore restituisca un'informazione se esiste anche un metodo d'accesso che la potrebbe restituire.

La situazione è meno soddisfacente nel caso della classe `Scanner`, il cui metodo `next` è un modificatore che restituisce un valore (tale metodo è effettivamente un modificatore, perché se lo invocate due volte consecutivamente restituisce due risultati, in generale, diversi, quindi deve aver modificato qualcosa all'interno dell'oggetto `Scanner`), ma, sfortunatamente, non esiste un metodo d'accesso che sia in grado di restituire lo stesso valore. Questa caratteristica rende a volte problematico l'utilizzo di uno `Scanner`: bisogna fare molta attenzione a memorizzare o elaborare ciascun valore restituito dal metodo `next`, perché non ci sarà una seconda possibilità di ispezionarlo. Sarebbe decisamente preferibile che ci fosse un altro metodo, che si potrebbe chiamare `peek`, che restituisca (anche ripetutamente) il valore successivo in ingresso, senza “consumarlo”.

## 8.2.4 Minimizzare gli effetti collaterali

Un effetto collaterale di un metodo è una modifica apportata ai dati che sia osservabile al di fuori del metodo stesso: i metodi modificatori hanno un effetto collaterale, perché modificano il proprio parametro implicito.

Altri effetti collaterali, però, andrebbero evitati. In generale, un metodo non dovrebbe modificare le proprie variabili parametro. Consideriamo questo esempio:

```

    /**
     * Calcola il saldo totale dei conti bancari ricevuti.
     * @param accounts un vettore di conti bancari
    */
    public double getTotalBalance(ArrayList<BankAccount> accounts)
    {
        double sum = 0;
        while (accounts.size() > 0)
        {
            BankAccount account = accounts.remove(0); // sconsigliato
            sum = sum + account.getBalance();
        }
        return sum;
    }

```

Questo metodo *elimina* tutti i conti bancari dal vettore a cui fa riferimento la variabile parametro `accounts`. Dopo aver invocato

```
double total = getTotalBalance(allAccounts);
```

il vettore `allAccounts` è vuoto! Un effetto collaterale come questo risulterebbe probabilmente inatteso per la maggior parte dei programmati: è molto meglio che il metodo ispezioni gli elementi presenti nel vettore senza eliminarli.

Un altro esempio di effetto collaterale è la visualizzazione di dati in uscita. Consideriamo di nuovo il metodo `printBalance` discusso nel Paragrafo 8.2.2:

```

public void printBalance() // sconsigliato
{
    System.out.println("The balance is now $" + balance);
}

```

Questo metodo modifica l'oggetto `System.out`, che non fa parte dell'oggetto `BankAccount` con cui è stato invocato: è un effetto collaterale.

Per evitare questo tipo di effetti collaterali, è meglio che le classi che progettate non compiano attività di acquisizione e visualizzazione di dati, che andranno concentrate in un'unica posizione, che solitamente sarà il metodo `main` del programma.

Quando si progettano metodi  
è bene minimizzarne  
gli effetti collaterali.



## Auto-valutazione

3. Perché la classe `CashRegister` del Capitolo 4 non ha una buona coesione?
4. Perché la classe `Coin` non dipende dalla classe `CashRegister`?
5. Perché è meglio minimizzare le dipendenze tra classi?
6. Il metodo `substring` della classe `String` è un metodo d'accesso o un metodo modificatore?
7. La classe `Rectangle` è immutabile?
8. Se a fa riferimento a un conto bancario, l'invocazione `a.deposit(100)` modifica l'oggetto  
a. Si tratta di un effetto collaterale?
9. Considerate la classe `Student` del Capitolo 7 e immaginate di aggiungervi questo metodo:  

```
void read(Scanner in)
{
    while (in.hasNextDouble())
    {
```

```
        addScore(in.nextDouble());
    }
}
```

Oltre a modificare le valutazioni memorizzate, questo metodo ha un effetto collaterale?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R8.6, R8.7 e R8.11, al termine del capitolo.



## Suggerimenti per la programmazione 8.1

### Coerenza

In questo paragrafo avete appreso due criteri per analizzare la qualità dell'interfaccia pubblica di una classe: bisogna cercare di massimizzare la coesione e di eliminare le dipendenze non indispensabili. C'è, però, un altro criterio a cui dovreste porre attenzione: la *coerenza*. Quando progettate un insieme di metodi, seguite uno schema coerente per i loro nomi e i loro parametri: questo è, banalmente, un marchio di buona fattura.

Purtroppo troverete un certo numero di incoerenze anche nella libreria standard. Ecco un esempio: per mostrare una finestra di dialogo che riceve valori in ingresso invocate il metodo

```
JOptionPane.showInputDialog(promptString)
```

mentre per mostrare una finestra di dialogo che visualizza un messaggio invocate il metodo

```
JOptionPane.showMessageDialog(null, messageString)
```

A cosa serve, in questo secondo caso, il parametro `null`? Il metodo `showMessageDialog` vuole un parametro che specifichi la finestra di appartenenza, parametro che deve valere `null` se questa non è necessaria. Il metodo `showInputDialog`, però, non richiede una tale finestra. Perché questa incoerenza? Non c'è alcun motivo: sarebbe stato semplice mettere a disposizione un metodo `showMessageDialog` avente la stessa firma del metodo `showInputDialog`.

Incoerenze come questa non sono errori gravi, ma sono comunque fastidiose, soprattutto perché si potrebbero evitare molto facilmente.



## Argomenti avanzati 8.1

### Invocazione per valore e invocazione per riferimento

Nel Paragrafo 8.2.4 vi abbiamo raccomandato di non invocare un metodo modificatore usando una variabile parametro. In questa sezione parleremo di un argomento correlato: cosa accade quando si assegna un nuovo valore a una variabile parametro? Considerate questo metodo:

```
public class BankAccount
{
    ...
    /**

```

```

    Trasferisce denaro da questo conto
    e cerca di aggiungerlo al saldo di un altro.
    @param amount la somma di denaro da trasferire
    @param otherBalance il saldo a cui aggiungere il denaro
*/
public void transfer(double amount, double otherBalance) ②
{
    balance = balance - amount;
    otherBalance = otherBalance + amount;
    // l'argomento non verrà aggiornato
} ③
}

```

Vediamo ora cosa succede quando si invoca il metodo `transfer`:

```

double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance); ①
System.out.println(savingsBalance); ④

```

Forse state pensando che, dopo l'invocazione del metodo `transfer`, la variabile `savingsBalance` abbia assunto il valore 1500, ma non è così. Nel momento in cui il metodo inizia la propria esecuzione, alla variabile parametro `otherBalance` viene assegnato il valore di `savingsBalance` (come si può vedere nella Figura 2). Quindi, è questa variabile, `otherBalance`, che viene ad assumere un nuovo valore, e questa modifica non ha alcun effetto su `savingsBalance`, semplicemente perché `otherBalance` è una variabile diversa. Quando il metodo termina, la variabile `otherBalance` viene eliminata e `savingsBalance` non è stata incrementata.

In Java, le variabili parametro vengono inizializzate con i valori delle espressioni usate come argomenti. Quando, poi, il metodo termina la propria esecuzione, le variabili parametro vengono eliminate. Gli informatici chiamano questo meccanismo invocazione (o chiamata) “per valore”.

In virtù di questa semantica, in Java un metodo non può mai modificare il contenuto di una variabile che gli viene passata come argomento: il metodo elabora una variabile diversa.

Altri linguaggi di programmazione, come C++, consentono l'utilizzo di un meccanismo alternativo, l'invocazione “per riferimento”, che è in grado di modificare l'argomento usato nell'invocazione di un metodo. Talvolta nei manuali di Java leggerete che in questo linguaggio “i numeri sono passati per valore e gli oggetti sono passati per riferimento”. Tecnicamente ciò non è del tutto corretto, perché in Java sia i numeri sia i riferimenti a oggetto vengono passati per valore.

La confusione può nascere perché, in Java, un metodo è in grado di modificare un oggetto ricevuto come parametro, dal momento che ne riceve un riferimento (Figura 3).

```

public class BankAccount
{
    /**
     * Trasferisce denaro da questo conto a un altro
     * @param amount la somma di denaro da trasferire
     * @param otherAccount il conto a cui trasferire il denaro
}

```

In Java, un metodo non può mai modificare il contenuto di una variabile che gli viene passata come argomento.

```

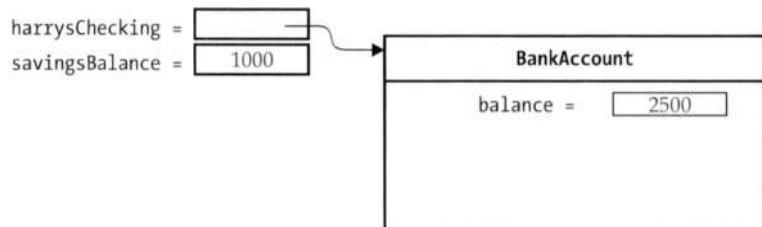
public void transfer(double amount, BankAccount otherAccount) ②
{
    balance = balance - amount;
    otherAccount.deposit(amount);
} ③
}

```

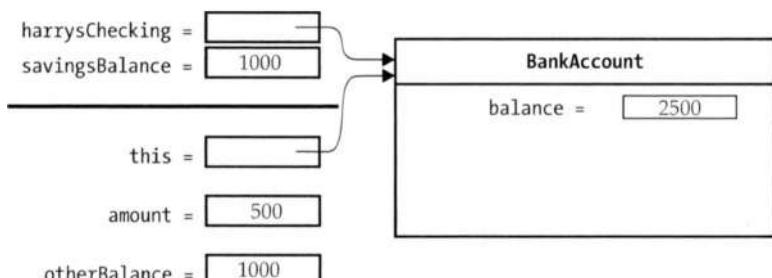
**Figura 2**

La modifica di una variabile  
parametro di tipo primitivo  
non ha alcun effetto su chi  
ha invocato il metodo

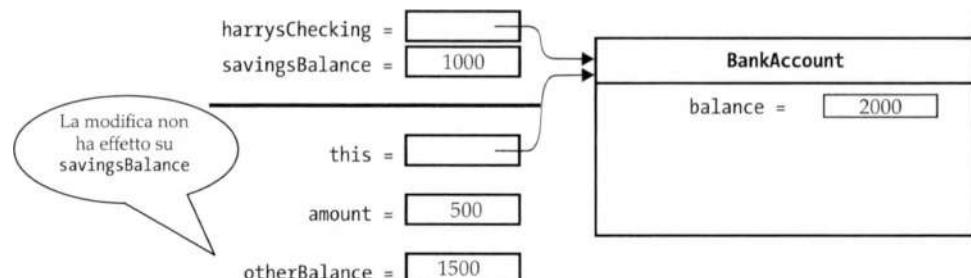
- ① Prima dell'invocazione del metodo



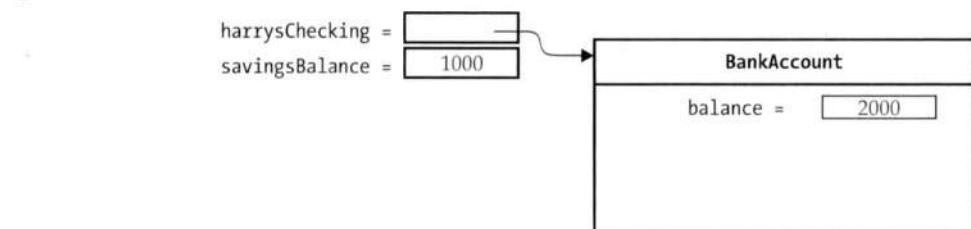
- ② Inizializzazione dei parametri del metodo



- ③ Subito prima di terminare l'esecuzione del metodo



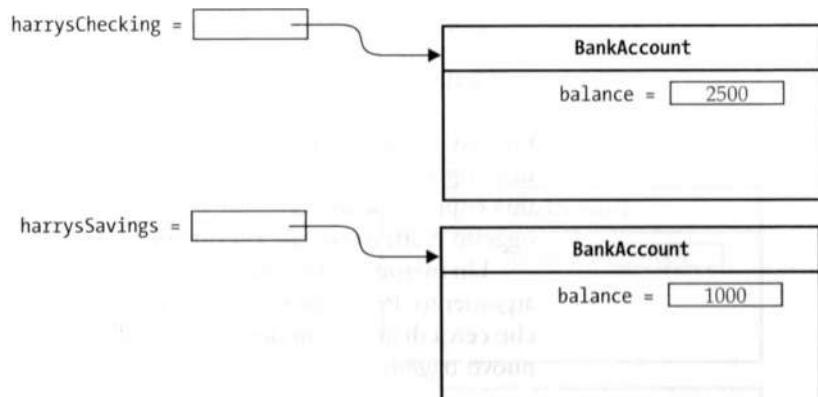
- ④ Dopo l'invocazione del metodo



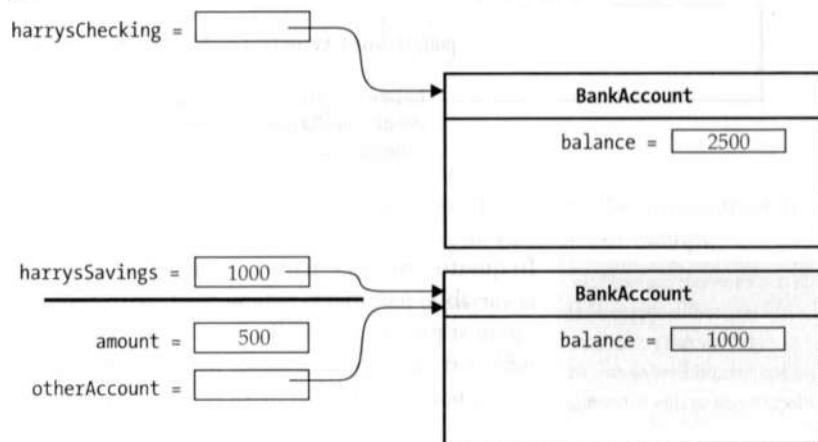
**Figura 3**

Un metodo può modificare gli oggetti di cui ha ricevuto un riferimento

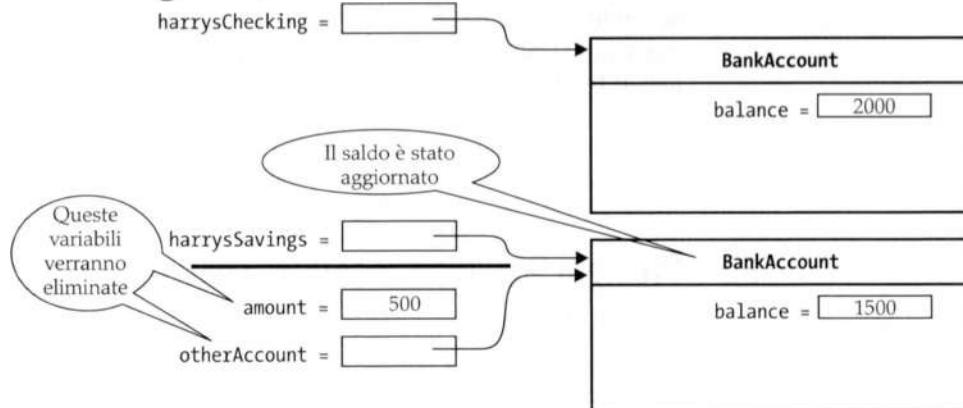
① Prima dell'invocazione del metodo



② Inizializzazione dei parametri del metodo



③ Subito prima di terminare l'esecuzione del metodo



In questo caso al metodo `transfer` viene passato un riferimento a un oggetto:

```
BankAccount harrysSavings = new BankAccount(1000);
harrysChecking.transfer(500, harrysSavings); ①
System.out.println(harrysSavings.getBalance());
```

Questo esempio funziona come ci si aspetta: la variabile parametro `otherAccount` contiene una copia del riferimento `harrysSavings`. Nel Paragrafo 2.8 avete visto cosa significa fare una copia di un riferimento a un oggetto: si ottiene un altro riferimento al medesimo oggetto e, attraverso tale riferimento, il metodo è in grado di modificare l'oggetto stesso.

Un metodo, comunque, non può *sostituire* un riferimento che gli viene passato come argomento. Per apprezzare questa sottile differenza, consideriamo il metodo seguente, che cerca di fare in modo che la variabile parametro `otherAccount` faccia riferimento a un nuovo oggetto:

```
public class BankAccount
{
    ...
    public void transfer(double amount, BankAccount otherAccount)
    {
        balance = balance - amount;
        double newBalance = otherAccount.balance + amount;
        otherAccount = new BankAccount(newBalance); // non funzionerà
    }
}
```

In Java, un metodo può modificare lo stato di un oggetto di cui ha ricevuto un riferimento, ma non può sostituire il riferimento all'oggetto con un altro riferimento.

In questo caso non stiamo cercando di modificare lo *stato* dell'oggetto a cui fa riferimento la variabile parametro `otherAccount`; vorremmo, invece, sostituire l'oggetto con uno nuovo, come si può vedere nella Figura 4. All'interno del metodo, il riferimento memorizzato nella variabile parametro `otherAccount` viene sostituito con un riferimento a un nuovo conto bancario, ma, se invochiamo il metodo in questo modo

```
harrysChecking.transfer(500, savingsAccount);
```

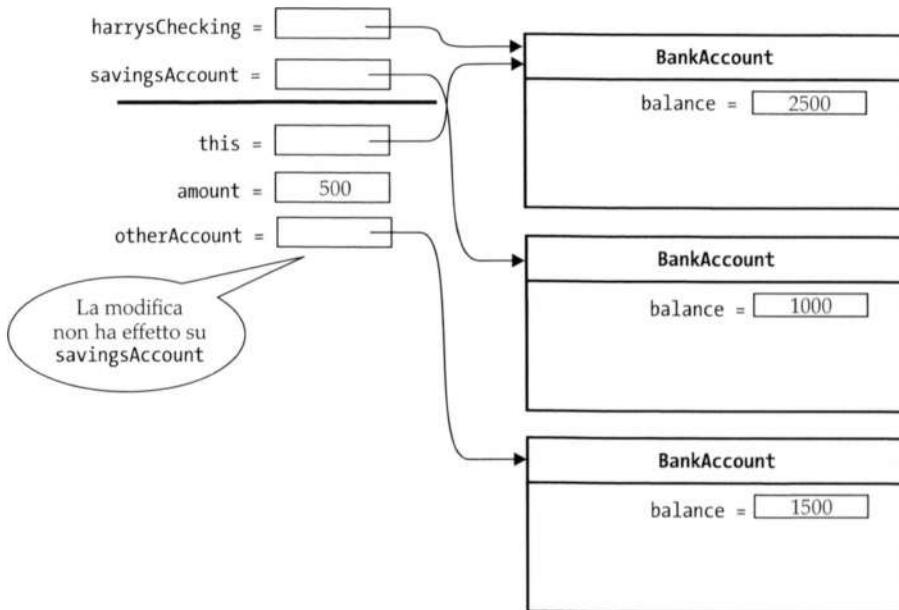
la modifica non ha alcun effetto sulla variabile `savingsAccount` che è stata usata come argomento nell'invocazione. Questo esempio, quindi, dimostra che gli oggetti non vengono passati per riferimento. Per riassumere, in Java un metodo:

- non può modificare il contenuto di una variabile passata come argomento;
- può modificare un oggetto di cui riceve il riferimento come argomento.

## 8.3 Problem Solving: progettare i dati di un oggetto

Quando si progetta una classe bisogna innanzitutto considerare ciò che servirà ai programmatore che la useranno, mettendo loro a disposizione i metodi necessari da invocare per manipolare oggetti di tale classe. Quando, poi, si passa all'implementazione della classe, bisogna individuarne le variabili di esemplare, cosa non sempre banale. Per nostra fortuna, c'è un insieme di schemi ricorrenti che possono essere adattati al progetto delle nostre classi e ne parleremo nei prossimi paragrafi.

**Figura 4**  
La sostituzione  
di un riferimento  
in una variabile  
parametro non ha effetti  
sull'invocante



### 8.3.1 Gestione di un totale

Una variabile di esemplare che conserva un totale viene aggiornata nei metodi che lo incrementano o lo decrementano.

Molte classi hanno bisogno di tenere traccia di una quantità totale, che può aumentare o diminuire quando vengono invocati determinati metodi. Ecco alcuni esempi:

- un conto bancario ha un saldo che aumenta a ogni versamento e diminuisce a ogni prelievo;
- un registratore di cassa ha un totale che aumenta quando un articolo viene aggiunto allo scontrino e torna a zero quando l'acquisto termina;
- un'automobile ha nel serbatoio una quantità di carburante che aumenta quando si fa rifornimento e diminuisce quando si guida.

In tutti questi casi la strategia realizzativa è decisamente simile: si usa una variabile di esemplare che rappresenta il totale attuale. Ad esempio, nel caso del registratore di cassa:

```
private double purchase;
```

Poi, individuate i metodi che modificano il totale. Solitamente c'è un metodo che lo incrementa di una certa quantità:

```
public void recordPurchase(double amount)
{
    purchase = purchase + amount;
}
```

In relazione alla natura della classe, ci può poi essere un metodo che decrementa il totale o che lo azzerà. Nel caso del registratore di cassa, si può aggiungere un metodo che azzeri il totale:

```
public void clear()
{
    purchase = 0;
}
```

Infine, di solito c'è un metodo che consente di ispezionare il totale attuale, di facile implementazione:

```
public double getAmountDue()
{
    return purchase;
}
```

Tutte le classi che gestiscono un totale seguono questo schema fondamentale. Individuate i metodi che modificano il totale e scrivete il codice che serve per aumentarlo o diminuirlo. Quindi, identificate i metodi che restituiscono o usano il totale e fate in modo che lo leggano quando necessario.

### 8.3.2 Conteggio di eventi

Un contatore di eventi viene incrementato nei metodi che corrispondono a tali eventi.

Spesso durante la vita di un oggetto c'è bisogno di contare quante volte si verificano determinati eventi. Ad esempio:

- in un registratore di cassa si può voler sapere quanti articoli sono stati aggiunti a uno scontrino;
- un conto bancario può addebitare una commissione per ogni transazione effettuata, per cui c'è bisogno di contarle.

Usate un contatore, come questo:

```
private int itemCount;
```

Incrementate il contatore in quei metodi che corrispondono agli eventi che volete contare:

```
public void recordPurchase(double amount)
{
    purchase = purchase + amount;
    itemCount++;
}
```

A volte ci può essere bisogno di azzerare il contatore, ad esempio quando viene emesso uno scontrino oppure un estratto conto mensile:

```
public void clear()
{
    purchase = 0;
    itemCount = 0;
}
```

Può darsi che sia presente un metodo che fornisce il conteggio all'utente della classe, oppure no: il conteggio potrebbe essere utilizzato soltanto per calcolare una commissione dovuta o un valore medio, all'interno dei metodi della classe. Scoprite quali metodi della classe usano il valore del contatore e fate in modo che leggano il valore attuale.

### 8.3.3 Gestione di una raccolta di valori

Alcuni oggetti contengono una raccolta di numeri, stringhe o altri oggetti. Ad esempio, una domanda con risposta a scelta multipla è caratterizzata da un certo numero di risposte alternative, mentre un registratore di cassa può aver bisogno di memorizzare tutti i prezzi degli articoli dell'acquisto in corso.

Un oggetto può memorizzare una raccolta di altri oggetti in un array o in un vettore.

Per memorizzare una raccolta di valori usate un array o un vettore (solitamente è più semplice usare un vettore, perché non ci si deve preoccupare del numero di valori). Ad esempio:

```
public class Question
{
    private ArrayList<String> choices;
    ...
}
```

Nel costruttore, inizializzate la variabile di esemplare in modo che contenga una raccolta vuota:

```
public Question()
{
    choices = new ArrayList<String>();
}
```

Bisogna, poi, fornire qualche metodo per aggiungere valori alla raccolta. Di solito si progetta un metodo che aggiunge un valore alla fine della raccolta:

```
public void add(String option)
{
    choices.add(option);
}
```

L'utente di un oggetto di tipo `Question` può invocare più volte il metodo `add` per aggiungere le varie risposte.

### 8.3.4 Gestione delle proprietà di un oggetto

Una proprietà di un oggetto può essere ispezionata con un metodo `get` e modificata con un metodo `set`.

Una proprietà è un valore caratteristico di un oggetto che può essere ispezionato e/o modificato dall'utente. Ad esempio, un oggetto `Student` può avere un nome e un identificativo univoco (ID). Aggiungete alla classe una variabile di esemplare per memorizzare il valore della proprietà, con metodi che lo ispezionano e lo modificano.

```
public class Student
{
    private String name;
```

```

    . . .
    public String getName() { return name; }
    public void setName(String newName) { name = newName; }
    . . .
}

```

Spesso nel metodo *set* si aggiunge codice che verifica la validità del nuovo valore. Ad esempio, potremmo voler rifiutare un nuovo nome di lunghezza zero:

```

public void setName(String newName)
{
    if (newName.length() > 0) { name = newName; }
}

```

Alcune proprietà non dovrebbero cambiare dopo aver ricevuto un valore nel costruttore: ad esempio, il valore identificativo (ID) di uno studente potrebbe essere fisso (diversamente dal nome dello studente, che potrebbe anche cambiare). In tal caso, basta non mettere a disposizione il metodo *set*:

```

public class Student
{
    private int id;
    . . .
    public Student(int anId) { id = anId; }
    public int getId() { return id; }
    // non mettiamo il metodo setId
    . . .
}

```

### 8.3.5 Oggetti con stati diversi

Alcuni oggetti hanno un comportamento che varia in relazione a ciò che è accaduto loro in passato. Ad esempio, un oggetto *Fish*, che rappresenti un pesce, potrebbe andare in cerca di cibo quando è affamato, ma ignorarlo dopo aver mangiato. Un tale oggetto deve ricordarsi se ha mangiato di recente.

Usate una variabile di esemplare che rappresenti lo stato dell'oggetto, con costanti che rappresentano i diversi stati possibili:

```

public class Fish
{
    private int hungry;
    . . .
    public static final int NOT_HUNGRY = 0; // non affamato
    public static final int SOMEWHAT_HUNGRY = 1; // un po' affamato
    public static final int VERY_HUNGRY = 2; // molto affamato
    . . .
}

```

Se il comportamento di un oggetto dipende dal fatto che assuma uno stato tra alcuni possibili, usate una variabile di esemplare per memorizzare il suo stato corrente.

In alternativa, si può usare un tipo enumerativo, come visto nella sezione Argomenti avanzati 5.4.

Determinate, poi, quali metodi modificano lo stato. In questo esempio, un pesce che abbia appena mangiato non sarà affamato, ma, se nuota, la sua fame aumenterà:

```
public void eat() // mangia
{
    hungry = NOT_HUNGRY;
    ...
}

public void move() // nuota
{
    ...
    if (hungry < VERY_HUNGRY) { hungry++; }
}
```

Infine, individuate in quali metodi lo stato dell'oggetto ha effetti sul suo comportamento. Quando un pesce molto affamato nuota, il suo primo obiettivo sarà il cibo:

```
public void move() // nuota
{
    if (hungry == VERY_HUNGRY)
    {
        ... // va in cerca di cibo
    }
    ...
}
```

### 8.3.6 Descrizione della posizione di un oggetto

Per progettare un modello  
di un oggetto che si sposta dovette  
memorizzarne la posizione  
e aggiornarla.

Alcuni oggetti si spostano, ricordando la propria posizione. Ad esempio:

- un treno procede lungo i binari tenendo traccia della distanza dall'arrivo;
- un insetto che vive in uno spazio rappresentato da una scacchiera si sposta da un quadro all'altro oppure compie una rotazione di 90 gradi in senso orario o antiorario;
- una palla di cannone sparata verso l'alto scende attirata dalla forza di gravità terrestre.

Questi oggetti hanno bisogno di memorizzare la propria posizione e, dipendentemente dalla natura del loro movimento, può essere necessario memorizzare anche la direzione o la velocità.

Se l'oggetto si sposta lungo una linea, si può rappresentare la sua posizione come la distanza da un punto fisso:

```
private double distanceFromTerminus;
```

Se, invece, l'oggetto si sposta su una scacchiera o una griglia ortogonale, deve ricordare la propria posizione e la direzione del movimento:

```
private int row; // riga
private int column; // colonna
private int direction; // 0=nord, 1=est, 2=sud, 3=ovest
```

Per costruire un modello di un oggetto fisico, come una palla di cannone, spesso è necessario tenere traccia tanto della posizione quanto della velocità, eventualmente in due o tre direzioni. Ecco come si può rappresentare il modello di una palla di cannone che viene sparata verticalmente verso l'alto:

```
private double zPosition;
private double zVelocity;
```

La classe, poi, dovrà avere metodi che aggiornano la posizione. Nel caso più semplice, all'oggetto verrà comunicato l'entità dello spostamento:

```
public void move(double distanceMoved)
{
    distanceFromTerminus = distanceFromTerminus + distanceMoved;
}
```

Se il movimento avviene su una griglia o una scacchiera, bisogna aggiornare la riga o la colonna, in relazione all'orientamento corrente:

```
public void moveOneUnit()
{
    if (direction == NORTH) { row--; }
    else if (direction == EAST) { column++; }
    else if (direction == SOUTH) { row++; }
    else if (direction == WEST) { column--; }
}
```

L'Esercizio P8.10 mostra come si possa aggiornare la posizione di un oggetto fisico quando sia nota la sua velocità.

Nella gestione di un oggetto in movimento, occorre ricordare che il programma simulerà il vero movimento secondo una qualche strategia. Individuate le regole che governano la simulazione, che possono prevedere spostamenti lungo una linea oppure su una griglia con coordinate intere. Tali regole determinano, ovviamente, le modalità di rappresentazione della posizione dell'oggetto. Successivamente identificate i metodi che spostano l'oggetto e aggiornatene la posizione seguendo le regole della simulazione stessa.



## Auto-valutazione

10. Immaginate di voler contare il numero di transazioni di un conto bancario nel periodo coperto da un estratto conto, aggiungendo un contatore alla classe `BankAccount`:

```
public class BankAccount
{
    private int transactionCount;
    ...
}
```

Quali metodi devono aggiornare questo contatore?

11. Nella sezione Consigli pratici 3.1, la classe `CashRegister` non disponeva del metodo `getTotalPurchase`: bisognava invocare i metodi `receivePayment` e `giveChange`. Quali regole,

- tra quelle viste nel Paragrafo 8.2.4, sono violate da tale schema progettuale? Qual è un'alternativa migliore?
12. Perché nell'esempio del Paragrafo 8.3.3 è necessario il metodo `add`? O, per meglio dire, perché l'utilizzatore di un oggetto di tipo `Question` non può semplicemente invocare il metodo `add` della classe `ArrayList<String>`?
  13. Immaginate di voler migliorare la classe `CashRegister` vista nella sezione Consigli pratici 3.1 in modo che memorizzi i prezzi di tutti gli articoli acquistati, per visualizzare lo scontrino completo al termine di un acquisto. Di quale variabile di esemplare avete bisogno? Quali metodi dovrete modificare?
  14. Considerate la classe `Employee`, che rappresenti lavoratori dipendenti e che abbia, come proprietà, lo stipendio e il codice fiscale. Quale di queste proprietà deve essere corredata del solo metodo `get` e quale, invece, deve avere entrambi i metodi, `get` e `set`?
  15. Immaginate che il metodo `setName` visto nel Paragrafo 8.3.4 venga modificato in modo che restituisca `true` se e solo se ha impostato un nuovo nome. È una buona idea?
  16. Osservate la variabile di esemplare `direction` usata nell'esempio dell'insetto, nel Paragrafo 8.3.6. Di quale schema progettuale costituisce un esempio?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi E8.24, E8.25 e E8.26, al termine del capitolo.

## 8.4 Variabili statiche e metodi statici

Una variabile statica appartiene alla classe, non a un oggetto della classe.

A volte occorre memorizzare valori che appartengono più correttamente a una classe che a un suo qualsiasi oggetto: a questo scopo si utilizza una **variabile statica** (*static variable*). Ecco un classico esempio: vogliamo assegnare ai conti bancari numeri identificativi consecutivi, cioè vogliamo che il costruttore del conto crei il primo conto con il numero 1001, il secondo con il numero 1002, e così via. Dobbiamo pertanto disporre di un unico valore, memorizzato nella variabile `lastAssignedNumber`, che sia una proprietà *della classe*, non di un oggetto della classe. Una tale variabile viene detta statica e la si dichiara usando la parola riservata `static`.

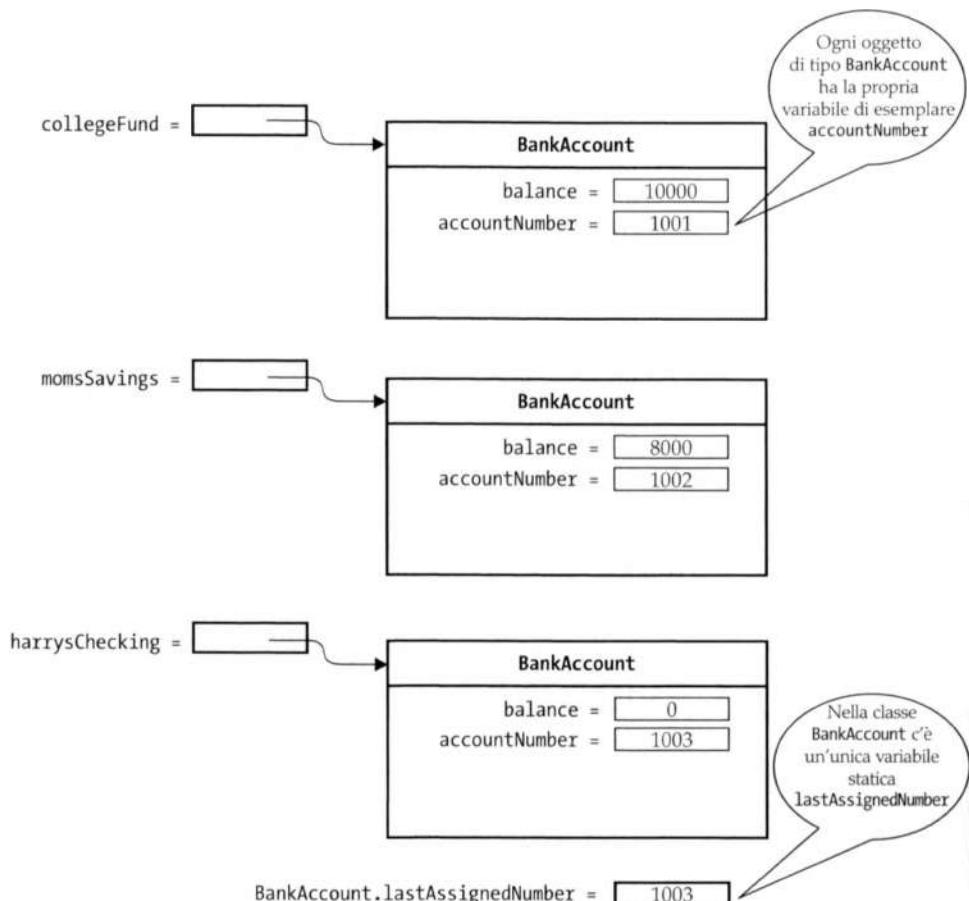
```
public class BankAccount
{
    private double balance;
    private int accountNumber;
    private static int lastAssignedNumber = 1000;

    public BankAccount()
    {
        lastAssignedNumber++;
        accountNumber = lastAssignedNumber;
    }
    ...
}
```

Ciascun oggetto di tipo `BankAccount` ha le proprie variabili di esemplare `balance` e `accountNumber`, ma tutti condividono un'unica copia della variabile `lastAssignedNumber` (osservate la Figura 5), collocata in una porzione di memoria a sé stante, al di fuori di qualsiasi oggetto di tipo `BankAccount`.

**Figura 5**

Variabili di esemplare  
e una variabile statica



Come per le variabili di esemplare, bisogna dichiarare `private` anche le variabili statiche, per essere certi che i metodi di altre classi non ne modifichino il valore. Fanno eccezione a questa regola le *costanti* statiche, che possono essere `private` o pubbliche.

Per esempio, la classe `BankAccount` potrebbe voler definire una costante pubblica, come questa:

```
public class BankAccount
{
    public static final double OVERDRAFT_FEE = 29.95;
    ...
}
```

In qualsiasi metodo di qualunque classe potete fare riferimento a una tale costante scrivendo `BankAccount.OVERDRAFT_FEE`.

Un metodo statico non viene invocato mediante un oggetto.

Talvolta una classe definisce metodi che non vengono invocati con un oggetto come parametro implicito: un metodo di questo tipo viene detto **metodo statico** (*static method*). Un tipico esempio di metodo statico è il metodo `sqrt` della classe `Math`. Dal momento che i numeri non sono oggetti, non potete utilizzarli per invocare metodi. Ad esempio, se `x` fosse un numero, l'invocazione `x.sqrt()` non sarebbe sintatticamente lecita in Java. Quindi, la classe `Math` mette a disposizione un metodo statico, che viene invocato come

`Math.sqrt(x)`. Non viene costruito alcun oggetto di tipo `Math`: l'identificatore `Math` serve soltanto a dire al compilatore dove si trova il metodo `sqrt`.

Potete ovviamente definire vostri metodi statici, da usare in altre classi. Ecco un esempio:

```
public class Financial
{
    /**
     * Calcola la percentuale di una data quantità.
     * @param percentage la percentuale da applicare
     * @param amount la quantità di cui calcolare la percentuale
     * @return la percentuale richiesta di amount
    */
    public static double percentOf(double percentage, double amount)
    {
        return (percentage / 100) * amount;
    }
}
```

Per invocare un metodo statico, si scrive il nome della classe che lo contiene:

```
double tax = Financial.percentOf(taxRate, total);
```

Nella programmazione orientata agli oggetti i metodi statici non sono molto frequenti, anche se il metodo `main` è sempre statico: quando un programma inizia la propria esecuzione non è ancora stato creato alcun oggetto, quindi il primo metodo eseguito da qualsiasi programma deve essere un metodo statico.



## Auto-valutazione

17. Citate due variabili statiche della classe `System`.
18. Citate una costante statica della classe `Math`.
19. Perché questo metodo, che calcola la media dei numeri interi presenti in un array, deve essere statico?  
`public static double average(double[] values)`
20. Il vostro amico Harry vi dice che ha trovato un'ottima strategia per evitare l'utilizzo di quegli antipatici oggetti: mette tutto il codice in un'unica classe e dichiara statici tutti i metodi e tutte le variabili, così il metodo `main` può invocare qualsiasi altro metodo statico e tutti i metodi possono accedere alle variabili statiche. Questa strategia può funzionare? Vi sembra valida?

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi R8.24, E8.6 e E8.7, al termine del capitolo.



## Suggerimenti per la programmazione 8.2

### Minimizzare l'uso di metodi statici

È possibile risolvere qualunque problema di programmazione usando classi che contengano soltanto metodi statici e, prima dell'avvento della programmazione orientata agli oggetti,

questo approccio era il più comune. Si tratta, però, di uno stile di progettazione che porta a soluzioni assai poco orientate agli oggetti, con conseguente difficoltà nell'evoluzione dei programmi nel tempo.

Considerate l'esempio visto nella sezione Consigli pratici 7.1: un programma legge i voti di uno studente e ne visualizza il voto finale, ottenuto sommandoli tutti dopo l'eliminazione del voto minimo. Abbiamo risolto il problema realizzando una classe, `Student`, che memorizza i voti di uno studente, ma, naturalmente, avremmo potuto scrivere semplicemente un programma costituito da pochi metodi statici:

```
public class ScoreAnalyzer
{
    public static double[] readInputs() { . . . }
    public static double sum(double[] values) { . . . }
    public static double minimum(double[] values) { . . . }
    public static double finalScore(double[] values)
    {
        if (values.length == 0) { return 0; }
        else if (values.length == 1) { return values[0]; }
        else { return sum(values) - minimum(values); }
    }

    public static void main(String[] args)
    {
        System.out.println(finalScore(readInputs()));
    }
}
```

Questa soluzione va bene se l'unico obiettivo è quello di risolvere un semplice compito assegnato a scuola. Immaginate, invece, di dover poi modificare il programma in modo che gestisca più studenti: un programma orientato agli oggetti può agevolmente far evolvere la classe `Student` in modo da memorizzare i voti di più studenti, mentre aggiungere funzionalità a metodi statici genera rapidamente confusione (come si può vedere nell'Esercizio E8.8).



## Errori comuni 8.1

### Tentare l'accesso a variabili di esemplare in metodi statici

Un metodo statico non ha un oggetto come parametro implicito, quindi non ha accesso diretto alle variabili di esemplare di un oggetto. Ad esempio, il codice seguente è sbagliato:

```
public class SavingsAccount
{
    private double balance;
    private double interestRate;

    public static double interest(double amount)
    {
        return (interestRate / 100) * amount;
        // ERRORE: metodo statico che accede a una variabile di esemplare
    }
}
```

Dato che conti di risparmio distinti possono avere tassi di interesse diversi, il metodo `interest` non dovrebbe essere statico.

## Argomenti avanzati 8.2

### Forme alternative per inizializzare variabili di esemplare e variabili statiche

Come avete visto, le variabili di esemplare vengono inizializzate a un valore predefinito (`0`, `false` o `null`, in relazione al tipo). In seguito, seguendo lo stile che preferiamo in questo libro, potete assegnare loro qualsiasi valore in un costruttore.

Tuttavia, esistono altri due meccanismi per specificare un valore iniziale per le variabili di esemplare. Proprio come per le variabili locali, potete indicare valori iniziali anche per le variabili di esemplare nella loro dichiarazione, come in questo esempio:

```
public class Coin
{
    private double value = 1;
    private String name = "Dollar";
    ...
}
```

Questi valori saranno utilizzati per *qualsiasi* oggetto che viene costruito.

Esiste anche un'altra sintassi, molto meno comune: potete inserire uno o più *blocchi di inizializzazione* all'interno della dichiarazione della classe e tutti gli enunciati del blocco saranno eseguiti ogni volta che si costruisce un oggetto. Ecco un esempio:

```
public class Coin
{
    private double value;
    private String name;
    {
        value = 1;
        name = "Dollar";
    }
    ...
}
```

Per le variabili statiche si usa, analogamente, un blocco di inizializzazione statico:

```
public class BankAccount
{
    private static int lastAssignedNumber;
    static
    {
        lastAssignedNumber = 1000;
    }
    ...
}
```

Tutti gli enunciati presenti nel blocco di inizializzazione statico vengono eseguiti una sola volta, quando la classe viene caricata. Questi blocchi di inizializzazione, comunque, sono raramente utilizzati nella pratica comune.

Quando un oggetto viene costruito, vengono eseguiti, nell'ordine in cui compaiono, le inizializzazioni di variabili e i blocchi di inizializzazione, seguiti dal codice del costruttore. Dal momento che le regole per i meccanismi alternativi di inizializzazione sono piuttosto complesse, per l'attività di costruzione raccomandiamo di usare semplicemente i costruttori.



## Argomenti avanzati 8.3

### Importazione statica

Esiste una variante della direttiva `import` che consente di utilizzare metodi statici e variabili statiche di una classe senza il relativo prefisso. Ad esempio:

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

public class RootTester
{
    public static void main(String[] args)
    {
        double r = sqrt(PI); // invece di Math.sqrt(Math.PI)
        out.println(x); // invece di System.out
    }
}
```

Le importazioni statiche rendono più agevole la lettura dei programmi, soprattutto quando si utilizzano molte funzioni matematiche.

## 8.5 Problem Solving: iniziare da un problema più semplice

**Quando si sviluppa la soluzione di un problema, è bene iniziare da un problema semplificato.**

Con l'aumentare delle vostre conoscenze di programmazione vi sarà chiesto di risolvere problemi di complessità sempre maggiore. Per affrontare un problema complesso, dovete usare un'abilità fondamentale: semplificare il problema e, per prima cosa, risolvere la forma semplificata del problema in esame.

Si tratta di una buona strategia per molteplici motivi. Innanzitutto, di solito risolvendo il problema semplificato si impara qualcosa di utile. Inoltre, il problema nella sua interezza potrebbe sembrare insormontabile e può essere difficile trovare il modo di iniziare a risolverlo. Invece, dopo aver avuto successo nella soluzione del problema semplificato, sarete molto più motivati nel cercare di risolvere quello più complesso.

Per riuscire a scomporre un problema in una sequenza di problemi più semplici serve un po' di esperienza e il modo migliore per imparare a farlo è provare. Nello svolgimento del prossimo esercizio che vi verrà assegnato, chiedetevi quale sia la parte più semplice del problema che sia utile per la soluzione finale e partite da lì. Dopo aver acquisito un po' di esperienza sarete in grado di progettare un piano di lavoro che costruisca la soluzione completa sfruttando una serie di fasi intermedie.

Vediamo un esempio. Il problema consiste nell'individuare una disposizione di immagini che le veda allineate lungo il loro bordo superiore, separate da una piccola spaziatura e disposte in una sequenza di righe orizzontali, iniziando una nuova riga quando l'immagine successiva non trova posto nella riga corrente.



Per risolvere il problema disponiamo della classe `Picture` dotata di un costruttore

```
public Picture(String filename)
```

e dei metodi:

```
public void move(int dx, int dy)
public Rectangle getBounds()
```

Delineate un piano costituito da una serie di problemi, ciascuno dei quali sia un'estensione abbastanza semplice del precedente, terminando con il problema originario.

Invece di affrontare l'intero problema, delineiamo un piano di lavoro che preveda la soluzione di una serie di problemi più semplici:

1. Disegnare una sola immagine.



2. Disegnare due immagini affiancate.



3. Disegnare due immagini affiancate con un piccolo spazio interposto.



4. Disegnare tutte le immagini in una sola lunga riga.



5. Disegnare una riga di immagini fino all'esaurimento dello spazio disponibile, poi disegnare un'ulteriore immagine all'inizio della riga successiva.



Iniziamo a seguire questo piano di lavoro.

1. Lo scopo di questo primo passo è quello di prendere confidenza con la classe `Picture`. Ipotizziamo che le immagini siano memorizzate nei file `picture1.jpg`, ..., `picture20.jpg`, quindi carichiamo nel programma la prima immagine:

```
public class Gallery1
{
    public static void main(String[] args)
    {
        Picture pic = new Picture("picture1.jpg");
    }
}
```

Tutto ciò è sufficiente per visualizzare una prima immagine.

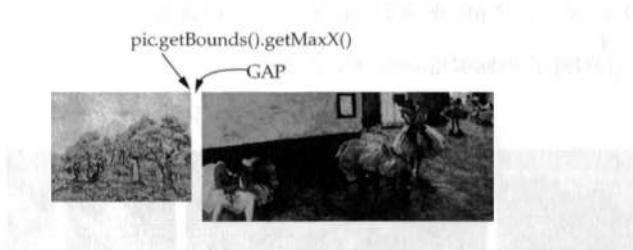


2. Adesso affianchiamo alla prima immagine la successiva. Per farlo dobbiamo spostarci nella posizione corrispondente alla coordinata `x` più a destra che caratterizza, appunto, la prima immagine.



```
Picture pic = new Picture("picture1.jpg");
Picture pic2 = new Picture("picture2.jpg");
pic2.move(pic.getBounds().getMaxX(), 0);
```

3. Nel passo successivo separiamo le due immagini aggiungendo una piccola spaziatura durante lo spostamento verso destra della seconda immagine



```
final int GAP = 10; // spaziatura
```

```
Picture pic = new Picture("picture1.jpg");
Picture pic2 = new Picture("picture2.jpg");
double x = pic.getBounds().getMaxX() + GAP;
pic2.move(x, 0);
```

4. Ora visualizziamo tutte le figure usando un'unica riga. Leggiamo le figure all'interno di un ciclo, disponendo ciascuna di esse alla destra della precedente. Nel ciclo abbiamo bisogno di riferirci a due figure: quella che stiamo per visualizzare e quella precedente (si veda il Paragrafo 6.7.6).



```
final int GAP = 10; // spaziatura
final int PICTURES = 20;

Picture pic = new Picture("picture1.jpg");
for (int i = 2; i <= PICTURES; i++)
{
    Picture previous = pic;
    pic = new Picture("picture" + i + ".jpg");
    double x = previous.getBounds().getMaxX() + GAP;
    pic.move(x, 0);
}
```

5. Però non vogliamo che tutte le immagini vengano visualizzate su un'unica riga, perché il margine destro di un'immagine non deve trovarsi alla destra di MAX\_WIDTH.

```

double x = previous.getBounds().getMaxX() + GAP;
if (x + pic.getBounds().getWidth() < MAX_WIDTH)
{
    . . . // pic si aggiunge alla riga attuale
}
else
{
    . . . // pic deve iniziare una riga nuova
}

```



Se l'immagine non trova posto nella riga attuale, la dobbiamo visualizzare in una riga nuova, al di sotto di tutte le immagini della riga attuale. Assegniamo alla variabile `maxY` il valore massimo tra le coordinate `y` di tutte le immagini già visualizzate nella riga, aggiornandola ogni volta che si aggiunge una nuova immagine:

```
maxY = Math.max(maxY, pic.getBounds().getMaxY());
```

L'enunciato seguente posiziona un'immagine all'inizio della riga successiva:

```
pic.move(0, maxY + GAP);
```

A questo punto abbiamo scritto codice che risolve tutte le situazioni semplificate del problema: sappiamo come allineare immagini, come separarle con una piccola spaziatura, come capire quando si deve iniziare una nuova riga e dove farla partire.

Sapendo questo, non è difficile progettare la soluzione finale: ecco il programma.

### File **Gallery6.java**

```

1 public class Gallery6
2 {
3     public static void main(String[] args)
4     {
5         final int MAX_WIDTH = 720;
6         final int GAP = 10;
7         final int PICTURES = 20;
8
9         Picture pic = new Picture("picture1.jpg");
10        double maxY = 0;

```

```

11
12     for (int i = 2; i <= PICTURES; i++)
13     {
14         maxY = Math.max(maxY, pic.getBounds().getMaxY());
15         Picture previous = pic;
16         pic = new Picture("picture" + i + ".jpg");
17         double x = previous.getBounds().getMaxX() + GAP;
18         if (x + pic.getBounds().getWidth() < MAX_WIDTH)
19         {
20             pic.move(x, previous.getBounds().getY());
21         }
22         else
23         {
24             pic.move(0, maxY + GAP);
25         }
26     }
27 }
28 }
```



## Auto-valutazione

21. Immaginate di dover trovare in un elenco di parole tutte quelle che non contengono lettere ripetute. Da quale semplice problema è bene iniziare?
22. Dovete scrivere un programma di analisi del DNA che verifichi se una sottostringa di una stringa è contenuta in un'altra stringa. Quale semplice problema dovreste risolvere per primo?
23. Dovete eliminare gli “occhi rossi” da un’immagine, cercando cerchi di colore rosso. Da quale semplice problema inizierete?
24. Considerate il problema di individuare numeri interi all’interno di una stringa. Ad esempio, nella stringa seguente sono presenti tre numeri: “In 1987, a typical personal computer cost \$3000 and had 512 kilobytes of RAM”. Scomponete questo problema in una sequenza di problemi più semplici.

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi R8.26 e P8.5, al termine del capitolo.

## 8.6 Pacchetti

Un pacchetto (*package*)  
è un insieme di classi correlate.

Un programma Java è, in generale, costituito da più classi. Fino ad ora la maggior parte dei vostri programmi erano composti da un piccolo numero di classi, ma, quando i programmi aumentano di dimensione, limitarsi a distribuire le classi in più file non basta e c’è bisogno di un meccanismo aggiuntivo per strutturare il codice.

In Java, i pacchetti rappresentano questo meccanismo strutturale: un **pacchetto** (*package*) è un insieme di classi correlate. Per esempio, la libreria di Java è composta da parecchie centinaia di pacchetti, alcuni dei quali sono elencati nella Tabella 1.

**Tabella 1**

Pacchetti principali della libreria di Java

Pacchetto	Scopo	Esempio di classi
java.lang	Supporto al linguaggio	Math
java.util	Varie utilità	Random
java.io	Input e output	PrintStream
java.awt	Abstract Windowing Toolkit	Color
java.applet	Applet	Applet
java.net	Connessione di rete	Socket
java.sql	Accesso a database tramite Structured Query Language	ResultSet
javax.swing	Interfaccia utente Swing	JButton
org.w3c.dom	Document Object Model (DOM) per documenti XML	Document

## 8.6.1 Organizzare classi in pacchetti

Per inserire una delle vostre classi in un pacchetto dovete scrivere questa riga all'inizio del file sorgente che contiene la classe:

```
package nomePacchetto;
```

Il nome di un pacchetto è formato da uno o più identificatori, separati da punti (nel Paragrafo 8.6.3 troverete alcuni suggerimenti per la composizione dei nomi di pacchetto).

Come esempio, inseriamo all'interno di un pacchetto chiamato `com.horstmann.bigjava` la classe `Financial` che abbiamo introdotto in questo capitolo. Il file `Financial.java` deve iniziare in questo modo:

```
package com.horstmann.bigjava;
public class Financial
{
    . .
}
```

Oltre ai pacchetti che hanno un nome, come `java.util` o `com.horstmann.bigjava`, esiste un pacchetto speciale, chiamato *pacchetto predefinito (default package)*, che è senza nome: se non inserite un enunciato `package` all'inizio di un file sorgente, le classi in esso dichiarate verranno inserite nel pacchetto predefinito.

## 8.6.2 Importare pacchetti

Se volete usare una classe di un pacchetto, potete specificarla nel codice tramite il suo nome completo, costituito dal nome del pacchetto seguito da quello della classe. Per esempio, `java.util.Scanner` si riferisce alla classe `Scanner` nel pacchetto `java.util`:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

La direttiva `import` permette di utilizzare una classe di un pacchetto usando soltanto il nome della classe, senza il prefisso del pacchetto.

Questo modo di scrivere il codice è molto scomodo. In alternativa, potete *importare* un nome mediante un enunciato `import`:

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

dopodiché potete riferirvi alla classe `Scanner` senza scrivere il prefisso che indica il pacchetto.

Potete anche importare *tutte le classi* di un pacchetto, mediante un enunciato `import` che termini con `.*`. Per esempio, potete usare l'enunciato seguente per importare tutte le classi del pacchetto `java.util`:

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

Questo enunciato consente di fare riferimento a classi come `Scanner` o `Random` senza usare il prefisso `java.util`.

Non avrete, però, mai bisogno di importare esplicitamente le classi del pacchetto `java.lang`: questo pacchetto contiene le classi di Java più basilari, come `Math` e `Object`, che sono sempre disponibili. In pratica, in ciascun file sorgente viene considerata implicita la presenza dell'enunciato `import java.lang.*;`.

Infine, non c'è bisogno di importare altre classi presenti nel pacchetto in cui si trova la classe che state scrivendo. Ad esempio, quando realizzate la classe `homework1.Tester`, non avete bisogno di importare la classe `homework1.Bank`: il compilatore troverà la classe senza bisogno di un enunciato di importazione perché si trova nello stesso pacchetto, `homework1`.

### 8.6.3 Nomi di pacchetto

Collocare in un pacchetto le classi tra loro correlate è chiaramente un comodo meccanismo per organizzarle, ma c'è una ragione più importante che giustifica l'utilizzo dei pacchetti: evitare il **conflitto di nomi**. In un progetto di grandi dimensioni è inevitabile che due persone individuino lo stesso nome per il medesimo concetto. Succede perfino nella libreria delle classi standard di Java (che attualmente è cresciuta fino a comprendere migliaia di classi): c'è una classe `Timer` nel pacchetto `java.util` e un'altra classe, sempre chiamata `Timer`, nel pacchetto `javax.swing`. Potete sempre indicare esattamente al compilatore Java quale classe `Timer` vi serve, semplicemente riferendovi alla classe mediante il suo nome completo, `java.util.Timer` o `javax.swing.Timer`.

Naturalmente, perché la convenzione per i nomi dei pacchetti funzioni, deve esistere qualche sistema per garantire che tali nomi di pacchetto siano univoci. Sarebbe un bel problema se il costruttore di auto BMW inserisse tutto il suo codice Java nel pacchetto

## Sintassi di Java

### 8.1 Specifica di pacchetto

#### Sintassi

```
package nomePacchetto;
```

#### Esempio

```
package com.horstmann.bigjava;
```

Le classi di questo file appartengono  
a questo pacchetto.

Un nome di dominio con le componenti scritte  
al contrario è una buona scelta come nome  
di pacchetto.

`bmw` e qualche altro programmatore (magari Britney M. Walters) avesse la stessa brillante idea. Per evitare questo, gli inventori di Java raccomandano di utilizzare uno schema per l'assegnazione dei nomi ai pacchetti che sfrutta l'univocità dei nomi di dominio in Internet.

Per costruire nomi di pacchetti non ambigui usate un nome di dominio, invertendo l'ordine delle stringhe che lo compongono.

Per esempio, io posseggo il dominio `horstmann.com` e nessun altro nel mondo ha un dominio con lo stesso nome (ho avuto la fortuna che nessun altro avesse già acquisito il nome `horstmann.com`, quando l'ho richiesto; se vi chiamate Walters, scoprirete tristemente che qualcun altro vi ha battuto nella registrazione del dominio `walters.com`). Per ottenere un nome di pacchetto, invertite l'ordine delle stringhe che compongono il nome del dominio, creando un prefisso per i nomi dei pacchetti, nel mio caso `com.horstmann`.

Se non possedete un vostro nome di dominio, potete sempre creare un nome di pacchetto che avrà elevata probabilità di essere univoco scrivendo il vostro indirizzo di posta elettronica al contrario. Per esempio, se l'indirizzo di posta elettronica di Britney Walters fosse `walters@cs.sjsu.edu`, potrebbe usare il nome `edu.sjsu.cs.walters` per i pacchetti delle sue classi.

Alcuni docenti vorranno che mettiate ciascuna delle vostre esercitazioni in un pacchetto separato, come `homework1`, `homework2`, e così via. La motivazione sta di nuovo nell'evitare conflitti fra nomi: potrete così avere due classi, `homework1.Bank` e `homework2.Bank`, con proprietà un po' diverse.

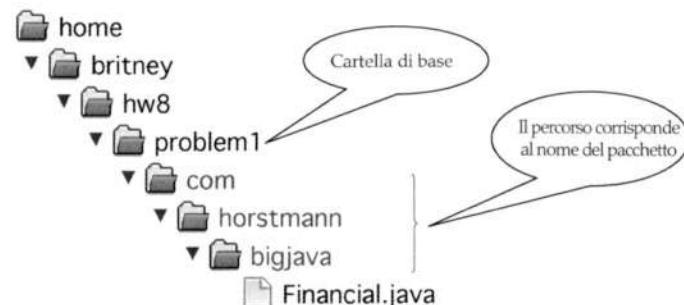
## 8.6.4 Pacchetti e file di codice sorgente

Il percorso del file che contiene una classe deve corrispondere al suo nome di pacchetto.

Un file sorgente, contenente una classe, deve trovarsi in una cartella il cui nome corrisponde a quello del pacchetto a cui appartiene la classe. Le parti del nome del pacchetto, separate da punti, rappresentano le cartelle annidate, una dentro l'altra. Per esempio, le classi del pacchetto `com.horstmann.bigjava` devono essere nella cartella `com/horstmann/bigjava` all'interno della *cartella di base* ("base directory") in cui lavorate. Se, ad esempio, la vostra cartella di base è `/home/problem1`, potete collocare i file delle classi del pacchetto `com.horstmann.bigjava` nella cartella `/home/problem1/com/horstmann/bigjava`, come mostrato nella Figura 6 (stiamo usando nomi di file secondo lo stile di UNIX; in Windows, usereste `c:\home\problem1\com\horstmann\bigjava`).

**Figura 6**

Cartella di base e cartelle per i pacchetti





## Auto-valutazione

25. Quali di questi sono pacchetti?
  - a. java
  - b. java.lang
  - c. java.util
  - d. java.lang.Math
26. È vero che un programma Java privo di enunciati `import` è vincolato a usare soltanto classi appartenenti al pacchetto predefinito o al pacchetto `java.lang`?
27. Se per i vostri programmi usate la cartella di base `/home/me/cs101` (probabilmente `c:\Users\Me\cs101` in Windows) e il vostro docente vi chiede di inserire in pacchetti le vostre soluzioni degli esercizi, in quale cartella metterete la classe `hw1.problem1.TicTacToeTester`?

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi R8.28, E8.18 e E8.19, al termine del capitolo.



## Errori comuni 8.2

### Fare confusione con i punti

In Java, il simbolo costituito dal carattere punto (.) si usa come separatore nei casi seguenti:

- All'interno di nomi di pacchetti (`java.util`)
- Fra un nome di pacchetto e un nome di classe (`homework1.Bank`)
- Fra un nome di classe e un nome di classe interna (`Ellipse2D.Double`)
- Fra un nome di classe e un nome di variabile statica (`Math.PI`)
- Fra un oggetto e un metodo (`account.getBalance()`)

Quando vedete una lunga catena di nomi separati da punti, diventa arduo scoprire quale parte costituisce il nome di un pacchetto, di una classe, di una variabile o di un metodo. Considerate questo esempio:

```
java.lang.System.out.println(x);
```

Dal momento che `println` è seguito da una parentesi aperta, deve essere il nome di un metodo. Pertanto, `out` può essere una classe che abbia un metodo statico `println` oppure un oggetto (naturalmente, noi sappiamo già che `out` è un riferimento a un oggetto di tipo `PrintStream`). Inoltre, non è del tutto chiaro, al di fuori del contesto, se `System` sia un altro oggetto, con una variabile pubblica `out`, oppure una classe con una variabile statica di nome `out`. A giudicare dal numero di pagine che il manuale di riferimento del linguaggio Java dedica a questo argomento, perfino il compilatore ha qualche difficoltà a interpretare queste sequenze di stringhe separate da punti.

Per evitare problemi, è utile adottare uno stile di scrittura rigoroso per il codice Java. Se i nomi di classe iniziano sempre con una lettera maiuscola, mentre i nomi delle variabili, dei metodi e dei pacchetti iniziano sempre con una lettera minuscola, si può evitare di fare confusione.



## Argomenti avanzati 8.4

Variabili e metodi che non vengono dichiarati né `public` né `private` sono disponibili per tutte le classi dello stesso pacchetto, situazione solitamente indesiderata.

### Accesso di pacchetto

Se nella dichiarazione di una classe, di una variabile o di un metodo non viene specificata la modalità di accesso, né `public` né `private`, allora tutti i metodi di tutte le classi che appartengono allo stesso pacchetto hanno diritto di accesso a tale caratteristica. Ad esempio, se una classe è dichiarata `public`, tutte le altre classi, appartenenti a qualunque pacchetto, la possono usare. Se, invece, una classe viene dichiarata senza specificare alcuna modalità di accesso, può essere utilizzata soltanto dalle altre classi *del pacchetto a cui appartiene*. L'accesso di pacchetto è ragionevole per le classi, ma è estremamente inadeguato per le variabili di esemplare.

È assai comune, e sbagliato, *dimenticare* la parola riservata `private`, aprendo così una potenziale falla nella sicurezza del progetto. Ad esempio, a tutt'oggi la classe `Window` del pacchetto `java.awt` contiene questa dichiarazione:

```
public class Window extends Container
{
    String warningString;
    ...
}
```

Non c'è veramente alcun valido motivo per consentire l'accesso di pacchetto alla variabile di esemplare `warningString`: infatti, nessun'altra classe la usa.

L'accesso di pacchetto alle variabili di esemplare è raramente utile e costituisce sempre un potenziale rischio per la sicurezza. Nella maggior parte dei casi l'accesso di pacchetto viene concesso alle variabili di esemplare per semplice dimenticanza della parola `private`. È bene prendere l'abitudine di rivedere le dichiarazioni delle variabili di esemplare, controllando che `private` sia sempre presente.



## Consigli pratici 8.1

### Programmare con i pacchetti

Questi Consigli pratici spiegano in dettaglio come inserire i vostri programmi in pacchetti.

**Problema.** Inserire ciascuna esercitazione in un pacchetto separato: in tal modo, in pacchetti distinti si possono avere classi con lo stesso nome ma con implementazioni diverse (come `homework1.problem1.Bank` e `homework1.problem2.Bank`).

#### Fase 1 Scegliete un nome per il pacchetto

Se il vostro docente non vi assegna un nome di pacchetto da usare obbligatoriamente, come `homework1.problem2`, usate un nome di pacchetto che vi identifichi in modo univoco. Iniziate con il vostro indirizzo di posta elettronica scritto al contrario: ad esempio, `walters@cs.sjsu.edu` diventa `edu.sjsu.cs.walters`; poi, aggiungete un sotto-pacchetto, come `edu.sjsu.cs.walters.cs1project`, che descriva il progetto o l'esercitazione.

**Fase 2** Scegliete una *cartella di base*

La cartella di base è la cartella che contiene le cartelle per i vari pacchetti, ad esempio /home/britney oppure c:\Users\Britney.

**Fase 3** Create nella cartella di base una sottocartella che corrisponda al nome del pacchetto

La sottocartella deve essere contenuta nella vostra cartella di base e ciascun segmento del nome della sottocartella deve corrispondere a un segmento del nome del pacchetto. Ad esempio

mkdir -p /home/britney/homework1/problem2 (in Unix)

oppure

mkdir /s c:\Users\Britney\homework1\problem2 (in Windows)

**Fase 4** Copiate i vostri file sorgente all'interno della sottocartella relativa al pacchetto

Ad esempio, se l'esercitazione comprende i file Tester.java e Bank.java, inseriteli in queste posizioni

/home/britney/homework1/problem2/Tester.java  
/home/britney/homework1/problem2/Bank.java

oppure

c:\Users\Britney\homework1\problem2\Tester.java  
c:\Users\Britney\homework1\problem2\Bank.java

**Fase 5** Usate l'enunciato package in ogni file sorgente

La prima riga di ciascun file, preceduta soltanto da eventuali commenti, deve essere un enunciato package che indichi il nome del pacchetto, come

package homework1.problem2;

**Fase 6** Compilate i vostri file sorgente *a partire dalla cartella di base*

Ponetevi nella cartella di base (scelta nella Fase 2) e compilate i vostri file. Ad esempio

cd /home/britney  
javac homework1/problem2/Tester.java

oppure

c:  
cd \Users\Britney  
javac homework1\problem2\Tester.java

Notate che il compilatore Java ha bisogno del *nome del file sorgente, non del nome della classe*: dovete, quindi, fornire i separatori di percorso (/ in UNIX e \ in Windows) e l'estensione nel nome del file (.java).

**Fase 7** Eseguite il vostro programma *dalla cartella di base*

Diversamente dal compilatore, l'interprete Java ha bisogno del *nome della classe* che contiene il metodo `main` (e *non del nome di un file*): usate, quindi, i punti come separatori nel nome di pacchetto, e non usate l'estensione del nome del file. Ad esempio

```
cd /home/britney
java homework1.problem2.Tester
```

oppure

```
c:
cd \Users\Britney
java homework1.problem2.Tester
```

## 8.7 Ambienti per il collaudo di unità

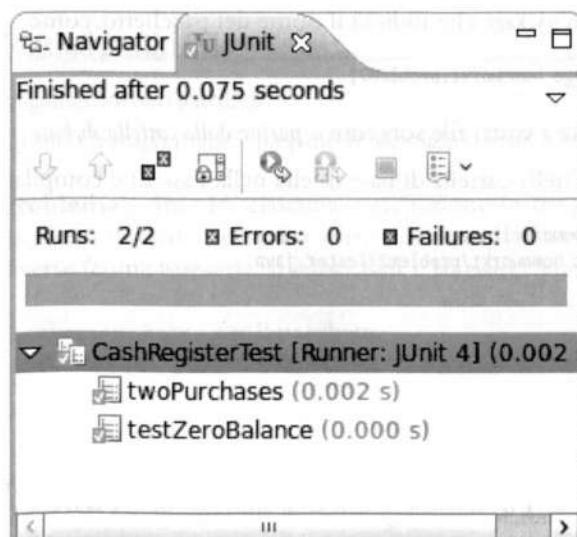
Fino ad ora abbiamo seguito un approccio al collaudo estremamente semplice: progettiamo classi di collaudo il cui metodo `main` calcola valori e visualizza, a coppie, i valori calcolati e quelli previsti. Questa strategia ha due principali limiti: innanzitutto, se il metodo `main` effettua molte verifiche, il suo codice diventa complesso e confuso; poi, se durante una delle prove si verifica un'eccezione, le prove seguenti non vengono eseguite.

Gli ambienti per il collaudo di unità (*unit testing framework*) sono stati progettati proprio per eseguire velocemente pacchetti di prove e valutarne il risultato; inoltre, rendono più semplice l'aggiunta di casi di prova a *test suite* esistenti. Uno degli ambienti di collaudo più utilizzati è JUnit, che è gratuitamente disponibile all'indirizzo <http://junit.org> e che fa anche parte di molti ambienti di sviluppo più completi, come BlueJ e Eclipse. Parliamo qui di JUnit 4, la versione più recente disponibile in questo momento.

Gli ambienti per il collaudo di unità semplificano il compito di scrivere classi che contengano molti casi di prova.

**Figura 7**

Collaudo di unità con JUnit



Lavorando con JUnit, per ogni classe che viene sviluppata si progetta una classe ausiliaria dedicata al collaudo, con un metodo per ogni caso di prova che si vuole eseguire, contrassegnato da una speciale “annotazione”: una caratteristica avanzata di Java che introduce nel codice un simbolo che viene interpretato da strumenti diversi dal compilatore. Nel caso di JUnit i metodi di collaudo vengono contrassegnati con `@Test`.

All'interno di ciascun caso di prova vengono svolti calcoli e si valutano condizioni che si ritengono vere. Si trasferisce, poi, il risultato a un metodo che comunica l'esito del collaudo all'ambiente, generalmente il metodo `assertEquals`, che richiede come parametri i valori previsti e i valori calcolati, oltre a una tolleranza ritenuta accettabile nel caso di valori in virgola mobile.

C'è anche la consuetudine (non necessaria) che il nome della classe di collaudo termini con `Test`, come `CashRegisterTest`. Esamineate questo esempio tipico:

```
import org.junit.Test;
import org.junit.Assert;

public class CashRegisterTest
{
    @Test public void twoPurchases()
    {
        CashRegister register = new CashRegister();
        register.recordPurchase(0.75);
        register.recordPurchase(1.50);
        register.receivePayment(2, 0, 5, 0, 0);
        double expected = 0.25;
        Assert.assertEquals(expected, register.giveChange(), EPSILON);
    }
    // ulteriori casi di prova
    ...
}
```

Se tutti i casi di prova hanno esito positivo, il programma JUnit visualizza una barra verde, mentre mostra una barra rossa e un messaggio d'errore se qualche prova non va a buon fine.

La classe di collaudo può anche avere altri metodi (i cui nomi non dovrebbero essere annotati con `@Test`), tipicamente per compiere azioni ed eseguire calcoli che volete che siano condivisi da più metodi di collaudo.

La filosofia di JUnit è semplice: quando progettate una classe, scrivete anche la relativa classe ausiliaria per il collaudo; progettate i casi di prova mentre procedete con lo sviluppo del programma, un metodo per volta, semplicemente aggiungendoli alla classe di collaudo; ogni volta che correggete un malfunzionamento, aggiungete un caso di prova che lo stimoli, in modo da essere certi che l'errore non venga in qualche modo reintrodotto nel programma; infine, ogni volta che modificate la classe eseguite nuovamente tutti i collaudi inseriti nella classe ausiliaria.

Se tutti i collaudi vanno a buon fine l'interfaccia grafica visualizza una barra verde e vi potete rilassare. In caso contrario vedrete una barra rossa, ma anche questa è utile: è molto più semplice correggere un errore in una classe isolata, piuttosto che all'interno di un programma complesso.

La filosofia di JUnit prevede l'esecuzione di tutti i collaudi ogni volta che viene modificato il codice.



## Auto-valutazione

28. Scrivete con JUnit una classe di prova con un caso di prova per la classe Earthquake vista nel Capitolo 5.
29. Che significato ha il parametro EPSILON nel metodo `assertEquals`?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R8.30, E8.20 e E8.21, al termine del capitolo.



## Computer e società 8.1

### Lo sviluppo dei personal computer

Nel 1971, Marcian E. "Ted" Hoff, un ingegnere di Intel Corporation, stava lavorando a un chip per conto di un costruttore di calcolatrici elettroniche. Si accorse che, anziché creare un altro progetto su misura, sarebbe stato meglio sviluppare un chip per *uso generale*, che si sarebbe potuto *programmare* per fare in modo che si connettesse ai tasti e allo schermo di una calcolatrice: nacque così il *microprocessore*. A quel tempo la sua applicazione principale fu quella di unità di controllo per calcolatrici tascabili, lavatrici e oggetti simili: occorsero anni perché l'industria informatica si accorgesse che un'autentica CPU era disponibile nella forma di un singolo chip.

Coloro che si occupavano di elettronica e informatica a livello amatoriale furono i primi a scoprirla. Nel 1974, MITS Electronics mise in vendita il primo computer in *scatola di montaggio (kit)*, Altair 8800, al costo di 350 dollari circa. Il kit era composto dal microprocessore, da una scheda elettronica, da una quantità molto ridotta di memoria, da interruttori a levetta e da una fila di spie luminose. Gli acquirenti dovevano comprarlo e montarlo, per poi programmarlo in

linguaggio macchina mediante gli interruttori: non fu un grande successo.

Il primo grosso successo fu invece Apple II: un vero computer, con una tastiera, uno schermo e un'unità per dischetti. Quando fu messo in commercio gli utenti ebbero una macchina da 3000 dollari con cui poter giocare a "Space Invaders" ed eseguire un rudimentale programma di contabilità, oltre a programmare in linguaggio BASIC. Il primo Apple II non aveva ancora le lettere minuscole e, quindi, era inutilizzabile per l'elaborazione di testi. La svolta arrivò nel 1979, con un nuovo programma con funzioni di foglio elettronico, VisiCalc. In un foglio elettronico potete inserire dati contabili, con le loro relazioni, in una griglia formata da righe e da colonne; potete, poi, modificare alcuni dati e osservare, in tempo reale, come si modificano gli altri. Per esempio, potete osservare come, variando la composizione di manufatti prodotti da uno stabilimento, si possa influire su costi e profitti stimati. I dirigenti d'azienda si affrettarono a comprare VisiCalc e il computer che serviva per eseguirlo: per loro, il computer era una macchina per il foglio elettronico.

Di più: il computer era diventato un dispositivo personale. I dirigenti

erano finalmente liberi di fare i calcoli che volevano, senza limitarsi a quelli forniti dai "sommi sacerdoti" del centro di elaborazione dati.

Da quel momento possiamo disporre di "computer personali" (cioè *personal computer*, o PC) e una quantità innumerevole di utenti ha armeggiato con hardware e software, a volte fondando società di enorme successo, altre volte creando software gratuito per milioni di utilizzatori. Questa "libertà di armeggiare" è un aspetto molto importante dell'elaborazione personale: su un dispositivo personale potete installare il software che vi interessa, in modo da esaltare la vostra produttività o creatività, anche se non si tratta dello stesso software utilizzato dalla maggior parte delle persone. Allo stesso modo, potete aggiungere al computer i dispositivi periferici che desiderate. Durante i primi trent'anni di diffusione dei PC si è ritenuto che tutta questa libertà dovesse necessariamente esistere e che non ci fossero alternative.

Oggi stiamo entrando in un'epoca in cui *smartphone*, *tablet* e *smart TV* stanno sostituendo i tradizionali personal computer. Nonostante sia stupefacente pensare che il vostro telefono cellulare abbia una potenza

di elaborazione superiore a quella dei migliori PC degli Anni Novanta, è altrettanto spiacevole rendersi conto che abbiamo perduto, in certa misura, il controllo personale del dispositivo. In telefoni e tablet di alcuni marchi potete installare soltanto quelle applicazioni che vengono pubblicate dal produttore sul suo *app store* (“negozi di applicazioni”). Ad esempio, Apple non consente ai

bambini di apprendere usando un iPad il linguaggio Scratch sviluppato dal MIT, perché contiene una macchina virtuale. Spesso si ritiene che Apple dovrebbe aver interesse a stimolare nelle nuove generazioni l'entusiasmo verso il mondo della programmazione: al contrario, quell'azienda manifesta, in generale, una strategia di proibizione nei confronti della programmabilità dei

“suoi” dispositivi, per mettere fuori gioco altri ambienti competitivi, come Flash o Java.

Quando scegliete un dispositivo per telefonare o per guardare un film, dovreste chiedervi chi ne ha il controllo. State acquistando un dispositivo personale utilizzabile in qualsiasi modo vogliate oppure vi state legando a un flusso di dati controllato da qualcun altro?

## Riepilogo degli obiettivi di apprendimento

### Identificare le classi più idonee a risolvere un problema di programmazione

- Una classe dovrebbe rappresentare un singolo concetto nel dominio in cui è descritto il problema: le scienze, la matematica o l'economia.

### Obiettivi: coesione, coerenza e pochi effetti collaterali

- L'interfaccia pubblica di una classe ha una buona coesione se tutte le sue caratteristiche sono correlate al concetto rappresentato dalla classe.
- Una classe dipende da un'altra se i suoi metodi la usano in qualche modo.
- Una classe immutabile non ha metodi modificatori.
- I riferimenti a oggetti di una classe immutabile possono essere condivisi in sicurezza.
- Un effetto collaterale di un metodo è una modifica apportata ai dati che sia osservabile all'esterno del metodo.
- Quando si progettano metodi è bene minimizzarne gli effetti collaterali.
- In Java, un metodo non può mai modificare il contenuto di una variabile che gli viene passata come argomento.
- In Java, un metodo può modificare lo stato di un oggetto di cui ha ricevuto un riferimento, ma non può sostituire il riferimento all'oggetto con un altro riferimento.

### Schemi di progettazione per la rappresentazione dei dati di un oggetto

- Una variabile di esemplare che conserva un totale viene aggiornata nei metodi che lo incrementano o lo decrementano.
- Un contatore di eventi viene incrementato nei metodi che corrispondono a tali eventi.
- Un oggetto può memorizzare una raccolta di altri oggetti in un array o in un vettore.
- Una proprietà di un oggetto può essere ispezionata con un metodo *get* e modificata con un metodo *set*.
- Se il comportamento di un oggetto dipende dal fatto che assuma uno stato tra alcuni possibili, usate una variabile di esemplare per memorizzare il suo stato corrente.
- Per progettare un modello di un oggetto che si sposta dovete memorizzarne la posizione e aggiornarla.

### Metodi statici e variabili statiche

- Una variabile statica appartiene alla classe, non a un oggetto della classe.
- Un metodo statico non viene invocato mediante un oggetto.

### **Progettare programmi che svolgono compiti complessi**

- Quando si sviluppa la soluzione di un problema, è bene iniziare da un problema semplificato.
- Delineate un piano costituito da una serie di problemi, ciascuno dei quali sia un'estensione abbastanza semplice del precedente, terminando con il problema originario.

### **Usare i pacchetti per organizzare insiemi di classi correlate**

- Un pacchetto (*package*) è un insieme di classi correlate.
- La direttiva `import` permette di utilizzare una classe di un pacchetto usando soltanto il nome della classe, senza il prefisso del pacchetto.
- Per costruire nomi di pacchetti non ambigui usate un nome di dominio, invertendo l'ordine delle stringhe che lo compongono.
- Il percorso del file che contiene una classe deve corrispondere al suo nome di pacchetto.
- Variabili e metodi che non vengono dichiarati né `public` né `private` sono disponibili per tutte le classi dello stesso pacchetto, situazione solitamente indesiderata.

### **Usare JUnit per scrivere collaudi di unità**

- Gli ambienti per il collaudo di unità semplificano il compito di scrivere classi che contengano molti casi di prova.
- La filosofia di JUnit prevede l'esecuzione di tutti i collaudi ogni volta che viene modificato il codice.

## **Esercizi di riepilogo e approfondimento**

- ★ **R8.1.** Analizzate un sistema di *car sharing* nel quale i guidatori di automobile possono accogliere passeggeri a bordo, guadagnando denaro durante i propri spostamenti e riducendo il traffico. I passeggeri attendono in appositi punti di prelevamento (*pickup point*), vengono scaricati alle loro destinazioni e pagano per la distanza percorsa, mentre i guidatori vengono pagati una volta al mese. Una *app* consente a guidatori e passeggeri di inserire i dati relativi al proprio percorso e ai tempi coinvolti. La stessa app invia opportune notifiche a guidatori e passeggeri, oltre a gestire i pagamenti. Individuate le classi che potrebbero essere utili per progettare tale sistema.
- ★ **R8.2.** Immaginate di voler progettare un *social network* per gestire i progetti di tirocinio nella vostra università. Lo sponsor di un progetto lo descrive, specificando le capacità richieste, la data in cui desidera che sia portato a termine e il lavoro che presume sia necessario svolgere. Gli studenti usano un'applicazione di ricerca che consente loro di trovare progetti corrispondenti alle proprie capacità e disponibilità. Individuate le classi che potrebbero essere utili per progettare tale sistema.
- ★★ **R8.3.** Vi viene assegnato il compito di scrivere un programma che simuli il comportamento di un distributore automatico. Gli utenti selezionano un prodotto e pagano: se le monete inserite sono sufficienti a raggiungere il prezzo di acquisto del prodotto, questo viene fornito insieme all'eventuale resto, altrimenti le monete inserite vengono restituite all'utente. Fornite un nome adeguato per la classe che implementa questo programma e i nomi di due classi che sarebbero utili, spiegandone i motivi.
- ★★ **R8.4.** Vi viene assegnato il compito di scrivere un programma che acquisisca in ingresso il nome e l'indirizzo di un cliente, seguito da un elenco di articoli acquistati e dei relativi prezzi, per poi visualizzare la fattura corrispondente. Quali delle seguenti classi sarebbero utili per implementare questo programma? `Invoice` (*fattura*), `InvoicePrinter` (*visualizzatore di fattura*), `PrintInvoice` (*visualizza una fattura*), `InvoiceProgram` (*programma per fatture*).
- ★★★ **R8.5.** Vi viene assegnato il compito di scrivere un programma che elabori buste paga. Gli impiegati ricevono la loro busta paga settimanalmente e vengono pagati per ogni ora lavorata in base alla

loro paga oraria. Se, però, hanno lavorato più di 40 ore, le ore di straordinario vengono pagate il 150% della paga ordinaria. Fornite un nome appropriato per una classe di tipo "attore" che sarebbe adeguata per realizzare questo programma, poi individuate il nome di una classe che non sia di tipo "attore" ma che sarebbe una valida alternativa. In che modo la scelta tra queste due alternative influenza la struttura del programma?

- \*\* **R8.6.** Analizzate l'interfaccia pubblica della classe `java.lang.System` e discutete la sua coesione.
- \*\* **R8.7.** Supponete che un oggetto `Invoice` (*fattura*) contenga le descrizioni dei prodotti ordinati, insieme agli indirizzi di spedizione e di fatturazione del cliente. Disegnate un diagramma UML che mostri le dipendenze tra le classi `Invoice`, `Address` (*indirizzo*), `Customer` (*cliente*) e `Product` (*prodotto*).
- \*\* **R8.8.** Supponete che un distributore automatico contenga prodotti e che gli utenti vi inseriscano monete per comprare. Disegnate un diagramma UML che mostri le dipendenze tra le classi `VendingMachine` (*distributore automatico*), `Coin` (*moneta*) e `Product` (*prodotto*).
- \*\* **R8.9.** Da quali classi dipende la classe `Integer` della libreria standard?
- \*\* **R8.10.** Da quali classi dipende la classe `Rectangle` della libreria standard?
- \* **R8.11.** Catalogate i metodi della classe `Scanner` usati in questo libro come metodi d'accesso o modificatori.
- \* **R8.12.** Catalogate i metodi della classe `Rectangle` come metodi d'accesso o modificatori.
- \* **R8.13.** La classe `Resistor` dell'Esercizio P8.12 è mutabile o immutabile? Perché?
- \* **R8.14.** Quali delle seguenti classi sono immutabili?
  - a. `Rectangle`
  - b. `String`
  - c. `Random`
- \* **R8.15.** Quali delle seguenti classi sono immutabili?
  - a. `PrintStream`
  - b. `Date`
  - c. `Integer`

- \*\*\* **R8.16.** Considerate il metodo seguente:

```
public class DataSet
{
    /**
     * Legge numeri da uno scanner e li aggiunge a questo insieme.
     * @param in un oggetto di tipo Scanner
     */
    public void read(Scanner in) { . . . }
    . .
}
```

Descrivete gli effetti collaterali del metodo `read`. Quali sono sconsigliati, secondo quanto esposto nel Paragrafo 8.2.4? In che modo si possono eliminare tali effetti collaterali indesiderati? Che effetto ha sull'accoppiamento la modifica del progetto?

- \*\* **R8.17.** Quali effetti collaterali hanno i seguenti tre metodi (se ne hanno)?

```
public class Coin
{
```

```

    ...
    public void print()
    {
        System.out.println(name + " " + value);
    }

    public void print(PrintStream stream)
    {
        stream.println(name + " " + value);
    }

    public String toString()
    {
        return name + " " + value;
    }
}

```

- \*\*\* **R8.18.** Idealmente un metodo non dovrebbe avere effetti collaterali. Potete scrivere un programma in cui nessun metodo ha un effetto collaterale? Sarebbe un programma utile?
- \*\* **R8.19.** Esamine il metodo seguente, creato per scambiare tra loro i valori di due numeri interi:

```

public static void falseSwap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}

public static void main(String[] args)
{
    int x = 3;
    int y = 4;
    falseSwap(x, y);
    System.out.println(x + " " + y);
}

```

Perché il metodo non scambia il contenuto di `x` e di `y`?

- \*\*\* **R8.20.** In che modo potete scrivere un metodo che scambi il valore di due numeri in virgola mobile? *Suggerimento:* `java.awt.Point`.
- \*\* **R8.21.** Tracciate uno schema di ciò che accade in memoria, in modo da porre in evidenza il motivo per cui il metodo seguente non è in grado di scambiare tra loro due oggetti di tipo `BankAccount`:

```

public static void falseSwap(BankAccount a, BankAccount b)
{
    BankAccount temp = a;
    a = b;
    b = temp;
}

```

- \* **R8.22.** Considerate la classe `Die` del Capitolo 6, modificata mediante l'aggiunta di una variabile statica:

```

public class Die
{

```

```
private int sides;
private static Random generator = new Random();
public Die(int s) { . . . }
public int cast() { . . . }
}
```

Tracciate un grafico degli oggetti in memoria che mostri questi tre dadi:

```
Die d4 = new Die(4);
Die d6 = new Die(6);
Die d8 = new Die(8);
```

Non dimenticate di rappresentare i valori delle variabili `sides` e `generator`.

- \* **R8.23.** Provate a compilare il programma seguente e spiegate il messaggio d'errore che si ottiene.

```
public class Print13
{
    public void print(int x)
    {
        System.out.println(x);
    }

    public static void main(String[] args)
    {
        int n = 13;
        print(n);
    }
}
```

- \* **R8.24.** Analizzate i metodi della classe `Integer`. Quali sono statici? Perché?
- \*\* **R8.25.** Analizzate i metodi della classe `String` (ignorando quelli che ricevono un parametro di tipo `char[]`). Quali sono statici? Perché?
- \*\* **R8.26.** Analizzate il problema di *allineare completamente i margini* di un paragrafo di testo, avendo come obiettivo una determinata larghezza, inserendo il massimo numero possibile di parole su una riga e distribuendo equamente spazi aggiuntivi in modo che ogni riga abbia la larghezza richiesta. Individuate un piano di lavoro per scrivere un programma che legga un paragrafo di testo e lo visualizzi con i margini allineati. Descrivete una sequenza di programmi intermedi di complessità crescente, seguendo un approccio simile a quello visto nel Paragrafo 8.5.
- \*\* **R8.27.** Le variabili `in` e `out` della classe `System` sono variabili statiche pubbliche. È una buona scelta di progetto? Se non lo è, come la migliorereste?
- \*\* **R8.28.** Qualsiasi programma Java può essere riscritto eliminando gli enunciati `import`. Spiegate come ciò sia possibile e riscrivete il file `RectangleComponent.java` del Paragrafo 2.9.3 eliminando gli enunciati `import`.
- \* **R8.29.** Che cos'è il pacchetto predefinito (*default package*)? L'avete usato nella vostra programmazione prima di leggere questo capitolo?
- \*\* **R8.30 (collaudo).** Come si comporta JUnit quando un metodo di collaudo lancia un'eccezione? Provate e descrivete ciò che scoprite.

## Esercizi di programmazione

- \*\* E8.1. Realizzate la classe `Coin` descritta nel Paragrafo 8.2 e modificate la classe `CashRegister` in modo che si possano aggiungere monete al registratore di cassa mediante il nuovo metodo

```
void receivePayment(int coinCount, Coin coinType)
```

Tale metodo va invocato una volta per ogni tipo di moneta presente nel pagamento.

- \*\* E8.2. Modificate il metodo `giveChange` della classe `CashRegister` in modo che restituisca il numero di monete di un particolare tipo che vanno fornite come resto

```
int giveChange(Coin coinType)
```

Tale metodo va invocato una volta per ogni tipo di moneta, in ordine di valore decrescente.

- \* E8.3. I veri registratori di cassa possono gestire monete e banconote. Progettate un'unica classe che rappresenti ciò che accomuna questi due concetti, poi riprogettate la classe `CashRegister` e dotatela di un metodo che consenta la registrazione di pagamenti descritti da tale classe. Uno degli obiettivi principali è l'identificazione di un nome significativo per la classe.

- \*\* E8.4. Riprogettate la classe `BankAccount` in modo che sia immutabile: i metodi `deposit` e `withdraw` devono restituire nuovi oggetti `BankAccount` aventi il saldo opportuno.

- \* E8.5. Riprogettate la classe `Day` vista nella sezione Esempi completi 2.1 in modo che sia mutabile: i metodi `addDays`, `nextDay` e `previousDay` devono ora modificare il proprio parametro implicito ed essere di tipo `void`. Modificate opportunamente il programma dimostrativo.

- \*\* E8.6. Progettate i seguenti metodi statici per calcolare il volume e l'area della superficie di un cubo con altezza `h`, di una sfera con raggio `r`, di un cilindro con altezza `h` e base circolare di raggio `r` e di un cono con altezza `h` e base circolare di raggio `r`. Inseriteli nella classe `Geometry` e scrivete un programma che chieda all'utente di inserire i valori per `r` e per `h`, che invochi gli otto metodi e che visualizzi i risultati.

- `public static double cubeVolume(double h)`
- `public static double cubeSurface(double h)`
- `public static double sphereVolume(double r)`
- `public static double sphereSurface(double r)`
- `public static double cylinderVolume(double r, double h)`
- `public static double cylinderSurface(double r, double h)`
- `public static double coneVolume(double r, double h)`
- `public static double coneSurface(double r, double h)`

- \*\* E8.7. Risolvete nuovamente l'esercizio precedente realizzando le classi `Cube`, `Sphere`, `Cylinder` e `Cone`. Quale approccio è maggiormente orientato agli oggetti?

- \*\*\* E8.8. Modificate l'applicazione vista nei Consigli pratici 7.1 in modo che gestisca più studenti: per prima cosa chiedete all'utente di fornire i nomi di tutti gli studenti; poi, leggete i voti di tutte le singole prove d'esame, una dopo l'altra, chiedendo il punteggio di ciascuno studente (visualizzandone il nome); infine, visualizzate i nomi di tutti gli studenti, con il relativo voto finale. Usate un'unica classe, avente soltanto metodi statici.

- \*\*\* E8.9. Risolvete nuovamente l'esercizio precedente usando più classi: progettate la classe `GradeBook` (*registro delle valutazioni*) in modo che memorizzi un insieme di oggetti di tipo `Student`.

- \*\*\* **E8.10.** Progettate questi due metodi che calcolano il perimetro e l'area dell'ellisse e, aggiungendoli alla classe `Geometry`:

```
public static double perimeter(Ellipse2D.Double e)
public static double area(Ellipse2D.Double e)
```

La parte più difficile dell'esercizio è l'identificazione e la realizzazione di una formula corretta per il calcolo del perimetro. Perché in questo caso è sensato usare un metodo statico?

- \*\* **E8.11.** Progettate questi due metodi che calcolano (in gradi) l'angolo compreso tra l'asse  $x$  e la retta che passa per due punti, e la pendenza (cioè il coefficiente angolare) di tale retta:

```
public static double angle(Point2D.Double p, Point2D.Double q)
public static double slope(Point2D.Double p, Point2D.Double q)
```

Aggiungete i metodi alla classe `Geometry`. Perché in questo caso è sensato usare un metodo statico?

- \*\*\* **E8.12.** Progettate questi due metodi che verificano se un punto si trova all'interno o sul contorno dell'ellisse e, aggiungendoli alla classe `Geometry`:

```
public static boolean isInside(Point2D.Double p, Ellipse2D.Double e)
public static boolean isOnBoundary(Point2D.Double p, Ellipse2D.Double e)
```

- \*\* **E8.13.** Usando la classe `Picture` vista nella sezione Esempi completi 6.2, scrivete un metodo

```
public static Picture superimpose(Picture pic1, Picture pic2)
```

che sovrapponga due immagini, generando un'immagine le cui dimensioni (larghezza e altezza) siano uguali al valore massimo tra le dimensioni corrispondenti delle due immagini `pic1` e `pic2`. Nell'area in cui le due immagini si sovrappongono fate la media tra i colori, pixel per pixel.

- \*\* **E8.14.** Usando la classe `Picture` vista nella sezione Esempi completi 6.2, scrivete un metodo

```
public static Picture greenScreen(Picture pic1, Picture pic2)
```

che sovrapponga due immagini, generando un'immagine le cui dimensioni (larghezza e altezza) siano uguali al valore massimo tra le dimensioni corrispondenti delle due immagini `pic1` e `pic2`. Nell'area in cui le due immagini si sovrappongono usate i pixel di `pic1`, tranne quando sono verdi, nel qual caso usate i pixel di `pic2`.

- \* **E8.15.** Progettate il metodo

```
public static int readInt(
    Scanner in, String prompt, String error, int min, int max)
```

che visualizzi il messaggio (`prompt`), legga un numero intero e verifichi se si trova tra il valore minimo e il valore massimo specificati. In caso negativo, deve visualizzare il messaggio d'errore (`error`) e ripetere l'acquisizione del dato. Inserite il metodo nella classe `Input`.

- \*\* **E8.16.** Esimate il seguente algoritmo per calcolare  $x^n$ , con  $n$  intero. Se  $n < 0$ ,  $x^n$  è uguale a  $1/x^{-n}$ . Se  $n$  è positivo e pari, allora  $x^n = (x^{n/2})^2$ . Se  $n$  è positivo e dispari, allora  $x^n = x^{n-1} \cdot x$ . Progettate il metodo statico `double intPower(double x, int n)` che utilizzi questo algoritmo, inserendolo nella classe `Numeric`.

- \*\* **E8.17.** Migliorate la classe `Die` vista nel Capitolo 6, rendendo statica la variabile `generator`, in modo che tutti i dadi condividano un unico generatore di numeri casuali.

- \*\* **E8.18.** Realizzate le classi `CashRegister` e `Coin` descritte nell'Esercizio E8.1, inserendole nel pacchetto `money` e lasciando, invece, la classe `CashRegisterTester` nel pacchetto predefinito.
- \* **E8.19.** Inserite la classe `BankAccount` in un pacchetto il cui nome sia derivato dal vostro indirizzo di posta elettronica, come descritto nel Paragrafo 8.6, lasciando invece la classe `BankAccountTester` nel pacchetto predefinito.
- \*\* **E8.20 (collaudo).** Progettate con JUnit una classe di collaudo, `StudentTest`, avente tre metodi, ciascuno dei quali collauda un diverso metodo della classe `Student` vista nella sezione Consigli pratici 7.1.
- \*\* **E8.21 (collaudo).** Progettate con JUnit una classe di collaudo, `TaxReturnTest`, avente tre metodi, ciascuno dei quali collauda una diversa situazione fiscale della classe `TaxReturn` vista nel Capitolo 5.
- \* **E8.22 (grafica).** Scrivete i metodi seguenti per disegnare le lettere H, E, L e O in una finestra grafica; il parametro `p` indica il vertice superiore sinistro del rettangolo che racchiude la lettera. Tracciate linee ed ellissi, ma non usate `System.out` né il metodo `drawString`. Poi, invocate ripetutamente tali metodi per disegnare le parole “HELLO” e “HOLE”.
  - `public static void drawH(Graphics2D g2, Point2D.Double p);`
  - `public static void drawE(Graphics2D g2, Point2D.Double p);`
  - `public static void drawL(Graphics2D g2, Point2D.Double p);`
  - `public static void drawO(Graphics2D g2, Point2D.Double p);`
- \*\* **E8.23 (grafica).** Ripetete l'esercizio precedente progettando le classi `LetterH`, `LetterE`, `LetterL` e `LetterO`, ciascuna con un costruttore che riceve un parametro di tipo `Point2D.Double` (il vertice superiore sinistro del rettangolo che racchiude la lettera) e un metodo `draw(Graphics2D g2)`. Quale soluzione è più orientata agli oggetti?
- \*\* **E8.24.** Aggiungete alla classe `BankAccount` il metodo `ArrayList<Double> getStatement()` che restituisca un elenco di tutti i versamenti e prelievi sotto forma di valori, rispettivamente, positivi e negativi. Aggiungete anche il metodo `void clearStatement()` che svuoti l'elenco.
- \*\* **E8.25.** Progettate la classe `LoginForm` che simuli la procedura di autenticazione (*login*) che caratterizza molte pagine web, dotandola dei metodi seguenti:

```
public void input(String text)
public void click(String button)
public boolean loggedIn()
```

Per prima cosa viene acquisita una stringa con il nome dell'utente (*username*), poi la parola d'accesso (*password*). Il metodo `click` può essere invocato per simulare la pressione di un pulsante grafico fornendo come argomento "Submit" (per inviare i dati) oppure "Reset" (per azzerare la procedura). Dopo che l'utente si è autenticato correttamente (fornendo *username*, *password* e selezionando il pulsante *submit*) il metodo `loggedIn` restituisce `true` e ulteriori dati non hanno alcun effetto. Quando un utente cerca di superare l'autenticazione fornendo un nome utente o una parola d'accesso non validi, i dati acquisiti vengono ignorati, come quando si seleziona il pulsante *reset*. Dotate la classe di un costruttore che riceva come parametri le stringhe *username* e *password* che devono essere usate per l'autenticazione.

- \*\* **E8.26.** Progettate la classe `Robot` che simuli un robot che si muove su un piano infinito. La posizione del robot è individuata da un punto sul piano (con numeri interi come coordinate) e da una direzione di movimento (che può essere nord, est, sud o ovest). Progettate i metodi:

```
public void turnLeft()  
public void turnRight()  
public void move()  
public Point getLocation()  
public String getDirection()
```

I metodi `turnLeft` e `turnRight` cambiano la direzione (rispettivamente, in senso antiorario e orario) ma non la posizione del robot. Il metodo `move` sposta il robot di un'unità nella direzione che lo caratterizza, mentre il metodo `getDirection` restituisce una stringa: "N", "E", "S" oppure "W".

**Sul sito web dedicato al libro si trova una raccolta di progetti di programmazione più complessi.**



# 9

## Ereditarietà



### Obiettivi del capitolo

- Imparare l'ereditarietà
- Implementare sottoclassi che ereditino e sovrascrivano metodi della superclasse
- Comprendere il concetto di polimorfismo
- Capire il concetto di superclasse comune, `Object`, e conoscere i suoi metodi

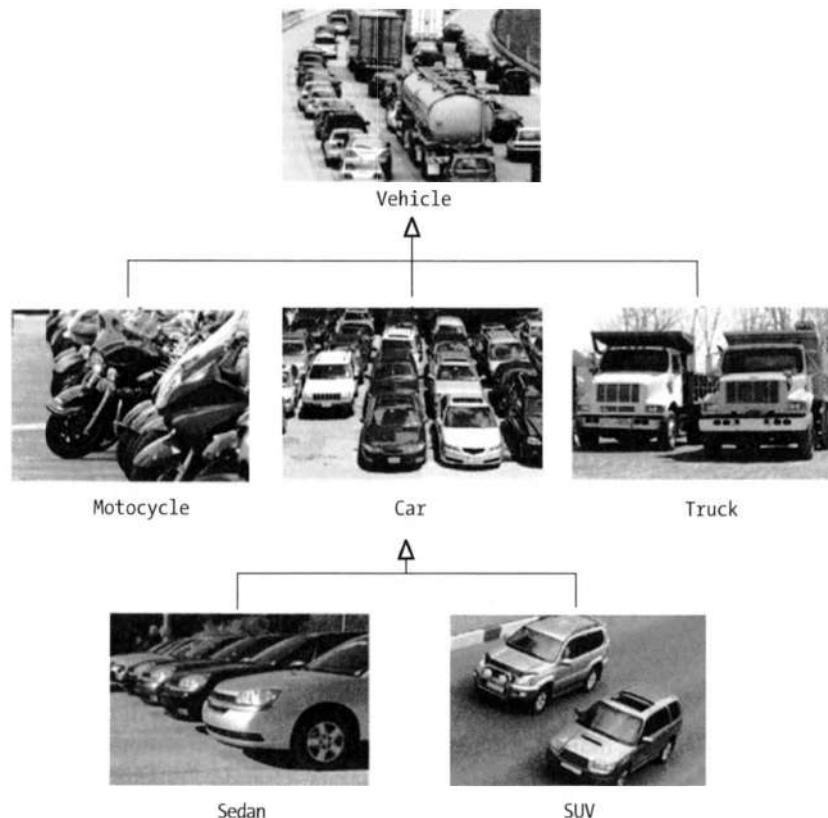
Spesso oggetti di classi correlate condividono un comportamento comune: ad esempio, automobili, biciclette e autobus sono tutti mezzi di trasporto. In questo capitolo vedrete come usare il concetto di ereditarietà per esprimere la relazione esistente tra una classe generica e una classe specializzata. Usando l'ereditarietà sarete in grado di condividere codice tra classi diverse, fornendo servizi che potranno essere utilizzati da più classi.

## 9.1 Gerarchie di ereditarietà

Una sottoclasse eredita dati e comportamenti da una superclasse.

Nella progettazione orientata agli oggetti, l'**ereditarietà** (*inheritance*) è una relazione esistente tra una classe più generica, detta **superclasse** (*superclass*), e una più specifica e specializzata, detta **sottoclasse** (*subclass*): la sottoclasse “eredita” i dati e i comportamenti dalla propria superclasse. Consideriamo, come esempio, la relazione esistente tra diversi tipi di veicoli, come descritta nella Figura 1.

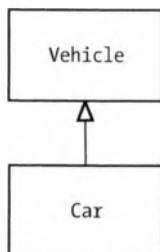
**Figura 1**  
Una gerarchia  
di ereditarietà tra classi  
di veicoli



Ogni automobile è un veicolo e le automobili condividono molte caratteristiche con tutti gli altri veicoli, come la capacità di trasportare persone da un luogo a un altro. Diciamo, quindi, che la classe `Car` (*automobile*) eredita le proprietà della classe `Vehicle` (*veicolo*). In questa relazione la classe `Vehicle` è la superclasse e la classe `Car` è la sottoclasse. Nella Figura 2, superclasse e sottoclasse sono collegate da una freccia che punta verso la superclasse.

In un programma, l'uso dell'ereditarietà consente di riutilizzare codice invece di ricopiarlo, un riutilizzo che assume due forme diverse. Innanzitutto, una sottoclasse eredita i metodi della superclasse. Ad esempio, se la classe `Vehicle` ha un metodo `drive`, che consente di guidare il veicolo in qualche modo, allora la sottoclasse `Car` eredita automaticamente tale metodo, non c'è bisogno di ricopiarlo.

**Figura 2**  
Un diagramma di ereditarietà



Un oggetto di una sottoclasse  
può sempre essere utilizzato al posto  
di un oggetto della sua superclasse.

La seconda forma è più sottile e consente di riutilizzare algoritmi che manipolano oggetti di tipo `Vehicle`. Dato che un'automobile è un tipo particolare di veicolo, possiamo far manipolare a uno di tali algoritmi anche un oggetto di tipo `Car` e tutto funzionerà correttamente. Il **principio di sostituzione** (*substitution principle*) afferma che si può sempre usare un oggetto di una sottoclasse in un punto in cui è prevista la presenza di un oggetto della sua superclasse. Consideriamo, ad esempio, un metodo che riceve un argomento di tipo `Vehicle`:

```
void processVehicle(Vehicle v)
```

Dato che `Car` è una sottoclasse di `Vehicle`, si può invocare tale metodo anche con un oggetto di tipo `Car`:

```
Car myCar = new Car(. . .);
processVehicle(myCar);
```

Perché mai dovremmo progettare un metodo che elabora veicoli anziché, più specificatamente, automobili? Questo metodo è più utile, in quanto può manipolare *qualsiasi* tipo di veicolo (compresi autocarri e motocicli).

In questo capitolo analizzeremo una semplice gerarchia di classi che rappresentano domande di un questionario e molto probabilmente avrete già risposto a molti questionari valutati automaticamente da un calcolatore. Un questionario (*quiz*) è composto da più domande (*question*), che possono essere di tipi diversi:

- con spazi da riempire (*fill-in-the-blank*);
- a scelta (singola o multipla) tra risposte predefinite (*single choice* o *multiple choice*);
- numeriche (*numeric*, nelle quali è spesso accettabile anche una risposta approssimata, ad esempio 1.33 è una risposta valida anche quando la risposta corretta sarebbe 4/3);
- aperta (*free response*).

La Figura 3 mostra una gerarchia di ereditarietà per questi tipi di domande.

In cima alla gerarchia troviamo il tipo `Question`: una generica “domanda” può visualizzare il proprio testo e può verificare se una risposta data è quella corretta.

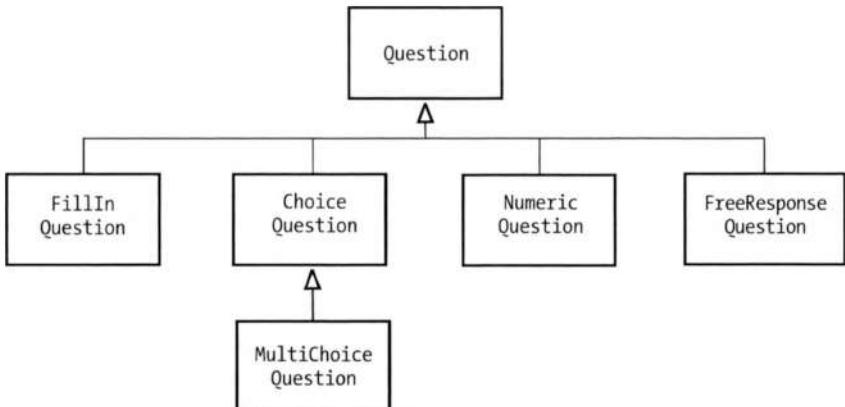
### File Question.java

```

1  /**
2   * Una domanda con un testo e una risposta corretta.
3  */
  
```

**Figura 3**

Gerarchia di ereditarietà per tipi di domande



```

4 public class Question
5 {
6     private String text;
7     private String answer;
8
9     /**
10      Costruisce una domanda con stringhe vuote come testo e risposta.
11 */
12     public Question()
13     {
14         text = "";
15         answer = "";
16     }
17
18     /**
19      Imposta il testo della domanda.
20      @param questionText il testo della domanda
21 */
22     public void setText(String questionText)
23     {
24         text = questionText;
25     }
26
27     /**
28      Imposta la risposta corretta di questa domanda.
29      @param correctResponse la risposta corretta
30 */
31     public void setAnswer(String correctResponse)
32     {
33         answer = correctResponse;
34     }
35
36     /**
37      Verifica se la risposta data è quella corretta.
38      @param response la risposta da verificare
39      @return true se la risposta è corretta, false altrimenti
40 */
41     public boolean checkAnswer(String response)
42     {
  
```

```
43     return response.equals(answer);
44 }
45
46 /**
47     Visualizza (il testo di) questa domanda.
48 */
49 public void display()
50 {
51     System.out.println(text);
52 }
53 }
```

Questa classe `Question` è veramente molto elementare e non gestisce domande a risposta multipla, domande numeriche e così via. Nei paragrafi che seguono vedrete come definire sottoclassi di `Question`, intanto vediamo un semplice programma di collaudo per la classe `Question`.

### File QuestionDemo1.java

```
1 import java.util.Scanner;
2
3 /**
4     Visualizza un semplice questionario con una sola domanda.
5 */
6 public class QuestionDemo1
7 {
8     public static void main(String[] args)
9     {
10         Scanner in = new Scanner(System.in);
11
12         Question q = new Question();
13         q.setText("Who was the inventor of Java?");
14         q.setAnswer("James Gosling");
15
16         q.display();
17         System.out.print("Your answer: ");
18         String response = in.nextLine();
19         System.out.println(q.checkAnswer(response));
20     }
21 }
```

### Esecuzione del programma

```
Who was the inventor of Java?
Your answer: James Gosling
true
```



### Auto-valutazione

1. In una relazione di ereditarietà tra le classi `Employee` (*dipendente*) e `Manager` (*dirigente*), quale dovrebbe essere la superclasse e quale la sottoclasse?
2. Che relazioni di ereditarietà ci sono tra le classi `BankAccount` (*conto bancario*), `CheckingAccount` (*conto corrente bancario*) e `SavingsAccount` (*conto bancario di risparmio*)?

3. Consultando la documentazione API, elencate tutte le superclassi della classe `JFrame`.
4. Dato il metodo `doSomething(Car c)`, che richiede un argomento di tipo `Car`, elencate le classi della Figura 1 i cui oggetti *non possono* essere passati come argomento a tale metodo.
5. La classe `Quiz` (*questionario*) dovrebbe essere una sottoclasse della classe `Question`? Perché?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R9.2, R9.8 e R9.10, al termine del capitolo.



## Suggerimenti per la programmazione 9.1

### Usare un'unica classe per variazioni di valori e l'ereditarietà per variazioni di comportamento

Lo scopo dell'ereditarietà è quello di rappresentare oggetti aventi *comportamenti* diversi. Quando si inizia a studiare l'ereditarietà si tende a usarla troppo spesso, creando molte classi anche quando le diversità tra i loro oggetti potrebbero essere espresse con una semplice variabile di esemplare.

Considerate un programma che tiene traccia dell'efficienza nel consumo di carburante in una flotta di automobili, registrando le distanze percorse e l'entità dei rifornimenti. Alcune automobili della flotta sono a motore ibrido. Si deve progettare la sottoclasse `HybridCar`? Non in questa applicazione: quando si tratta di fare rifornimento e percorrere chilometri i veicoli ibridi non si comportano in modo diverso dalle altre automobili, hanno soltanto una migliore efficienza nel consumo di carburante. Un'unica classe `Car`, con la variabile di esemplare

```
double milesPerGallon;
```

che memorizza l'efficienza in miglia per galloni, è assolutamente sufficiente.

Se, invece, dovete scrivere un programma che mostra come si riparano diversi tipi di veicoli, allora ha senso utilizzare una classe `HybridCar` distinta, perché, in merito alle riparazioni, le automobili ibride hanno un comportamento diverso dalle altre.

## 9.2 Realizzare sottoclassi

In questo paragrafo vedrete come definire una sottoclasse e come le sottoclassi ereditino automaticamente le funzionalità della propria superclasse.

Supponiamo di voler scrivere un programma che gestisca domande, come questa:

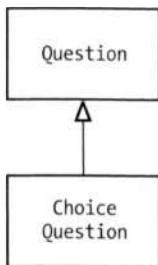
In which country was the inventor of Java born?

1. Australia
2. Canada
3. Denmark
4. United States

Potreste scrivere una classe, `ChoiceQuestion`, partendo da zero e dotandola di metodi per impostare il testo della domanda, per visualizzarla e per verificare se una risposta data sia quella corretta, ma non c'è bisogno di farlo. Usate, invece, l'ereditarietà e realizzate la classe `ChoiceQuestion` come sottoclasse della classe `Question` (come indicato nella Figura 4).

**Figura 4**

La classe ChoiceQuestion  
è una sottoclasse  
della classe Question



Una sottoclasse eredita tutti i metodi  
che non sovrascrive.

Una sottoclasse può sovrascrivere  
un metodo ereditato  
dalla sua superclasse fornendone  
una nuova implementazione.

In Java si definisce una sottoclasse specificando cosa la rende *diversa* dalla sua superclasse.

Un oggetto di una sottoclasse possiede automaticamente tutte le variabili di esemplare che sono dichiarate nella superclasse: occorre dichiarare soltanto quelle variabili di esemplare che non sono presenti negli oggetti della superclasse.

La sottoclasse eredita tutti i metodi della sua superclasse e deve definire soltanto i metodi che sono *nuovi* (cioè non sono presenti nella superclasse) oppure che *modificano* l'implementazione di metodi ereditati, se il comportamento ereditato non è adeguato. Quando si fornisce una nuova implementazione di un metodo ereditato si dice che lo si **sovrascrive** (*override*).

Un oggetto di tipo ChoiceQuestion differisce da uno di tipo Question per tre aspetti:

- memorizza le diverse opzioni possibili per la risposta;
- c'è un metodo per aggiungere le opzioni possibili per la risposta;
- il metodo `display` della classe ChoiceQuestion mostra le opzioni possibili per la risposta, in modo che chi deve rispondere alla domanda possa sceglierne una.

Pur ereditando il comportamento dalla classe Question, la classe ChoiceQuestion deve evidenziare queste tre differenze:

```
public class ChoiceQuestion extends Question
{
    // questa variabile di esemplare viene aggiunta alla sottoclasse
    private ArrayList<String> choices;

    // questo metodo viene aggiunto alla sottoclasse
    public void addChoice(String choice, boolean correct) { . . . }

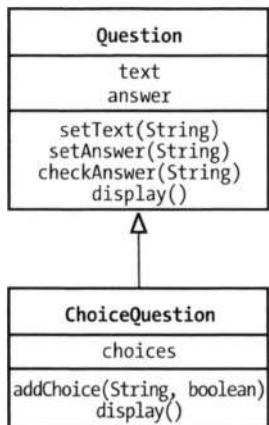
    // questo metodo sovrascrive un metodo della superclasse
    public void display() { . . . }
}
```

La parola riservata `extends`  
definisce la relazione di ereditarietà.

La parola riservata `extends` definisce la relazione di ereditarietà. La Figura 5 mostra come si rappresentano in un diagramma UML i metodi e le variabili di esemplare quando una classe eredita da un'altra.

**Figura 5**

La classe `ChoiceQuestion` aggiunge una variabile di esemplare e un metodo, oltre a sovrascriverne un altro



La Figura 6 mostra la struttura interna di un oggetto di tipo `ChoiceQuestion`: ha le variabili di esemplare che sono state definite nella superclasse `Question` (`text` e `answer`) e ne aggiunge un'altra, `choices`.

Il metodo `addChoice` è specifico della classe `ChoiceQuestion`: lo si può applicare soltanto a oggetti di tipo `ChoiceQuestion` e non a generici oggetti di tipo `Question`.

Al contrario, il metodo `display` è un metodo che già esisteva nella superclasse, ma la sottoclasse lo sovrascrive, in modo che possa visualizzare correttamente le opzioni tra cui scegliere la risposta.

## Sintassi di Java

### 9.1 Dichiarazione di sottoclassi

#### Sintassi

```

public class NomeSottoclasse extends NomeSuperclasse
{
    variabili di esemplare
    metodi
}
  
```

#### Esempio

Dichiarazione delle variabili di esemplare che vengono aggiunte alla sottoclasse.

Dichiarazione dei metodi che vengono aggiunti nella sottoclasse.

Dichiarazione dei metodi che vengono sovrascritti nella sottoclasse.

#### Sottoclasse

#### Superclasse

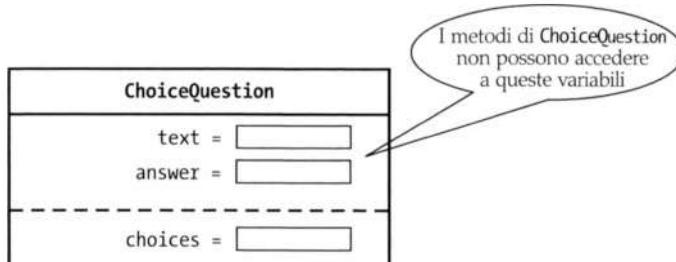
```

public class ChoiceQuestion extends Question
{
    private ArrayList<String> choices;
    public void addChoice(String choice, boolean correct) { . . . }
    public void display() { . . . }
}
  
```

La parola riservata `extends` caratterizza l'ereditarietà.

**Figura 6**

Struttura interna di un oggetto di una sottoclasse



Tutti gli altri metodi della classe `Question` vengono automaticamente ereditati dalla classe `ChoiceQuestion` e possono essere invocati con oggetti della sottoclasse:

```
choiceQuestion.setAnswer("2");
```

Tuttavia, le variabili di esemplare private della superclasse sono inaccessibili e soltanto i metodi della superclasse vi possono accedere: la sottoclasse non ha, nei loro confronti, maggiori diritti di accesso di qualsiasi altra classe.

Quindi, ad esempio, i metodi della sottoclasse `ChoiceQuestion` non possono accedere direttamente alla variabile `answer`: devono usare i metodi dell'interfaccia pubblica della classe `Question` per accedere ai suoi dati privati, esattamente come farebbe qualunque altro metodo.

Per illustrare meglio questo punto, implementiamo il metodo `addChoice`, che riceve due argomenti: l'opzione da aggiungere (che viene inserita in fondo alla lista delle opzioni) e un valore booleano che indica se si tratta dell'opzione corretta. Ad esempio:

```
choiceQuestion.addChoice("Canada", true)
```

Il primo argomento ricevuto deve essere aggiunto in fondo alla lista a cui fa riferimento la variabile di esemplare `choices`. Se il secondo argomento è il valore `true`, allora la variabile di esemplare `answer` deve diventare uguale al numero associato alla scelta che si sta aggiungendo. Ad esempio, se `choices.size()` vale 2, allora alla variabile `answer` viene assegnata la stringa "2".

```

public void addChoice(String choice, boolean correct)
{
    choices.add(choice);
    if (correct)
    {
        // converte choices.size() in stringa
        String choiceString = "" + choices.size();
        setAnswer(choiceString);
    }
}
  
```

Non si può accedere direttamente alla variabile di esemplare `answer` della superclasse, ma, fortunatamente, la classe `Question` mette a disposizione il metodo `setAnswer`, che possiamo invocare. Con quale oggetto lo invochiamo? Con la domanda che stiamo modificando, cioè con il parametro implicito del metodo `ChoiceQuestion.addChoice`. Ricordate: se invocate un metodo usando il parametro implicito, non c'è bisogno di

specificare esplicitamente il parametro implicito stesso, basta scrivere il nome del metodo che si vuole invocare.

```
setAnswer(choiceString);
```

Se preferite, però, potete rendere più evidente che il metodo viene invocato con il parametro implicito, in questo modo:

```
this.setAnswer(choiceString);
```



## Auto-valutazione

6. Se `q` è un oggetto di tipo `Question` e `cq` è un oggetto di tipo `ChoiceQuestion`, quali delle seguenti invocazioni sono valide?
  - a. `q.setAnswer(response)`
  - b. `cq.setAnswer(response)`
  - c. `q.addChoice(choice, true)`
  - d. `cq.addChoice(choice, true)`
7. Data la seguente definizione della classe `Employee`, dichiarate la classe `Manager` che eredita da `Employee` aggiungendo una variabile di esemplare, `bonus`, che memorizzi il premio di produzione. Non indicate costruttori e metodi.

```
public class Employee
{
    private String name;
    private double baseSalary;

    public void setName(String newName) { . . . }
    public void setBaseSalary(double newSalary) { . . . }
    public String getName() { . . . }
    public double getSalary() { . . . }
}
```

8. Quali sono le variabili di esemplare della classe `Manager` definita nella risposta alla domanda precedente?
9. Nella classe `Manager` definita nella risposta alla domanda 7, definite l'intestazione (ma non l'implementazione) di un metodo che sovrascriva il metodo `getSalary` della classe `Employee`.
10. Quali sono i metodi ereditati dalla classe `Manager` definita nella risposta alla domanda precedente?

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi R9.4, E9.9 e E9.13, al termine del capitolo.



## Errori comuni 9.1

### Duplicare variabili di esemplare della superclasse

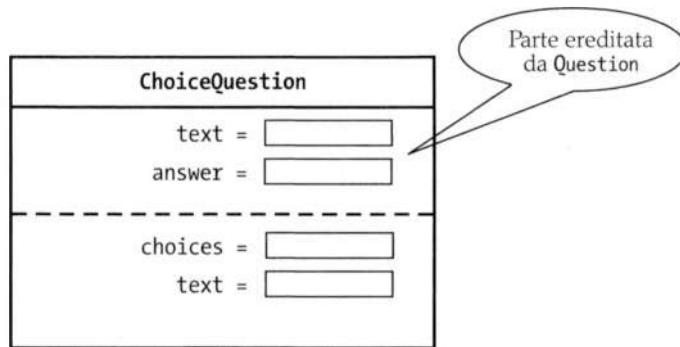
Una sottoclasse non ha accesso alle variabili di esemplare private della superclasse.

```
public ChoiceQuestion(String questionText)
{
    text = questionText;
    // ERRORE: cerca di accedere alla variabile privata della superclasse
}
```

Spesso i programmatori alle prime armi cercano di “risolvere” questo problema aggiungendo alla sottoclasse *un’altra* variabile di esemplare avente lo stesso nome.

```
public class ChoiceQuestion extends Question
{
    private ArrayList<String> choices;
    private String text; // NON FATELO
    ...
}
```

Ora il costruttore viene certamente compilato, ma non aggiorna il testo giusto! Un tale oggetto di tipo `ChoiceQuestion` ha due variabili di esemplare, entrambe di nome `text`. Il costruttore assegna un valore a una delle due, mentre il metodo `display` (della superclasse) visualizza l’altra. La soluzione corretta prevede di accedere alla variabile di esemplare della superclasse attraverso l’interfaccia pubblica della superclasse stessa. Nel nostro esempio, il costruttore di `ChoiceQuestion` dovrebbe invocare il metodo `setText` della classe `Question`.



## Errori comuni 9.2



### Confondere superclassi e sottoclassi

Se confrontate un oggetto di tipo `ChoiceQuestion` e un oggetto di tipo `Question`, scoprirete che:

- La parola riservata `extends` suggerisce che l’oggetto di tipo `ChoiceQuestion` sia una versione “estesa” di un oggetto di tipo `Question`.
- L’oggetto di tipo `ChoiceQuestion` è “più grande”, perché ha una variabile di esemplare in più, `choices`.
- L’oggetto di tipo `ChoiceQuestion` è “più potente”, perché ha un metodo in più, `addChoices`.

L’oggetto di tipo `ChoiceQuestion` sembra essere decisamente superiore. Allora, perché mai la classe `ChoiceQuestion` viene chiamata **sottoclasse** e la classe `Question` viene chiamata **superclasse**?

La terminologia **super/sotto** è mutuata dalla teoria degli insiemi. Osservate l'insieme di tutte le domande: non tutte sono oggetti di tipo `ChoiceQuestion`, alcune di loro sono domande di altro tipo, quindi l'insieme degli oggetti di tipo `ChoiceQuestion` è un **sottoinsieme** dell'insieme di tutti gli oggetti di tipo `Question`, e quest'ultimo è un **superinsieme** dell'insieme degli oggetti di tipo `ChoiceQuestion`. Gli oggetti più specializzati, che appartengono al sottoinsieme, hanno uno stato più ricco di proprietà e hanno maggiori potenzialità.

## 9.3 Sovrascrivere metodi

Un metodo che sovrascrive l'omonimo metodo della superclasse ne può estendere o sostituire la funzionalità.

Una sottoclasse eredita i metodi della superclasse, ma, se il comportamento del metodo ereditato non è adeguato alle esigenze della sottoclasse, può essere **sovrascritto** (*overridden*), specificandone una nuova implementazione nella sottoclasse.

Consideriamo il metodo `display` della classe `ChoiceQuestion`, che sovrascrive il metodo `display` della superclasse in modo da visualizzare le diverse opzioni tra cui va scelta la risposta. Questo metodo *estende* la funzionalità del metodo definito nella superclasse: ciò significa che il metodo della sottoclasse esegue le stesse azioni previste dall'esecuzione del metodo della superclasse (in questo caso, visualizza il testo della domanda) e fa anche qualcosa in più (in questo caso, visualizza le opzioni). In altre situazioni, invece, un metodo di una sottoclasse *sostituisce* la funzionalità del corrispondente metodo della superclasse, realizzando un comportamento completamente diverso.

Passiamo ora all'implementazione del metodo `display` della classe `ChoiceQuestion`. Il metodo deve:

- Visualizzare il testo della domanda.
- Visualizzare le opzioni per la risposta.

La seconda parte è facile, perché le opzioni tra cui scegliere la risposta sono contenute in una variabile di esemplare della sottoclasse stessa.

```
public class ChoiceQuestion extends Question
{
    ...
    public void display()
    {
        // visualizza il testo della domanda
        ...
        // visualizza le risposte tra cui scegliere
        for (int i = 0; i < choices.size(); i++)
        {
            int choiceNumber = i + 1;
            System.out.println(choiceNumber + ": " + choices.get(i));
        }
    }
}
```

Ma come possiamo accedere al testo della domanda, per visualizzarlo? Non possiamo accedere direttamente alla variabile di esemplare `text` della superclasse, perché è privata.

Per invocare un metodo della superclasse si usa la parola riservata `super`:

```
public void display()
{
    // visualizza il testo della domanda
    super.display(); // così va bene
    // visualizza le risposte tra cui scegliere
    ...
}
```

Se dimenticassimo la parola `super`, il metodo non funzionerebbe in modo corretto.

```
public void display()
{
    // visualizza il testo della domanda
    display(); // ERRORE: invoca this.display()
    // visualizza le risposte tra cui scegliere
    ...
}
```

Dato che il parametro implicito `this` fa riferimento a un oggetto di tipo `ChoiceQuestion` e nella classe `ChoiceQuestion` esiste un metodo `display`, sarà quello a essere invocato: ma è il metodo che stiamo scrivendo! Scritto in quel modo, il metodo invocherebbe se stesso indefinitamente.

Vale la pena di osservare che `super`, diversamente da `this`, *non* è un riferimento a un oggetto. *Non* esiste un oggetto della superclasse a sé stante, in qualche modo “associato” a un oggetto della sottoclasse: semplicemente, quest’ultimo contiene direttamente le variabili di esemplare della superclasse. Sintatticamente `super` è soltanto una parola riservata del linguaggio che provoca forzatamente l’esecuzione del metodo della superclasse.

Ecco il programma completo che consente di rispondere a un questionario composto da due oggetti di tipo `ChoiceQuestion`. Li costruiamo e li passiamo, uno alla volta, al metodo `presentQuestion`, che visualizza la domanda e verifica se l’utente ha risposto correttamente.

## Sintassi di Java

## 9.2 Invocazione di un metodo della superclasse

### Sintassi

`super.nomeMetodo(parametri);`

### Esempio

Invoca il metodo della superclasse invece del proprio metodo.

```
public void display()
{
    super.display();
    ...
}
```

Se vi dimenticate `super`, questo metodo invoca se stesso.

**File QuestionDemo2.java**

```

1 import java.util.Scanner;
2
3 /**
4     Visualizza un semplice questionario con due domande a scelta singola.
5 */
6 public class QuestionDemo2
7 {
8     public static void main(String[] args)
9     {
10         ChoiceQuestion first = new ChoiceQuestion();
11         first.setText("What was the original name of the Java language?");
12         first.addChoice("*7", false);
13         first.addChoice("Duke", false);
14         first.addChoice("Oak", true);
15         first.addChoice("Gosling", false);
16
17         ChoiceQuestion second = new ChoiceQuestion();
18         second.setText("In which country was the inventor of Java born?");
19         second.addChoice("Australia", false);
20         second.addChoice("Canada", true);
21         second.addChoice("Denmark", false);
22         second.addChoice("United States", false);
23
24         presentQuestion(first);
25         presentQuestion(second);
26     }
27
28 /**
29     Presenta una domanda all'utente e ne verifica la risposta.
30     @param q la domanda
31 */
32 public static void presentQuestion(ChoiceQuestion q)
33 {
34     q.display();
35     System.out.print("Your answer: ");
36     Scanner in = new Scanner(System.in);
37     String response = in.nextLine();
38     System.out.println(q.checkAnswer(response));
39 }
40 }
```

**File ChoiceQuestion.java**

```

1 import java.util.ArrayList;
2
3 /**
4     Una domanda con più risposte, a scelta singola.
5 */
6 public class ChoiceQuestion extends Question
7 {
8     private ArrayList<String> choices;
9
10    /**
```

```
11     Costruisce una domanda ancora priva di risposte tra cui scegliere.  
12 */  
13 public ChoiceQuestion()  
14 {  
15     choices = new ArrayList<String>();  
16 }  
17  
18 /**  
19     Aggiunge alla domanda una risposta tra cui scegliere.  
20     @param choice la risposta da aggiungere  
21     @param correct true se questa è la risposta corretta, false altrimenti  
22 */  
23 public void addChoice(String choice, boolean correct)  
24 {  
25     choices.add(choice);  
26     if (correct)  
27     {  
28         // converte choices.size() in stringa  
29         String choiceString = "" + choices.size();  
30         setAnswer(choiceString);  
31     }  
32 }  
33  
34 public void display()  
35 {  
36     // visualizza il testo della domanda  
37     super.display();  
38     // visualizza le risposte tra cui scegliere  
39     for (int i = 0; i < choices.size(); i++)  
40     {  
41         int choiceNumber = i + 1;  
42         System.out.println(choiceNumber + ": " + choices.get(i));  
43     }  
44 }  
45 }
```

## Esecuzione del programma

```
What was the original name of the Java language?  
1: *7  
2: Duke  
3: Oak  
4: Gosling  
Your answer: 1  
false  
In which country was the inventor of Java born?  
1: Australia  
2: Canada  
3: Denmark  
4: United States  
Your answer: 2  
true
```



## Auto-valutazione

11. Cosa c'è di sbagliato nella seguente implementazione del metodo `display`?

```
public class ChoiceQuestion extends Question
{
    ...
    public void display()
    {
        System.out.println(text);
        for (int i = 0; i < choices.size(); i++)
        {
            int choiceNumber = i + 1;
            System.out.println(choiceNumber + ": " + choices.get(i));
        }
    }
}
```

12. Cosa c'è di sbagliato nella seguente implementazione del metodo `display`?

```
public class ChoiceQuestion extends Question
{
    ...
    public void display()
    {
        this.display();
        for (int i = 0; i < choices.size(); i++)
        {
            int choiceNumber = i + 1;
            System.out.println(choiceNumber + ": " + choices.get(i));
        }
    }
}
```

13. Tornate ad analizzare l'implementazione del metodo `addChoice`, che invoca il metodo `setAnswer` della superclasse. Perché non abbiamo avuto bisogno di scrivere `super.setAnswer`?
14. Nella classe `Manager` vista nella domanda di auto-valutazione 7, sovrascrivete il metodo `getName` in modo che i dirigenti abbiano un asterisco prima del nome (ad esempio, \*Lin, Sally).
15. Nella classe `Manager` vista nella domanda di auto-valutazione 9, sovrascrivete il metodo `getSalary` in modo che restituisca la somma tra il salario e il premio di produzione.

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi E9.4, E9.5 e E9.14, al termine del capitolo.



## Errori comuni 9.3

### Sovraccaricare accidentalmente

In Java due metodi possono avere lo stesso nome, ma devono avere parametri di tipi diversi. Ad esempio, la classe `PrintStream` ha vari metodi di nome `println`, tra i quali

```
void println(int x)
```

e

```
void println(String x)
```

Sono metodi diversi, ciascuno con la propria implementazione: il compilatore Java li considera completamente indipendenti e solitamente diciamo che il nome `println` è **sovraffunzionato** (*overloaded*). Si tratta di un fenomeno diverso dalla sovrascrittura, dove un metodo di sottoclasse fornisce una diversa implementazione di un metodo della superclasse avente *gli stessi* parametri.

Se avete intenzione di sovrascrivere un metodo ma specificate almeno un parametro di tipo diverso, introducete accidentalmente nella classe un metodo sovraffunzionato. In questo esempio

```
public class ChoiceQuestion extends Question
{
    ...
    public void display(PrintStream out)
        // NON sovrscrive il metodo display()
    {
        ...
    }
}
```

il compilatore non protesta: pensa che vogliate fornire un metodo `display` per gestire argomenti di tipo `PrintStream`, pur ereditando l'altro metodo `display` privo di parametri.

Quando sovrascrivete un metodo, controllate con cura che i tipi dei parametri siano esatti.



## Errori comuni 9.4

### Dimenticare `super` nell'invocazione di metodi della superclasse

Quando si estendono le funzionalità di un metodo di una superclasse, un errore comune consiste nel dimenticarsi `super`. Ad esempio, per calcolare lo stipendio di un dirigente, bisogna ispezionare il salario del sottostante oggetto `Employee`, per poi aggiungere il premio di produzione:

```
public class Manager extends Employee
{
    ...
    public double getSalary()
    {
        double baseSalary = getSalary();
        // Errore: doveva essere super.getSalary()
        return baseSalary + bonus;
    }
}
```

Qui `getSalary()` si riferisce al metodo `getSalary` applicato al parametro implicito del metodo in esecuzione, che è di tipo `Manager`: la classe `Manager` ha un metodo `getSalary`, che viene quindi invocato. Questa invocazione è ricorsiva e non terminerà mai:

bisogna, invece, dire al compilatore che il metodo `getSalary` da invocare è quello della superclasse.

Quando in una sottoclasse invocate un metodo della superclasse all'interno di un metodo avente lo stesso nome, dovete ricordarvi di usare la parola riservata `super`.



## Argomenti avanzati 9.1

### Invoke il costruttore della superclasse

Un costruttore di una sottoclasse  
invoca il costruttore della superclasse  
senza parametri, a meno che non ci  
sia una diversa indicazione esplicita.

Per invocare un costruttore  
della superclasse in un costruttore  
di una sottoclasse si usa la parola  
riservata `super` come primo  
enunciato.

Un costruttore di una sottoclasse  
può fornire argomenti a un costruttore  
della superclasse usando la parola  
riservata `super`.

Prendiamo in esame la procedura di costruzione di un oggetto di una sottoclasse. Un costruttore di una sottoclasse può inizializzare soltanto le variabili di esemplare definite nella sottoclasse stessa, ma anche le variabili di esemplare della superclasse devono essere inizializzate: se non si fornisce esplicitamente un'indicazione diversa, le variabili di esemplare della superclasse vengono inizializzate invocando il costruttore della superclasse privo di parametri.

Per specificare, invece, che si vuole invocare un costruttore diverso, come *primo enunciato* del costruttore della sottoclasse bisogna usare la parola riservata `super`, seguita dai parametri da fornire al costruttore della superclasse, indicati fra parentesi tonde.

Supponiamo, ad esempio, che la superclasse `Question` abbia un costruttore che inizializza il testo della domanda. Ecco come un costruttore di una sottoclasse potrebbe invocare quel costruttore della superclasse:

```
public ChoiceQuestion(String questionText)
{
    super(questionText);
    choices = new ArrayList<String>();
}
```

Nel nostro programma d'esempio abbia usato implicitamente il costruttore della superclasse privo di parametri. Se, però, tutti i costruttori della superclasse hanno parametri, bisogna necessariamente usare la sintassi con `super` e fornire gli argomenti richiesti da uno di tali costruttori.

Quando la parola riservata `super` è seguita da una coppia di parentesi indica l'invocazione del costruttore della superclasse e deve essere *il primo enunciato nel costruttore della sottoclasse*. Se, invece, `super` è seguita da un punto e da un nome di metodo, indica l'invocazione di un metodo della superclasse, come avete visto nel paragrafo precedente: questo può avvenire in qualsiasi punto di qualunque metodo di una sottoclasse.

## 9.4 Polimorfismo

In questo paragrafo vedrete come si possa usare l'ereditarietà per elaborare oggetti di tipi diversi in uno stesso programma.

Consideriamo il primo programma usato come esempio in questo capitolo: presentava all'utente la domanda contenuta in un oggetto di tipo `Question`. Il secondo programma, invece, presentava due domande di tipo `ChoiceQuestion`. Siamo in grado di scrivere un programma che presenti all'utente un questionario con domande dei due tipi?

## Sintassi di Java

### 9.3 Invocazione di un costruttore della superclasse

#### Sintassi

```
public NomeClasse(TipoDiParametro nomeParametro, ...)
{
    super(parametri);
    ...
}
```

#### Esempio

Per prima cosa invoca il costruttore della superclasse.

Il corpo del costruttore può contenere altri enunciati.

```
public ChoiceQuestion(String questionText)
{
    super(questionText);
    choices = new ArrayList<String>;
```

Se non c'è questa riga, viene invocato il costruttore della superclasse privo di parametri.

Con l'ereditarietà questo obiettivo è facile da raggiungere. Per presentare una domanda all'utente non abbiamo bisogno di sapere con precisione di quale tipo sia la domanda stessa: semplicemente, visualizziamo la domanda e verifichiamo se l'utente ha fornito la risposta corretta. La superclasse `Question` mette a disposizione metodi che realizzano entrambi questi compiti, quindi possiamo definire la seguente funzione `presentQuestion`, che si aspetta di ricevere un oggetto di tipo `Question`:

```
public static void presentQuestion(Question q)
{
    q.display();
    System.out.print("Your answer: ");
    Scanner in = new Scanner(System.in);
    String response = in.nextLine();
    System.out.println(q.checkAnswer(response));
}
```

Un riferimento a una sottoclasse può essere utilizzato in ogni punto del programma che preveda la presenza di un riferimento alla sua superclasse.

Come detto nel Paragrafo 9.1, possiamo fornire un oggetto di una sottoclasse ogni volta che è previsto un oggetto di una superclasse:

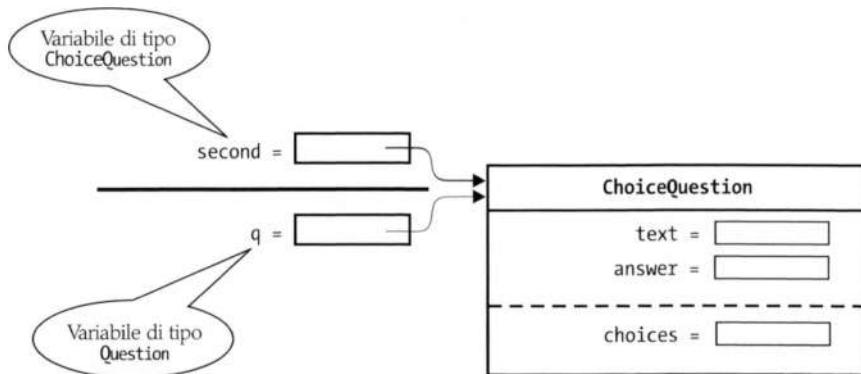
```
ChoiceQuestion second = new ChoiceQuestion();
...
presentQuestion(second); // si può passare un oggetto ChoiceQuestion
```

Quando viene eseguito il metodo `presentQuestion`, i riferimenti memorizzati nelle variabili `second` e `q` puntano al medesimo oggetto, di tipo `ChoiceQuestion` (Figura 7), però la variabile `q` non conosce il tipo effettivo dell'oggetto a cui fa riferimento (Figura 8).

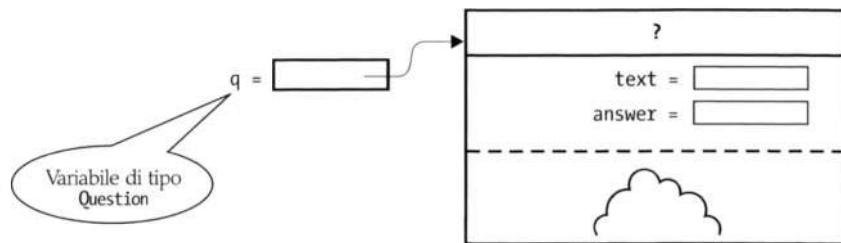
Dato che `q` è una variabile di tipo `Question`, la si può usare per invocare i metodi `display` e `checkAnswer`, ma non il metodo `addChoice`, perché non è un metodo della superclasse `Question`.

**Figura 7**

Variabili di tipi diversi che fanno riferimento al medesimo oggetto

**Figura 8**

Un riferimento di tipo `Question` può puntare a un oggetto di qualsiasi sottoclasse di `Question`



In effetti è giusto che sia così: dopo tutto, in questa invocazione del metodo `presentQuestion` la variabile `q` fa riferimento a un oggetto `ChoiceQuestion`, ma in un'altra invocazione `q` potrebbe fare riferimento a un semplice oggetto `Question` oppure a un oggetto di una diversa sottoclasse di `Question`.

Ora analizziamo più in dettaglio il metodo `presentQuestion`, che inizia con questa invocazione:

```
q.display(); // invoca Question.display oppure ChoiceQuestion.display ?
```

**Quando la macchina virtuale invoca un metodo di esemplare, usa il metodo della classe a cui appartiene il parametro implicito: si parla di "ricerca dinamica del metodo".**

**Il polimorfismo ("avere molte forme") ci consente di manipolare oggetti che hanno in comune alcune funzionalità, anche se queste sono implementate in modi diversi.**

Quale dei due metodi `display` viene invocato? Se esaminate, nel seguito, ciò che viene visualizzato dall'esecuzione del programma che stiamo progettando, vedrete che il metodo invocato dipende dal contenuto della variabile parametro `q`. Nel primo caso `q` si riferisce a un oggetto `Question`, per cui viene invocato il metodo `Question.display`, ma nel secondo caso `q` si riferisce a un oggetto `ChoiceQuestion`, per cui viene invocato il metodo `ChoiceQuestion.display`, che visualizza l'elenco delle opzioni.

In Java, l'invocazione di un metodo viene sempre determinata dal tipo effettivo dell'oggetto con cui il metodo viene invocato, non dal tipo della variabile che contiene il riferimento all'oggetto: si parla di **ricerca dinamica del metodo** (*dynamic method lookup*).

La ricerca dinamica del metodo da eseguire ci consente di manipolare in modo omogeneo oggetti che sono esemplari di classi diverse, una caratteristica che si chiama **polimorfismo** (dal greco "multiforme"): chiediamo a più oggetti di assolvere a un determinato compito e ciascuno lo fa a modo suo.

Il polimorfismo agevola l'evoluzione e l'estensione dei programmi. Immaginate di voler utilizzare un nuovo tipo di domanda che richiede di effettuare un calcolo, per il quale si vuole accettare anche una risposta approssimata. Tutto ciò che dobbiamo definire è una nuova classe, `NumericQuestion`, che estenda `Question` e che abbia il proprio metodo `checkAnswer`. A questo punto possiamo invocare la funzione `presentQuestion` più volte, con un assortimento di domande normali, domande con opzioni a scelta singola e domande con risposta di tipo numerico. Non c'è alcuna modifica da fare alla funzione `presentQuestion!` Grazie alla ricerca dinamica dei metodi, le invocazioni dei metodi `display` e `checkAnswer` selezionano automaticamente il metodo della classe giusta.

### File QuestionDemo3.java

```
1 import java.util.Scanner;
2
3 /**
4  * Questo programma mostra un questionario con due tipi di domande.
5 */
6 public class QuestionDemo3
7 {
8     public static void main(String[] args)
9     {
10         Question first = new Question();
11         first.setText("Who was the inventor of Java?");
12         first.setAnswer("James Gosling");
13
14         ChoiceQuestion second = new ChoiceQuestion();
15         second.setText("In which country was the inventor of Java born?");
16         second.addChoice("Australia", false);
17         second.addChoice("Canada", true);
18         second.addChoice("Denmark", false);
19         second.addChoice("United States", false);
20
21         presentQuestion(first);
22         presentQuestion(second);
23     }
24
25 /**
26  * Presenta una domanda all'utente e ne verifica la risposta.
27  * @param q la domanda
28 */
29 public static void presentQuestion(Question q)
30 {
31     q.display();
32     System.out.print("Your answer: ");
33     Scanner in = new Scanner(System.in);
34     String response = in.nextLine();
35     System.out.println(q.checkAnswer(response));
36 }
37 }
```

### Esecuzione del programma

```
Who was the inventor of Java?
Your answer: Bjarne Stroustrup
false
```



In which country was the inventor of Java born?

- 1: Australia
  - 2: Canada
  - 3: Denmark
  - 4: United States
- Your answer: 2  
true



## Auto-valutazione

16. Nell'ipotesi che `SavingsAccount` sia una sottoclasse di `BankAccount`, quale dei seguenti enunciati è valido in Java?
  - a. `BankAccount account = new SavingsAccount();`
  - b. `SavingsAccount account2 = new BankAccount();`
  - c. `BankAccount account = null;`
  - d. `SavingsAccount account2 = account;`
17. Se `account` è una variabile di tipo `BankAccount` che contiene un riferimento diverso da `null`, che informazioni abbiamo in merito all'oggetto a cui si riferisce?
18. Dichiaret un array, `quiz`, che sia in grado di contenere sia oggetti `Question` sia oggetti `ChoiceQuestion`.
19. Quale metodo viene effettivamente invocato nel seguente frammento di codice?  
`ChoiceQuestion cq = . . .; // un valore diverso da null`  
`cq.display();`
20. L'invocazione `Math.sqrt(2)` richiede una ricerca dinamica del metodo da eseguire?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R9.7, E9.7 e P9.17, al termine del capitolo.



## Argomenti avanzati 9.2

### Ricerca dinamica e parametro implicito

Immaginiamo di aver aggiunto il metodo `presentQuestion` alla classe `Question`:

```
public void presentQuestion()
{
    display();
    System.out.print("Your answer: ");
    Scanner in = new Scanner(System.in);
    String response = in.nextLine();
    System.out.println(checkAnswer(response));
}
```

e analizziamo l'invocazione seguente:

```
ChoiceQuestion cq = new ChoiceQuestion();
cq.setText("In which country was the inventor of Java born?");
. .
cq.presentQuestion();
```

Quale metodo `display` e quale metodo `checkAnswer` verranno invocati all'interno del metodo `presentQuestion`? Se osservate il codice del metodo `presentQuestion`, noterete che tali metodi vengono eseguiti usando il parametro implicito:

```
public class Question
{
    ...
    public void presentQuestion()
    {
        this.display();
        System.out.print("Your answer: ");
        Scanner in = new Scanner(System.in);
        String response = in.nextLine();
        System.out.println(this.checkAnswer(response));
    }
}
```

Nella nostra invocazione, il parametro implicito `this` è un riferimento a un oggetto di tipo `ChoiceQuestion`. Per effetto della ricerca dinamica dei metodi, verranno automaticamente invocate le versioni dei metodi `display` e `checkAnswer` definite nella classe `ChoiceQuestion`. Tutto ciò accade anche se il metodo `presentQuestion` è stato dichiarato nella classe `Question`, che *non ha alcuna consapevolezza* nemmeno dell'esistenza della classe `ChoiceQuestion`!

Come potete vedere, il polimorfismo è un meccanismo davvero molto potente. La classe `Question` rende disponibile un metodo, `presentQuestion`, che specifica gli elementi comuni che riguardano la visualizzazione di una domanda, cioè il fatto di visualizzarne il testo e di acquisire la risposta dell'utente, verificandone la correttezza. Come si faccia, in concreto, a visualizzare un particolare tipo di domanda e a verificare la correttezza della corrispondente risposta è un compito che viene demandato alle sottoclassi.



## Argomenti avanzati 9.3

### Classi astratte

Quando estendete una classe esistente, potete scegliere se ridefinire o meno i metodi della superclasse, ma, talvolta, si vogliono *obbligare* i programmatore a ridefinire un metodo. Questo avviene quando non esiste una buona soluzione predefinita da realizzare nella superclasse e solo i programmatore della sottoclasse possono sapere come implementare il metodo nel modo appropriato.

Ecco un esempio. Immaginate che la “First National Bank of Java” decida che ciascun tipo di conto bancario debba prevedere il pagamento di una commissione mensile. Di conseguenza, aggiungiamo alla classe `Account` il metodo `deductFees`, che abbia il compito di prelevare tale commissione:

```
public class Account
{
    ...
    public void deductFees() { . . . }
    ...
}
```

Che cosa dovrebbe fare questo metodo? Potremmo naturalmente scrivere un metodo che non fa nulla, ma un programmatore che progetta una nuova sottoclasse potrebbe

dimenticarsi di sovrascrivere il metodo `deductFees` e, di conseguenza, il nuovo tipo di conto erediterebbe dalla superclasse il metodo che non fa nulla. Esiste una strategia migliore: dichiarare che `deductFees` è un **metodo astratto** (*abstract method*), in questo modo

```
public abstract void deductFees();
```

Un metodo astratto non ha implementazione, obbligando così i programmati di sottoclassi a specificare implementazioni concrete per questo metodo (naturalmente in alcune sottoclassi si può decidere di implementare un metodo che non fa nulla, ma si tratta di una scelta deliberata, non di un'impostazione predefinita che viene ereditata silenziosamente).

Non potete costruire oggetti di classi aventi metodi astratti. Per esempio, se la classe `Account` ha un metodo astratto, il compilatore segnala un errore nel momento in cui si tenta di crearne un esemplare, con `new Account()`.

Una classe di cui non potete costruire esemplari è detta **classe astratta**, mentre una classe in cui potete farlo viene detta talvolta **classe concreta**. In Java, tutte le classi astratte devono essere dichiarate con la parola riservata `abstract`:

```
public abstract class Account
{
    public abstract void deductFees();
    ...
}

public class SavingsAccount extends Account // classe non astratta
{
    ...
    public void deductFees() // implementazione
    {
        ...
    }
}
```

Se una classe estende una classe astratta senza fornire un'implementazione di tutti i metodi astratti è anch'essa astratta.

```
public abstract class BusinessAccount extends Account
{
    ...
    // nessuna implementazione di deductFees
}
```

Notate che non potete costruire un *oggetto* che sia esemplare di una classe astratta, ma potete sempre usare un *riferimento* il cui tipo sia una classe astratta. Naturalmente, l'oggetto effettivo a cui si riferisce deve essere un esemplare di una sottoclasse concreta:

```
Account anAccount; // corretto
anAccount = new Account(); // Errore: Account è una classe astratta
anAccount = new SavingsAccount(); // corretto
anAccount = null; // corretto
```

Le classi astratte servono a obbligare i programmati a creare sottoclassi: dichiarando che alcuni metodi sono astratti, si evita di avere metodi predefiniti inutili, che potrebbero essere ereditati per errore.



## Argomenti avanzati 9.4

### Metodi e classi final

In Argomenti avanzati 9.3 avete visto come potete obbligare altri programmati che creano sottoclassi di classi astratte a sovrascriverne i metodi astratti. Occasionalmente, potreste desiderare il contrario e voler *impedire* la creazione di sottoclassi o la sovrascrittura di determinati metodi: usate la parola riservata `final`. Per esempio, nella libreria standard di Java la classe `String` è stata dichiarata in questo modo:

```
public final class String { . . . }
```

Questo significa che nessuno può estendere la classe `String` e, di conseguenza, una variabile di tipo `String` deve contenere il riferimento a un oggetto di tipo `String`, non di una sua sottoclasse (che non può esistere).

Potete dichiarare `final` anche singoli metodi:

```
public class SecureAccount extends BankAccount
{
    . .
    public final boolean checkPassword(String password)
    {
        .
    }
}
```

In questo modo nessuno potrà sovrscrivere il metodo `checkPassword` con un altro metodo che restituisca semplicemente `true`.



## Argomenti avanzati 9.5

### Accesso protetto

Nel tentativo di realizzare il metodo `display` della classe `ChoiceQuestion` abbiamo incontrato una serie di difficoltà, perché tale metodo aveva bisogno di accedere alla variabile di esemplare `text` della superclasse: la nostra soluzione ha usato i metodi appropriati della superclasse per visualizzare il testo della domanda.

Java offre un'altra possibilità per risolvere questo problema. La superclasse può dichiarare una variabile di esemplare *protetta*:

```
public class Question
{
    protected String text;
    .
}
```

Ai dati protetti di un oggetto si può accedere dai metodi della sua classe e di tutte le sottoclassi di questa. Per esempio, la classe `ChoiceQuestion` eredita da `Question`, quindi i suoi metodi possono accedere alle variabili di esemplare `protected` della superclasse `Question`.

Ad alcuni programmati piace la modalità di accesso `protected`, perché sembra un compromesso fra la protezione assoluta, rendendo private tutte le variabili di esemplare, e la totale mancanza di protezione, rendendole tutte pubbliche. Tuttavia, l'esperienza ha dimostrato che le variabili di esemplare protette sono soggette allo stesso tipo di problemi che affliggono quelle pubbliche. Chi progetta la superclasse non può controllare gli autori delle sottoclassi e qualunque metodo di sottoclasse può alterare i dati della superclasse. Inoltre, è difficile modificare le classi con variabili protette: se l'autore della superclasse volesse cambiare l'implementazione dei dati, non potrebbe modificare le variabili protette, perché qualcuno, da qualche parte, potrebbe avere scritto una sottoclasse il cui codice dipende da esse.

Le variabili protette, in Java, hanno un altro svantaggio: sono accessibili non soltanto dalle sottoclassi, ma anche dalle altre classi che si trovano nello stesso pacchetto (Argomenti avanzati 8.4).

È meglio fare in modo che tutti i dati siano privati. Se volete che soltanto i metodi di una sottoclasse abbiano accesso ai dati, valutate la possibilità di dichiarare protetto un metodo *di ispezione*.



## Consigli pratici 9.1

### Progettare una gerarchia di ereditarietà

Quando manipolerete un insieme di classi, alcune delle quali sono più generiche e altre più specifiche, probabilmente vorrete organizzarle in una gerarchia di ereditarietà, perché questo vi consentirà di elaborare in modo omogeneo oggetti di classi diverse.

Come esempio, consideriamo una banca che offre ai propri clienti i seguenti tipi di conti:

- Un conto di risparmio (*savings account*) che matura interessi attivi. Gli interessi maturano mensilmente e vengono calcolati sulla base del saldo minimo registrato durante il mese.
- Un conto corrente (*checking account*) che non matura interessi, consente tre prelievi mensili senza commissione e addebita una commissione (*fee*) di \$1 per ogni ulteriore prelievo.

**Problema.** Vogliamo realizzare un programma che dovrà gestire un insieme di conti bancari di entrambi i tipi, strutturato in modo che vi si possano aggiungere altri tipi di conto senza dover modificare il ciclo di gestione principale. L'utente deve avere a disposizione un menu come questo:

```
D)eposit W)ithdraw M)onth end Q)uit
```

Nel caso di un versamento (*deposit*) o di un prelievo (*withdraw*), il programma deve chiedere all'utente il numero del conto e l'importo dell'operazione. Dopo ogni transazione il programma visualizza il saldo del conto interessato.

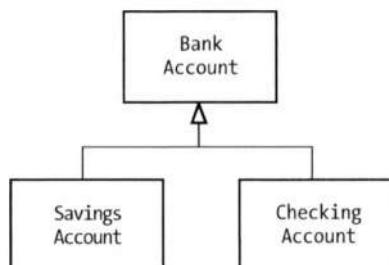
Quando viene selezionata la voce “Month end” (cioè *fine del mese*), il programma agisce su tutti i conti e accredita gli interessi o preleva le commissioni, in relazione al tipo di conto su cui opera, poi visualizza il saldo di tutti i conti. L’opzione *quit*, ovviamente, termina l’esecuzione del programma.

**Fase 1** Elencate le classi che fanno parte della gerarchia.

Nel nostro caso la descrizione del problema cita due classi: *SavingsAccount* e *CheckingAccount*. Ovviamente potremmo realizzarle separatamente, ma questa non sarebbe una buona idea, perché nelle due classi dovremmo ripetere il codice relativo alle funzionalità comuni, come l’aggiornamento del saldo di un conto: abbiamo bisogno di un’altra classe che si accolla le responsabilità comuni, anche se la descrizione del problema non la menziona in modo esplicito. Dobbiamo, quindi, scoprirla autonomamente, anche se in questo caso la soluzione è semplice: i conti di risparmio e i conti correnti sono casi particolari di conti bancari, quindi aggiungiamo la superclasse comune *BankAccount*.

**Fase 2** Organizzate le classi in una gerarchia di ereditarietà.

Disegnate un diagramma di ereditarietà che evidensi superclassi e sottoclassi. Ecco quello relativo al nostro esempio:



**Fase 3** Individuate le responsabilità comuni.

Nella Fase 2 avrete identificato la classe su cui si basa l’intera gerarchia: questa deve avere responsabilità sufficienti per portare a termine i vari compiti che caratterizzano il problema. Per scoprire quali siano questi compiti, scrivete lo pseudocodice che serve per l’elaborazione degli oggetti.

Per ogni comando dell’utente  
 Se è un versamento o un prelievo  
     Versa o preleva l’importo usando il conto specificato.  
     Visualizza il saldo del conto coinvolto.  
 Se è “fine mese”  
     Per ogni conto  
         Esegui l’elaborazione di fine mese.  
         Visualizza il saldo.

Esaminando lo pseudocodice, possiamo compilare la seguente lista di responsabilità comuni, che riguardano qualsiasi conto bancario:

Versare denaro.  
 Prelevare denaro.  
 Ispezionare il saldo.  
 Eseguire l'elaborazione di fine mese.

**Fase 4** Decidete quali metodi vanno sovrascritti nelle sottoclassi.

Per ogni sottoclasse e per ciascuna delle responsabilità comuni individuate nella Fase 3 bisogna decidere se il comportamento possa essere ereditato oppure se debba essere sovrascritto. Ricordatevi di definire nella classe che sta alla base della gerarchia tutti i metodi che dovranno essere ereditati o sovrascritti.

```
public class BankAccount
{
    . . .
    /**
     * Effettua un versamento in questo conto.
     * @param amount l'importo da versare
    */
    public void deposit(double amount) { . . . }

    /**
     * Effettua un prelievo da questo conto.
     * @param amount l'importo da prelevare
    */
    public void withdraw(double amount) { . . . }

    /**
     * Porta a termine l'elaborazione di fine mese,
     * se appropriata per questo conto.
    */
    public void monthEnd() { . . . }

    /**
     * Ispeziona il saldo attuale di questo conto bancario.
     * @return il saldo attuale
    */
    public double getBalance() { . . . }
}
```

Entrambe le classi, `SavingsAccount` e `CheckingAccount`, sovrascrivono il metodo `monthEnd`. La classe `SavingsAccount` deve anche sovrascrivere il metodo `withdraw`, per tenere traccia del saldo minimo durante il mese; la classe `CheckingAccount`, invece, all'interno del metodo `withdraw` deve aggiornare il contatore di transazioni.

**Fase 5** Definite l'interfaccia pubblica di ciascuna sottoclasse.

Solitamente le sottoclassi hanno responsabilità diverse da quelle della superclasse: elencatele, insieme ai metodi che devono sovrascrivere. Inoltre, dovete specificare come si costruiscono gli oggetti delle sottoclassi.

In questo esempio ci serve un modo per impostare il tasso di interesse nei conti di risparmio. Inoltre, dobbiamo specificare i costruttori e i metodi sovrascritti.

```

public class SavingsAccount extends BankAccount
{
    ...
    /**
     * Costruisce un conto di risparmio con saldo uguale a zero.
     */
    public SavingsAccount() { . . . }

    /**
     * Imposta il tasso d'interesse per questo conto.
     * @param rate il tasso d'interesse mensile, in percentuale
     */
    public void setInterestRate(double rate) { . . . }

    // questi metodi sovrascrivono i corrispondenti metodi della superclasse
    public void withdraw(double amount) { . . . }
    public void monthEnd() { . . . }
}

public class CheckingAccount extends BankAccount
{
    ...
    /**
     * Costruisce un conto corrente con saldo uguale a zero.
     */
    public CheckingAccount () { . . . }

    // questi metodi sovrascrivono i corrispondenti metodi della superclasse
    public void withdraw(double amount) { . . . }
    public void monthEnd() { . . . }
}

```

**Fase 6** Identificate le variabili di esemplare.

Elencate le variabili di esemplare di ciascuna classe. Se trovate una variabile di esemplare comune a tutte le classi, definitela nella classe che sta alla base della gerarchia.

Tutti i conti bancari hanno un saldo, quindi definiamo nella superclasse `BankAccount` la variabile di esemplare `balance`:

```

public class BankAccount
{
    private double balance;
    ...
}

```

La classe `SavingsAccount` ha bisogno di memorizzare il tasso di interesse e il saldo minimo durante il mese, che viene aggiornato da tutti i prelievi.

```

public class SavingsAccount extends BankAccount
{
    private double interestRate;
    private double minBalance;
    ...
}

```

La classe `CheckingAccount` deve contare le operazioni di prelievo, in modo da poter applicare le commissioni dovute quando il loro numero supera il limite di prelievi gratuiti.

```
public class CheckingAccount extends BankAccount
{
    private int withdrawals;
    . . .
}
```

**Fase 7** Implementate costruttori e metodi.

I metodi della classe `BankAccount` aggiornano o restituiscono il saldo.

```
public void deposit(double amount)
{
    balance = balance + amount;
}

public void withdraw(double amount)
{
    balance = balance - amount;
}

public double getBalance()
{
    return balance;
}
```

Progettando la superclasse `BankAccount` non possiamo ipotizzare quale sarà l'elaborazione da svolgere a fine mese: decidiamo di definire un metodo che non faccia nulla:

```
public void monthEnd()
{
    . . .
}
```

Nel metodo `withdraw` della classe `SavingsAccount` viene aggiornato il saldo minimo raggiunto durante il mese. Si noti l'invocazione dell'omonimo metodo della superclasse:

```
public void withdraw(double amount)
{
    super.withdraw(amount);
    double balance = getBalance();
    if (balance < minBalance)
    {
        minBalance = balance;
    }
}
```

Nel metodo `monthEnd` della classe `SavingsAccount` l'interesse maturato viene accreditato sul conto: dobbiamo invocare il metodo `deposit`, perché non abbiamo accesso diretto alla

variabile di esemplare `balance`. Infine, il saldo minimo raggiunto nel mese viene reso uguale al saldo attuale, per predisporre l'elaborazione per il mese successivo.

```
public void monthEnd()
{
    double interest = minBalance * interestRate / 100;
    deposit(interest);
    minBalance = getBalance();
}
```

Il metodo `withdraw` della classe `CheckingAccount` deve ispezionare il contatore dei prelievi: se è stato superato il limite di quelli gratuiti, viene addebitata una commissione. Anche in questo caso il metodo deve invocare l'omonimo metodo della superclasse:

```
public void withdraw(double amount)
{
    final int FREE_WITHDRAWALS = 3;
    final int WITHDRAWAL_FEE = 1;

    super.withdraw(amount);
    withdrawals++;
    if (withdrawals > FREE_WITHDRAWALS)
    {
        super.withdraw(WITHDRAWAL_FEE);
    }
}
```

L'elaborazione svolta a fine mese da un conto corrente consiste semplicemente nell'azzeramento del contatore di prelievi:

```
public void monthEnd()
{
    withdrawals = 0;
}
```

**Fase 8** Costruite oggetti di classi diverse e realizzate un collaudo.

Nel nostro programma d'esempio creiamo cinque conti correnti e cinque conti di risparmio, memorizzando i loro riferimenti in una lista di conti bancari. Poi, eseguiamo un ciclo che accetta comandi dall'utente per effettuare versamenti, prelievi ed elaborazioni di fine mese.

```
BankAccount[] accounts = . . .;
. . .
Scanner in = new Scanner(System.in);
boolean done = false;
while (!done)
{
    System.out.print("D)eposit W)ithdraw M)onth end Q)uit: ");
    String input = in.next();
    if (input.equals("D") || input.equals("W")) // versamento o prelievo
    {
        System.out.print("Enter account number and amount: ");
        int num = in.nextInt();
```

```

        double amount = in.nextDouble();

        if (input.equals("D")) { accounts[num].deposit(amount); }
        else { accounts[num].withdraw(amount); }

        System.out.println("Balance: " + accounts[num].getBalance());
    }
    else if (input.equals("M")) // elaborazione di fine mese
    {
        for (int n = 0; n < accounts.length; n++)
        {
            accounts[n].monthEnd();
            System.out.println(n + " " + accounts[n].getBalance());
        }
    }
    else if (input.equals("Q"))
    {
        done = true;
    }
}

```



## Esempi completi 9.1

### Progettare una gerarchia di dipendenti per l'elaborazione delle buste paga

**Problema.** Vogliamo implementare l'elaborazione delle buste paga per diverse categorie di dipendenti:

- Dipendente “a ore” (*hourly employee*), viene retribuito con una paga oraria, ma, se lavora più di 40 ore in una settimana, le ore eccedenti vengono pagate “una volta e mezza”.
- Dipendente salariato (*salaried employee*), viene retribuito con un salario settimanale, indipendentemente dalle ore lavorate.
- Dirigente (*manager*), è un dipendente salariato, al quale è dovuto anche un premio di produttività (*bonus*).

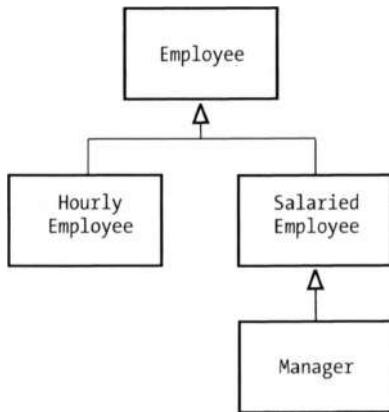
Il programma dovrà calcolare la retribuzione dovuta a un insieme di dipendenti. Per ogni dipendente, chiede all’utente il numero di ore lavorate in una determinata settimana e visualizza l’importo da pagare.

**Fase 1** Elencate le classi che fanno parte della gerarchia.

La descrizione del problema elenca tre classi: `HourlyEmployee`, `SalariedEmployee` e `Manager`. Inoltre, abbiamo bisogno di una classe che descriva le caratteristiche comuni alle tre categorie di dipendenti: `Employee`.

**Fase 2** Organizzate le classi in una gerarchia di ereditarietà.

Ecco il diagramma di ereditarietà per le nostre classi:



**Fase 3** Individuate le responsabilità comuni.

Per scoprire le responsabilità comuni, scriviamo lo pseudocodice che serve per l'elaborazione degli oggetti.

Per ogni dipendente

- Visualizza il nome del dipendente.
- Acquisisci il numero di ore lavorate.
- Calcola la paga dovuta per quelle ore.

Esaminando lo pseudocodice possiamo compilare la seguente lista di responsabilità comuni, che riguardano qualsiasi dipendente e, quindi, saranno attribuite alla classe `Employee`:

Ispezionare il nome.

Calcolare la paga dovuta per un dato numero di ore.

**Fase 4** Decidete quali metodi vanno sovrascritti nelle sottoclassi.

In questo esempio non c'è bisogno di alcuna variante nell'ispezione del nome del dipendente, ma la paga dovuta va calcolata in modo diverso da ciascuna sottoclasse, quindi il metodo `weeklyPay` sarà sovrascritto da tutte:

```

/**
 * Un dipendente ha un nome e una strategia
 * per calcolare la paga settimanale.
 */
public class Employee {
    ...
    /**
     * Ispeziona il nome di questo dipendente.
     * @return il nome
     */
    public String getName() { . . . }

    /**
     *
  
```

```

    Calcola l'importo dovuto per una settimana di lavoro.
    @param hoursWorked il numero di ore lavorate nella settimana
    @return l'importo dovuto per la settimana di lavoro
*/
public double weeklyPay(int hoursWorked) { . . . }
}

```

**Fase 5** Definite l'interfaccia pubblica di ciascuna sottoclasse.

Costruiremo dipendenti fornendo il nome e le informazioni relative al salario.

```

public class HourlyEmployee extends Employee
{
    . . .
    /**
     * Costruisce un impiegato a paga oraria
     * con un dato nome e un dato compenso orario.
    */
    public HourlyEmployee(String name, double wage) { . . . }
}

public class SalariedEmployee extends Employee
{
    . . .
    /**
     * Costruisce un impiegato salariato
     * con un dato nome e un dato salario annuo.
    */
    public SalariedEmployee(String name, double salary) { . . . }
}

public class Manager extends SalariedEmployee
{
    . . .
    /**
     * Costruisce un dirigente con un dato nome,
     * un dato salario annuo e un dato bonus settimanale.
    */
    public Manager(String name, double salary, double bonus) { . . . }
}

```

Questi costruttori devono assegnare il nome alla propria porzione di oggetto di tipo `Employee`, quindi definiamo nella classe `Employee` il metodo `setName`:

```

public class Employee
{
    . . .
    public void setName(String employeeName) { . . . }
}

```

Poi, naturalmente, ciascuna sottoclasse deve definire il metodo che calcola la paga settimanale:

```
// questo metodo sovrascrive l'omonimo metodo della superclasse
```

```
public double weeklyPay(int hoursWorked) { . . . }
```

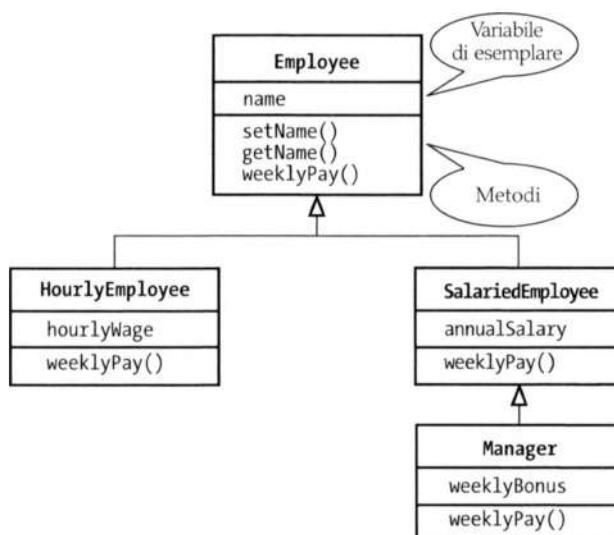
Vista la semplicità di questo esempio, non servono altri metodi.

**Fase 6** Identificate le variabili di esemplare.

Tutti i dipendenti hanno un nome, per cui la classe `Employee` deve avere la variabile di esemplare `name` (come si può vedere nella gerarchia aggiornata qui riportata).

Cosa possiamo dire in merito ai salari? I dipendenti "a ore" hanno una paga oraria, mentre i dipendenti salariati hanno un salario annuo. Anche se sarebbe possibile memorizzare entrambi questi valori in un'unica variabile di esemplare della superclasse, non sarebbe una buona idea: il codice risultante sarebbe complesso e sarebbe facile fare errori, perché dovrebbe dare un senso a quei numeri.

È preferibile che gli oggetti di tipo `HourlyEmployee` memorizzino la propria paga oraria, mentre gli oggetti di tipo `SalariedEmployee` memorizzeranno il proprio salario annuo. Gli oggetti di tipo `Manager`, poi, devono memorizzare anche il proprio premio di produzione settimanale.



**Fase 7** Implementate costruttori e metodi.

Nei costruttori delle sottoclassi dobbiamo ricordarci di assegnare un valore alle variabili di esemplare della superclasse.

```
public SalariedEmployee(String name, double salary)
{
    setName(name);
    annualSalary = salary;
}
```

In questo caso abbiamo invocato un metodo, ma nella sezione Argomenti avanzati 9.1 abbiamo visto come si possa invocare un costruttore della superclasse. Nel costruttore della classe `Manager` usiamo questa seconda tecnica:

```
public Manager(String name, double salary, double bonus)
{
    super(name, salary);
    weeklyBonus = bonus;
}
```

La paga settimanale può, poi, essere calcolata seguendo le istruzioni presenti nella descrizione del problema:

```
public class HourlyEmployee extends Employee
{
    ...
    public double weeklyPay(int hoursWorked)
    {
        double pay = hoursWorked * hourlyWage;
        if (hoursWorked > 40)
        {
            pay = pay + ((hoursWorked - 40) * 0.5) * hourlyWage;
        }
        return pay;
    }
}

public class SalariedEmployee extends Employee
{
    ...
    public double weeklyPay(int hoursWorked)
    {
        final int WEEKS_PER_YEAR = 52;
        return annualSalary / WEEKS_PER_YEAR;
    }
}
```

Nel caso della classe `Manager` dobbiamo invocare il metodo omonimo della superclasse `SalariedEmployee`:

```
public class Manager extends SalariedEmployee
{
    ...
    public double weeklyPay(int hoursWorked)
    {
        return super.weeklyPay(hoursWorked) + weeklyBonus;
    }
}
```

#### Fase 8 Costruite oggetti di classi diverse e realizzate un collaudo.

Nel nostro programma d'esempio popoliamo una lista di dipendenti e calcoliamo le paghe settimanali:

```
Employee[] staff = new Employee[3];
staff[0] = new HourlyEmployee("Morgan, Harry", 30);
staff[1] = new SalariedEmployee("Lin, Sally", 52000);
staff[2] = new Manager("Smith, Mary", 104000, 50);
```

```

Scanner in = new Scanner(System.in);
for (Employee e : staff)
{
    System.out.print("Hours worked by " + e.getName() + ": ");
    int hours = in.nextInt();
    System.out.println("Salary: " + e.weeklyPay(hours));
}

```

Nel pacchetto dei file scaricabili per questo libro, la cartella `worked_example_1` del Capitolo 9 contiene il codice sorgente completo delle classi.

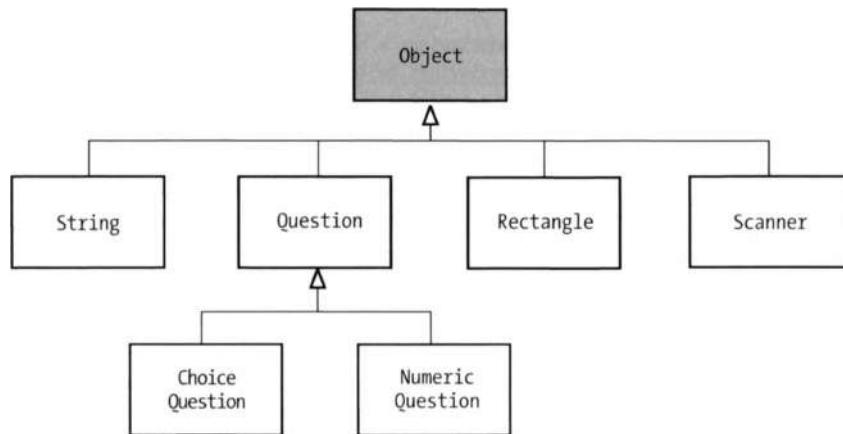
## 9.5 La superclasse universale: `Object`

In Java, ogni classe che venga dichiarata senza una esplicita clausola `extends` estende automaticamente la classe `Object`, quindi in Java la classe `Object` è la superclasse, diretta o indiretta, di *tutte* le classi (osservate la Figura 9). La classe `Object` definisce alcuni metodi molto generici, tra i quali citiamo:

- `toString`, che restituisce una stringa che descrive l'oggetto (Paragrafo 9.5.1);
- `equals`, che confronta tra loro due oggetti (Paragrafo 9.5.2);
- `hashCode`, che restituisce un codice numerico utile per memorizzare oggetti in un insieme (come si vedrà nella sezione Argomenti avanzati 14.1).

**Figura 9**

La classe `Object` è la superclasse di tutte le classi Java



### 9.5.1 Sovrascrivere il metodo `toString`

Il metodo `toString` restituisce una rappresentazione in forma di stringa per ciascun oggetto ed è spesso utilizzato durante il debugging.

Esaminiamo, come esempio, la classe `Rectangle` della libreria standard di Java. Il suo metodo `toString` restituisce una stringa che contiene lo stato del rettangolo:

```

Rectangle box = new Rectangle(5, 10, 20, 30);
String s = box.toString();
// s diventa "java.awt.Rectangle[x=5,y=10,width=20,height=30]"

```

Questo metodo `toString` viene invocato automaticamente tutte le volte che concatenate una stringa con un oggetto. Esamineate questa concatenazione:

```
"box=" + box;
```

A un lato dell'operatore di concatenazione `+` troviamo una stringa, ma all'altro lato c'è un riferimento a un oggetto. Il compilatore Java invoca automaticamente il metodo `toString` per "trasformare" tale oggetto in una stringa e, successivamente, le due stringhe vengono concatenate. In questo caso, il risultato è costituito da questa stringa:

```
"box=java.awt.Rectangle[x=5,y=10,width=20,height=30]"
```

Il compilatore può invocare il metodo `toString` perché sa che *tutti* gli oggetti hanno tale metodo, dal momento che tutte le classi estendono `Object`, che a sua volta definisce `toString`.

Come sapete, anche i numeri vengono convertiti in stringa quando li si concatena con una stringa. Per esempio:

```
int age = 18;
String s = "Harry's age is " + age;
// s diventa "Harry's age is 18"
```

In questo caso il metodo `toString` *non* è coinvolto, perché i numeri non sono oggetti e, quindi, non esiste un metodo `toString` per loro. L'insieme dei tipi fondamentali ha, però, ben pochi elementi e il compilatore sa come convertirli in stringhe.

Proviamo a usare il metodo `toString` con oggetti di tipo `BankAccount`:

```
BankAccount momSavings = new BankAccount(5000);
String s = momSavings.toString();
// s diventa simile a "BankAccount@d24606bf"
```

**TORNARE**

Il risultato è deludente: ciò che viene stampato è il nome della classe seguito dal **codice hash** (*hash code*) dell'oggetto, un valore che ci appare sostanzialmente casuale. Il codice hash può essere utilizzato per distinguere oggetti diversi, perché, come vedrete in maggiore dettaglio nella sezione Argomenti avanzati 14.1, è molto probabile che oggetti diversi abbiano un diverso codice hash.

A noi, però, il codice hash non interessa: noi vorremmo sapere cosa c'è *all'interno* dell'oggetto, ma, naturalmente, il metodo `toString` della classe `Object` non sa cosa ci sia all'interno della nostra classe `BankAccount`. Dobbiamo quindi sovrascrivere il metodo, fornendone la nostra versione personale nella classe `BankAccount`. Seguiremo lo stesso formato che usa il metodo `toString` della classe `Rectangle`: prima il nome della classe, poi i valori delle variabili di esemplare racchiusi fra parentesi quadre.

```
public class BankAccount
{
    ...
    public String toString()
    {
        return "BankAccount[balance=" + balance + "]";
    }
}
```

Sovrscrivete il metodo `toString` in modo che restituiscia una stringa che descrive lo stato dell'oggetto.

Ora funziona meglio:

```
BankAccount momSavings = new BankAccount(5000);
String s = momSavings.toString();
// s diventa "BankAccount[balance=5000]"
```

### 9.5.2 Il metodo equals

**Il metodo equals verifica se due oggetti hanno lo stesso contenuto.**

Oltre al metodo `toString`, la classe `Object` definisce anche il metodo `equals`, il cui compito è quello di verificare se due oggetti hanno lo stesso contenuto:

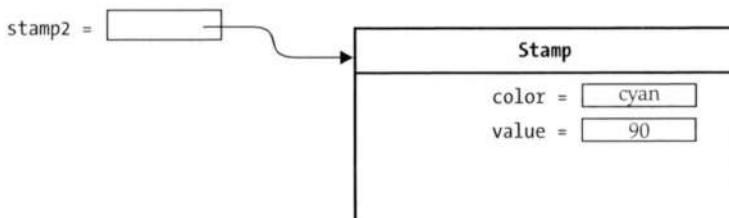
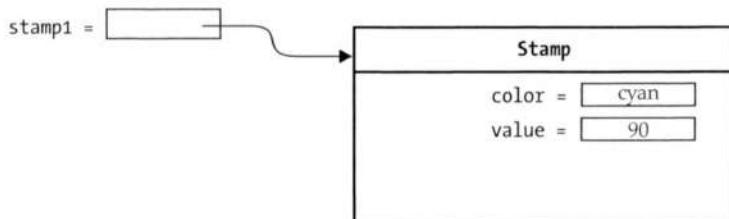
```
if (stamp1.equals(stamp2)) . . .
// hanno identico contenuto, come nella Figura 10
```

Quanto appena scritto è diverso dal confronto mediante l'operatore `==`, che verifica, invece, se due riferimenti puntano *al medesimo oggetto*:

```
if (stamp1 == stamp2) . . .
// sono riferimenti al medesimo oggetto, come nella Figura 11
```

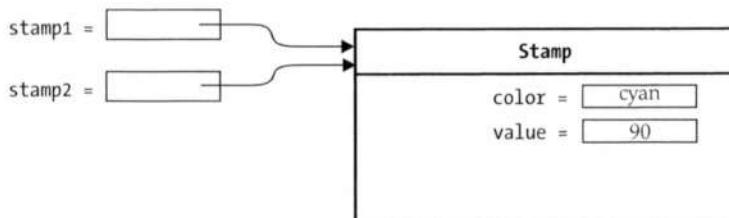
**Figura 10**

Due riferimenti a oggetti uguali



**Figura 11**

Due riferimenti al medesimo oggetto



Realizziamo il metodo `equals` nella classe `Stamp` (*francobollo*), sovrascrivendo il metodo `equals` della classe `Object`:

```

public class Stamp
{
    private String color;
    private int value;
    ...
    public boolean equals(Object otherObject)
    {
        ...
    }
    ...
}

```

A questo punto, però, abbiamo un piccolo problema: la classe `Object` non sa nulla di francobolli, quindi dichiara la variabile parametro `otherObject` del metodo `equals` in modo che sia di tipo `Object`, ma quando sovrascrivete un metodo non potete modificare il tipo dei suoi parametri. Per risolvere questo problema, eseguite un *cast* sulla variabile parametro, per farla diventare di tipo `Stamp`:

```
Stamp other = (Stamp) otherObject;
```

Ora potete confrontare i due francobolli:

```

public boolean equals(Object otherObject)
{
    Stamp other = (Stamp) otherObject;
    return color.equals(other.color) && value == other.value;
}

```

Osservate che questo metodo `equals` può accedere alle variabili di esemplare di *qualsiasi* oggetto di tipo `Stamp`: l'espressione `other.color` è perfettamente lecita.

### 9.5.3 L'operatore instanceof

Come avete visto, è possibile memorizzare un riferimento di tipo sottoclasse in una variabile di tipo superclasse:

```

ChoiceQuestion cq = new ChoiceQuestion();
Question q = cq; // va bene
Object obj = cq; // va bene

```

Più raramente dovrete eseguire la conversione inversa, assegnando un riferimento di tipo superclasse a una variabile di tipo sottoclasse.

Ad esempio, potrebbe esserci una variabile `obj` di tipo `Object`, ma potreste sapere con certezza che, in effetti, contiene un riferimento a un oggetto di tipo `Question`. In tal caso potreste usare un *cast* per cambiarne il tipo:

```
Question q = (Question) obj;
```

Se sapete che un oggetto è un esemplare di una determinata classe, potete usare un *cast* per convertirlo in quel tipo.

Questa conversione mediante *cast*, però, è in qualche modo pericolosa: se vi sbagliate e, invece, `obj` fa riferimento a un oggetto di tipo diverso, verrà lanciata un'eccezione di tipo "class cast", cioè un errore in fase di esecuzione relativo alla conversione tra classi.

## Sintassi di Java

### 9.4 L'operatore instanceof

#### Sintassi

*oggetto instanceof NomeTipo*

#### Esempio

Se *anObject* vale null, *instanceof* restituisce false.

Restituisce true se *anObject* può essere convertito in *Question*.

Con questa variabile si possono invocare metodi di *Question*.

```
if (anObject instanceof Question)
{
    Question q = (Question) anObject;
}
```

L'oggetto può appartenere a una sottoclasse di *Question*.

Due riferimenti al medesimo oggetto.

L'operatore *instanceof* verifica se un oggetto è di un determinato tipo.

Per evitare il rischio di un cast sbagliato, si può usare l'operatore *instanceof*, che verifica se un oggetto è di un determinato tipo. Ad esempio

```
obj instanceof Question
```

restituisce true se e solo se *obj* può essere convertito nel tipo *Question*, cosa che accade se *obj* fa effettivamente riferimento a un oggetto di tipo *Question* oppure a un oggetto di una sua sottoclasse, come *ChoiceQuestion*.

Usando l'operatore *instanceof* si può scrivere un cast sicuro, in questo modo:

```
if (obj instanceof Question)
{
    Question q = (Question) obj;
}
```

Si noti che *instanceof* non è un metodo, è un operatore, esattamente come + o <. Inoltre non opera su numeri né, in generale, su valori che non siano oggetti: alla sua sinistra deve esserci un riferimento a un oggetto, mentre alla sua destra ci deve essere il nome di un tipo di dato.

Non usate l'operatore *instanceof* per aggirare il polimorfismo:

```
if (q instanceof ChoiceQuestion) // non fate così! Errori comuni 9.5
{
    // elaborazione prevista per una domanda di tipo ChoiceQuestion
}
else if (q instanceof Question)
{
    // elaborazione prevista per una domanda di tipo Question
}
```

In una situazione come questa si deve realizzare un metodo nella classe `Question`, ad esempio `doTheTask`, per poi sovrascriverlo nella classe `ChoiceQuestion`; l'invocazione, poi, diventa:

```
q.doTheTask();
```



## Auto-valutazione

21. Perché l'invocazione `System.out.println(System.out)` visualizza un'informazione simile a `java.io.PrintStream@7a84e4`?
22. Il frammento di codice seguente viene compilato? Verrà poi eseguito? In caso di risposta negativa, quale errore viene segnalato?  

```
Object obj = "Hello";
System.out.println(obj.length());
```
23. Il frammento di codice seguente viene compilato? Verrà poi eseguito? In caso di risposta negativa, quale errore viene segnalato?  

```
Object obj = "Who was the inventor of Java?";
Question q = (Question) obj;
q.display();
```
24. Per quale motivo non memorizziamo semplicemente tutti gli oggetti in variabili di tipo `Object`?
25. Se `x` è un riferimento a un oggetto, che valore ha l'espressione `x instanceof Object`?

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi E9.10, E9.11 e E9.15, al termine del capitolo.



## Errori comuni 9.5

### Non usate verifiche per i tipi

Alcuni programmatore hanno l'abitudine di usare verifiche per i tipi, in modo da poter realizzare comportamenti variabili per ciascuna classe:

```
if (q instanceof ChoiceQuestion) // non fate così!
{
    // elaborazione prevista per una domanda di tipo ChoiceQuestion
}
else if (q instanceof Question)
{
    // elaborazione prevista per una domanda di tipo Question
}
```

È una strategia molto debole. Se aggiungete alla vostra gerarchia una nuova classe, come `NumericQuestion`, dovrete controllare con cura tutte le parti del vostro programma che contengono verifiche dei tipi di domande, aggiungendo un altro caso:

```
else if (q instanceof NumericQuestion)
{
```

```
// elaborazione prevista per una domanda di tipo NumericQuestion  
}
```

Pensiamo, invece, a cosa succede se aggiungiamo la classe `NumericQuestion` al nostro programma che gestisce questionari: *non dobbiamo fare alcuna modifica*, perché il programma usa il polimorfismo, non la verifica dei tipi.

Ogni volta che state per scrivere una verifica dei tipi all'interno di una gerarchia di classi, ripensate al vostro progetto e usate il polimorfismo. Dichiaret un metodo, ad esempio `doTheTask`, nella superclasse, per poi sovrascriverlo nelle sottoclassi; l'invocazione, poi, diventa:

```
q.doTheTask();
```



## Argomenti avanzati 9.6

### L'ereditarietà e il metodo `toString`

Avete già visto come scrivere un metodo `toString`: componete una stringa formata dal nome della classe e dalle coppie nome=valore delle variabili di esemplare, separate da virgolette. Se, però, volete che il vostro metodo `toString` sia utilizzabile dalle sottoclassi della vostra classe, dovete lavorare un po' di più.

Invece di scrivere esplicitamente nel metodo il nome della classe, dovreste invocare il metodo `getClass` (ereditato dalla classe `Object`) per ottenere un oggetto che descrive una classe e le sue proprietà. Quindi, invocatene il metodo `getName` per ottenere il nome della classe:

```
public String toString()  
{  
    return getClass().getName() + "[balance=" + balance + "]";  
}
```

In questo modo il metodo `toString` visualizza correttamente il nome della classe anche quando lo applicate a un esemplare di una sottoclasse, ad esempio `SavingsAccount`:

```
SavingsAccount momSavings = . . .;  
System.out.println(momSavings);  
// visualizza "SavingsAccount[balance=10000]"
```

Ovviamente dovreste sovrascrivere `toString` anche nella sottoclasse `SavingsAccount`, aggiungendo nomi e valori delle variabili di esemplare proprie della sottoclasse stessa. Notate che dovete invocare `super.toString` per ottenere le variabili di esemplare della superclasse, dato che la sottoclasse non vi può accedere direttamente:

```
public class SavingsAccount extends BankAccount  
{  
    . . .  
    public String toString()  
    {  
        return super.toString() + "[interestRate=" + interestRate + "]";  
    }  
}
```

In questo modo un conto bancario di risparmio viene convertito in una stringa del tipo `SavingsAccount[balance=10000][interestRate=5]` e le parentesi quadre indicano quali variabili di esemplare appartengono alla superclasse.



## Argomenti avanzati 9.7

### L'ereditarietà e il metodo equals

Avete appena visto come scrivere un metodo `equals`: eseguite un cast sulla variabile parametro `otherObject`, in modo da trasformarla nel tipo della vostra classe, quindi confrontate le variabili di esemplare del parametro implicito e del parametro esplicito convertito.

Ma cosa succede se qualcuno invoca `stamp1.equals(x)`, ma `x` non è un oggetto di tipo `Stamp`? Il cast errato genera un'eccezione e il programma termina bruscamente: è bene verificare che `otherObject` sia realmente un esemplare della classe `Stamp`. Il controllo più semplice si farebbe utilizzando l'operatore `instanceof`, ma tale verifica non è abbastanza specifica, perché darebbe esito positivo anche se `otherObject` appartenesse a una sottoclasse di `Stamp`. Per eliminare tale possibilità, occorre verificare che i due oggetti appartengano alla stessa classe. In caso contrario, restituite `false`.

```
if (getClass() != otherObject.getClass()) { return false; }
```

Inoltre, le specifiche del linguaggio Java richiedono che, quando `otherObject` è `null`, il metodo `equals` restituisca `false`.

Ecco una versione migliorata del metodo `equals` che prende in considerazione questi due casi:

```
public boolean equals(Object otherObject)
{
    if (otherObject == null) { return false; }
    if (getClass() != otherObject.getClass()) { return false; }
    Stamp other = (Stamp) otherObject;
    return color.equals(other.color) && value == other.value;
}
```

Quando sovrascrivete il metodo `equals` in una sottoclasse dovreste prima di tutto invocare il metodo `equals` della superclasse, per verificare se le variabili di esemplare della superclasse sono uguali. Ecco un esempio:

```
public CollectibleStamp extends Stamp
{
    private int year;
    ...
    public boolean equals(Object otherObject)
    {
        if (!super.equals(otherObject)) { return false; }
        CollectibleStamp other = (CollectibleStamp) otherObject;
        return year == other.year;
    }
}
```



## Computer e società 9.1

### Chi controlla Internet?

Nel 1962, J.C.R. Licklider era a capo del primo programma di ricerca sui calcolatori presso DARPA (Defense Advanced Research Projects Agency, Agenzia per i progetti di ricerca avanzata del Ministero della Difesa degli Stati Uniti d'America) e in quegli anni pubblicò una serie di articoli che descrivevano una "rete a galassie", tramite la quale gli utenti dei computer avrebbero potuto accedere ai dati e ai programmi presenti in siti diversi da quelli in cui lavoravano: stiamo parlando di un periodo molto precedente a quello in cui furono inventate le reti per calcolatori. Nel 1969, quattro calcolatori (tre in California e uno nello Utah) vennero connessi ad ARPANET, che precorse Internet. La rete crebbe rapidamente, collegando computer di varie università e organizzazioni di ricerca, e all'inizio si pensò che la maggior parte dei ricercatori l'avrebbe utilizzata per eseguire programmi su computer diversi dal proprio: usando l'esecuzione remota, un ricercatore di una istituzione avrebbe avuto accesso a un computer poco utilizzato che si trovasse in una diversa località. Divenne rapidamente evidente, però, che l'esecuzione remota non era il motivo prevalente di utilizzo della rete: l'*applicazione killer* era la posta elettronica, cioè lo scambio di messaggi fra utenti di computer situati in luoghi diversi.

Nel 1972, Bob Kahn propose di estendere ARPANET per creare la rete *Internet*: un insieme di reti intercomunicanti. Tutte le reti appartenenti a Internet condividono *protocolli* comuni per la trasmissione dei dati: Kahn e Vinton Cerf svilupparono

questo insieme di protocolli, oggi denominato TCP/IP (Transmission Control Protocol / Internet Protocol). Il primo gennaio 1983 tutti i computer connessi a Internet passarono contemporaneamente all'utilizzo di TCP/IP, in uso ancora oggi.

Nel tempo, divenne disponibile sulla rete Internet una quantità sempre maggiore di informazione, creata da ricercatori e amatori. Ad esempio, il Progetto Gutenberg ([www.gutenberg.org](http://www.gutenberg.org)) rende disponibile in formato elettronico il testo di importanti libri classici, i cui diritti d'autore siano scaduti. Nel 1989, Tim Berners-Lee iniziò il suo lavoro sui documenti ipertestuali, che consentono agli utenti una consultazione arricchita da collegamenti con documenti correlati: l'infrastruttura derivante è oggi nota con il nome di World Wide Web (*ragnatela mondiale*, WWW).

Le prime interfacce per accedere a queste informazioni erano, rispetto agli standard di oggi, incredibilmente confuse e difficili da usare e, nel marzo 1993, il traffico dovuto al sistema WWW era lo 0,1% del traffico totale in Internet. Tutto ciò cambiò radicalmente quando Marc Andersen, allora laureando al NCSA (National Center for Supercomputing Applications), progettò e rese disponibile Mosaic, un'applicazione che era in grado di visualizzare pagine Web in forma grafica, usando immagini, colori e diversi tipi di *font* di caratteri.

Andersen divenne famoso e la sua fama si trasferì all'azienda che fondò, Netscape; inoltre, Microsoft acquisì la licenza di Mosaic per creare Internet Explorer. Nel 1996, il traffico WWW rappresentava più della metà dei dati trasportati dalla rete Internet.

Internet ha una struttura fortemente democratica. Chiunque può pubblicare qualsiasi cosa e può leggere tutto ciò che viene pubblicato da altri: un fenomeno che spesso non va d'accordo con i governi e le aziende.

Molti governi, infatti, controllano l'infrastruttura di Internet nel proprio Paese. Ad esempio, un utente di Internet in Cina che faccia una ricerca sul massacro di Piazza Tienanmen o sull'inquinamento nella propria città, potrebbe non ottenere alcun risultato. Il Vietnam blocca l'accesso a Facebook, probabilmente nel timore che i dissidenti anti-governativi lo possano utilizzare per organizzarsi meglio. Il governo statunitense ha preso che le biblioteche e le scuole a sovvenzione pubblica installassero filtri per bloccare informazioni sessualmente esplicite o di istigazione all'odio; inoltre, le organizzazioni per la sicurezza nazionale hanno spiato l'utilizzo di Internet da parte dei cittadini.

Nei casi in cui il servizio Internet viene fornito da aziende telefoniche o televisive, queste a volte interferiscono con offerte di servizio alternative: le compagnie che offrono il servizio di telefonia cellulare spesso si rifiutano di veicolare servizio di *voice-over-IP* (telefonia via Internet) e le aziende di televisione via cavo rallentano la diffusione di video con tecnologia *streaming*.

La rete Internet è diventata un potente strumento per la diffusione dell'informazione, buona e cattiva: è nostra responsabilità, come cittadini, chiedere ai nostri governi di poter controllare autonomamente l'accesso a tale informazione.

## Riepilogo degli obiettivi di apprendimento

### Ereditarietà, superclasse e sottoclasse

- Una sottoclasse eredita dati e comportamenti da una superclasse.
- Un oggetto di una sottoclasse può sempre essere utilizzato al posto di un oggetto della sua superclasse.

### Implementare sottoclassi

- Una sottoclasse eredita tutti i metodi che non sovrascrive.
- Una sottoclasse può sovrascrivere un metodo ereditato dalla sua superclasse fornendone una nuova implementazione.
- La parola riservata `extends` definisce la relazione di ereditarietà.

### Una sottoclasse può sovrascrivere metodi della sua superclasse

- Un metodo che sovrascrive l'omonimo metodo della superclasse ne può estendere o sostituire la funzionalità.
- Per invocare un metodo della superclasse si usa la parola riservata `super`.
- Un costruttore di una sottoclasse invoca il costruttore della superclasse senza parametri, a meno che non ci sia una diversa indicazione esplicita.
- Per invocare un costruttore della superclasse in un costruttore di una sottoclasse si usa la parola riservata `super` come primo enunciato.
- Un costruttore di una sottoclasse può fornire argomenti a un costruttore della superclasse usando la parola riservata `super`.

### Uso del polimorfismo per elaborare oggetti di tipi correlati

- Un riferimento a una sottoclasse può essere utilizzato in ogni punto del programma che preveda la presenza di un riferimento alla sua superclasse.
- Quando la macchina virtuale invoca un metodo di esemplare, usa il metodo della classe a cui appartiene il parametro隐式: si parla di "ricerca dinamica del metodo".
- Il polimorfismo ("avere molte forme") ci consente di manipolare oggetti che hanno in comune alcune funzionalità, anche se queste sono implementate in modi diversi.

### La classe `Object` e i suoi metodi

- Sovrascrivete il metodo `toString` in modo che restituisca una stringa che descrive lo stato dell'oggetto.
- Il metodo `equals` verifica se due oggetti hanno lo stesso contenuto.
- Se sapete che un oggetto è un esemplare di una determinata classe, potete usare un `cast` per convertirlo in quel tipo.
- L'operatore `instanceof` verifica se un oggetto è di un determinato tipo.

## Esercizi di riepilogo e approfondimento

- ★ **R9.1.** Nella sezione Esempi completi 9.1:
  - quali sono le sottoclassi di `Employee`?
  - quali sono le superclassi di `Manager`?
  - quali sono le superclassi e le sottoclassi di `SalariedEmployee`?
  - quali classi sovrascrivono il metodo `weeklyPay` della classe `Employee`?

- e. quali classi sovrascrivono il metodo `setName` della classe `Employee`?  
f. quali sono le variabili di esemplare di un oggetto di tipo `HourlyEmployee`?
- \* **R9.2.** Nelle seguenti coppie di classi, individuate la superclasse e la sottoclasse:
- `Employee`, `Manager` (cioè *dipendente* e *dirigente*)
  - `GraduateStudent`, `Student` (*studente universitario* e *studente*)
  - `Person`, `Student` (*persona* e *studente*)
  - `Employee`, `Professor` (*dipendente* e *professore*)
  - `BankAccount`, `CheckingAccount` (*conto bancario* e *conto corrente bancario*)
  - `Vehicle`, `Car` (*veicolo* e *automobile*)
  - `Vehicle`, `Minivan` (*veicolo* e *automobile familiare*)
  - `Car`, `Minivan` (*automobile* e *automobile familiare*)
  - `Truck`, `Vehicle` (*autocarro* e *veicolo*)
- \* **R9.3.** In un programma che gestisce l'inventario in un negozio di piccoli elettrodomestici, perché non è utile definire la superclasse `SmallAppliance` (*piccolo elettrodomestico*) con le sottoclassi `Toaster` (*tostapane*), `CarVacuum` (*aspirapolvere per automobile*), `TravelIron` (*ferro da stirto da viaggio*) e così via?
- \* **R9.4.** Quali metodi eredita dalla propria superclasse la classe `ChoiceQuestion`? Quali metodi sovrascrive? Quali metodi aggiunge?
- \* **R9.5.** Quali metodi eredita dalla propria superclasse la classe `SavingsAccount` vista nella sezione Consigli pratici 9.1? Quali metodi sovrascrive? Quali metodi aggiunge?
- \* **R9.6.** Elencate le variabili di esemplare di un oggetto di tipo `CheckingAccount`, classe definita nella sezione Consigli pratici 9.1.
- \*\* **R9.7.** Se la classe `Sub` estende la classe `Sandwich`, quali fra le seguenti sono assegnazioni permesse dopo aver eseguito i primi due enunciati?
- ```
Sandwich x = new Sandwich();
Sub y = new Sub();
```
- `x = y;`
  - `y = x;`
  - `y = new Sandwich();`
  - `x = new Sub();`
- \* **R9.8.** Disegnate un diagramma di ereditarietà che mostri le relazioni eritarie fra le classi seguenti:
- `Person` (*persona*)
  - `Employee` (*dipendente*)
  - `Student` (*studente*)
  - `Instructor` (*docente*)
  - `Classroom` (*aula*)
  - `Object`
- \* **R9.9.** Disegnate un diagramma di ereditarietà che mostri le relazioni eritarie fra le classi seguenti, utilizzate in un programma orientato agli oggetti per la simulazione di traffico:
- `Vehicle` (*veicolo*)
  - `Car` (*automobile*)

- Truck (*autocarro*)
  - Sedan (*automobile berlina*)
  - Coupe (*automobile sportiva*)
  - PickupTruck (*piccolo autocarro*)
  - SportUtilityVehicle (*automobile SUV*)
  - Minivan (*automobile familiare*)
  - Bicycle (*bicicletta*)
  - Motorcycle (*motociclo*)
- ★ R9.10.** Quali relazioni di ereditarietà stabilireste fra le classi seguenti?
- Student (*studente*)
  - Professor (*professore*)
  - TeachingAssistant (*assistente del professore*)
  - Employee (*dipendente*)
  - Secretary (*segretario*)
  - DepartmentChair (*direttore di dipartimento*)
  - Janitor (*bidello*)
  - SeminarSpeaker (*oratore di seminario*)
  - Person (*persona*)
  - Course (*corso*)
  - Seminar (*seminario*)
  - Lecture (*lezione*)
  - ComputerLab (*esercitazione in laboratorio*)
- ★★ R9.11.** In che modo un cast come `(BankAccount) x` differisce da un cast tra valori numerici come `(int) x`?
- ★★★ R9.12.** Quale di queste condizioni restituisce true? Controllate la documentazione di Java per identificare le relazioni di ereditarietà e ricordate che `System.out` è un oggetto di tipo `PrintStream`.
- a. `System.out instanceof PrintStream`
  - b. `System.out instanceof OutputStream`
  - c. `System.out instanceof LogStream`
  - d. `System.out instanceof Object`
  - e. `System.out instanceof String`
  - f. `System.out instanceof Writer`

## Esercizi di programmazione

- ★ E9.1.** Progettate la classe `BasicAccount` come sottoclasse di `BankAccount` in modo che il suo metodo `withdraw` non consenta di prelevare una somma di denaro superiore a quella presente nel conto.
- ★ E9.2.** Progettate la classe `BasicAccount` come sottoclasse di `BankAccount` in modo che il suo metodo `withdraw` addebiti una penale di \$30 per ogni prelievo che produca uno scoperto.
- ★★ E9.3.** Riprogettate la classe `CheckingAccount` vista nella sezione Consigli pratici 9.1 in modo che, durante un mese, il primo scoperto provochi l'addebito di una penale di \$20, mentre per ogni altro scoperto che avvenga nello stesso mese la penale sia \$30.

- \*\* **E9.4.** Aggiungete alla gerarchia di domande vista nel Paragrafo 9.1 la classe `NumericQuestion` per gestire domande con risposta numerica. Se la differenza tra la risposta data e il valore previsto non è maggiore di 0.01, accettate la risposta come corretta.
- \*\* **E9.5.** Aggiungete alla gerarchia di domande vista nel Paragrafo 9.1 la classe `FillInQuestion`. Un esemplare di questa classe viene costruito fornendo una stringa che contiene la risposta corretta preceduta e seguita da un carattere di sottolineatura, come "The inventor of Java was James Gosling". La domanda va visualizzata così:
- The inventor of Java was \_\_\_\_\_
- \* **E9.6.** Modificate il metodo `checkAnswer` della classe `Question` in modo che ignori la differenza tra lettere maiuscole e minuscole e la presenza di spazi vuoti in numero maggiore del previsto. Ad esempio, la risposta "JAMES gosling" deve essere ritenuta equivalente a "James Gosling".
- \*\* **E9.7.** Aggiungete alla gerarchia di domande vista nel Paragrafo 9.1 la classe `AnyCorrectChoiceQuestion`, che consenta domande con più risposte corrette, scelte tra quelle proposte. L'esaminato deve fornire una qualsiasi delle risposte corrette. La stringa che memorizza le risposte corrette deve contenerle tutte, separate da spazi. Aggiungete al testo della domanda istruzioni opportune.
- \*\* **E9.8.** Aggiungete alla gerarchia di domande vista nel Paragrafo 9.1 la classe `MultiChoiceQuestion`, che consenta domande con più risposte corrette, scelte tra quelle proposte. L'esaminato deve fornire tutte (e sole) le risposte corrette, separate da spazi. Aggiungete al testo della domanda istruzioni opportune.
- \*\* **E9.9.** Aggiungete il metodo `addText` alla classe `Question` e progettate una versione diversa di `ChoiceQuestion` che invochi `addText` invece di memorizzare al proprio interno un vettore con le risposte disponibili.
- \* **E9.10.** Aggiungete il metodo `toString` alle classi `Question` e `ChoiceQuestion`.
- \*\* **E9.11.** Realizzate una classe `Person` e due sue sottoclassi, `Student` e `Instructor`. Una persona ha un nome e un anno di nascita, uno studente ha una disciplina di specializzazione e un docente ha un salario. Per ogni classe scrivete la dichiarazione, i costruttori e il metodo `toString`. Fornite un programma di prova per collaudare classi e metodi.
- \*\* **E9.12.** Progettate una classe `Employee`: ogni dipendente ha un nome e una retribuzione. Progettate, quindi, una classe `Manager` che erediti da `Employee`: aggiungete una variabile di esemplare di tipo `String`, chiamata `department`, e scrivete un metodo `toString` che restituisca una stringa con il nome, il reparto e la retribuzione. Progettate, infine, una classe `Executive` che erediti da `Manager`. Fornite un metodo `toString` appropriato in tutte le classi e scrivete un programma che collaudi classi e metodi.
- \*\* **E9.13.** La classe `java.awt.Rectangle` della libreria standard di Java non mette a disposizione un metodo per calcolare l'area o il perimetro di un rettangolo. Progettate la classe `BetterRectangle`, sottoclasse di `Rectangle`, che abbia i metodi `getPerimeter` e `getArea`, senza aggiungere variabili di esemplare. Nel costruttore invocate i metodi `setLocation` e `setSize` della classe `Rectangle`. Scrivete un programma che collaudi i metodi che avete progettato.
- \*\*\* **E9.14.** Risolvete nuovamente l'esercizio precedente ma nel costruttore della classe `BetterRectangle` invocate il costruttore della superclasse.

- \*\* **E9.15.** Un “punto etichettato” (*labeled point*) ha una coordinata *x*, una coordinata *y* e una stringa che funge da etichetta. Definite la classe `LabeledPoint` con il costruttore `LabeledPoint(int x, int y, String label)` e il metodo `toString` che visualizzi *x*, *y* e l’etichetta.
- \*\* **E9.16.** Risolvete nuovamente l’esercizio precedente memorizzando la posizione del punto in un oggetto di tipo `java.awt.Point`. Il metodo `toString` deve invocare il metodo `toString` della classe `Point`.
- \*\* **E9.17 (economia).** Modificate la classe `CheckingAccount` vista nella sezione Consigli pratici 9.1 in modo che prelevi una commissione di \$1 per ogni versamento o prelievo eseguito oltre le tre transazioni mensili gratuite. Inserite il codice che calcola la commissione dovuta in un metodo separato, che verrà invocato dai metodi `deposit` e `withdraw`.

Sul sito web dedicato al libro si trova una raccolta di progetti di programmazione più complessi.

# 10

## Interfacce



### Obiettivi del capitolo

- Saper dichiarare e usare interfacce
- Utilizzare le interfacce per ridurre l'accoppiamento tra classi
- Imparare a realizzare classi ausiliarie e classi interne
- Realizzare ricevitori di eventi in applicazioni grafiche

Per migliorare la produttività dei programmatore si vuole poter *riutilizzare* componenti software all'interno di più progetti. In questo capitolo apprenderete un'importante strategia di programmazione che consente di separare le parti riutilizzabili di un'elaborazione dalle parti che vanno modificate in relazione alla situazione in cui vengono riutilizzate. La parte riutilizzabile invoca metodi di una *interfaccia* e lavora di concerto con una classe che realizza i metodi dell'interfaccia stessa. Per realizzare una diversa applicazione basta progettare una diversa classe che realizzi la medesima interfaccia: il comportamento del programma si modifica in base a quello della classe che viene effettivamente utilizzata.

## 10.1 Uso di interfacce per il riutilizzo di algoritmi

Quando si fornisce un servizio, solitamente lo si vuole rendere disponibile per una clientela che sia la più vasta possibile. Un ristorante, ad esempio, serve i suoi clienti e, in Java, se ne potrebbe realizzare un modello usando un metodo come questo:

```
public void serve(Person client)
```

Ma cosa succede se il ristorante vuole avere, tra i propri clienti, anche altre creature viventi, che non siano persone (`Person`), come un gatto (`Cat`) o un altro animale domestico? In una situazione come questa ha senso definire un nuovo tipo di dato che abbia esattamente i metodi che servono per soddisfare le invocazioni effettuate dal metodo di elaborazione quando si occupa di un oggetto. Un tale tipo di dato si chiama *interfaccia*.

Ad esempio, il tipo interfaccia `Customer`, definito per rappresentare un generico cliente di un ristorante, potrebbe avere i metodi `eat` (`mangia`) e `pay` (`paga`). Il metodo che serve un cliente potrebbe, quindi, essere dichiarato in questo modo:

```
public void serve(Customer client)
```

Se le classi `Person` e `Cat` sono conformi all'interfaccia `Customer`, oggetti che siano esemplari di tali classi possono essere forniti come argomenti al metodo `serve`.

Come esempio pratico studieremo l'interfaccia `Comparable` della libreria di Java: definisce un metodo, `compareTo`, che, dati due oggetti, determina quale precede l'altro secondo un determinato criterio di ordinamento. In questo modo è possibile progettare un servizio di ordinamento che accetti raccolte di dati appartenenti a molte classi diverse: tutto ciò che serve è che le classi siano conformi all'interfaccia `Comparable`. Il servizio di ordinamento è interessato solamente al metodo `compareTo`, che usa per disporre gli oggetti in ordine.

Nei paragrafi che seguono imparerete a capire quando sia utile definire un tipo interfaccia, quali metodi deve contenere, come si definisce l'interfaccia stessa e come si dichiarano classi che siano conformi a una data interfaccia.

### 10.1.1 Individuare un tipo interfaccia

In questo paragrafo analizzeremo un servizio che calcola valori medi, con l'obiettivo di fare in modo che sia il più generale possibile. Iniziamo da un'implementazione del servizio che calcoli il saldo medio di un array di conti bancari:

```
public static double average(BankAccount[] objects)
{
    double sum = 0;
    for (BankAccount obj : objects)
    {
        sum = sum + obj.getBalance();
    }
    if (objects.length > 0) { return sum / objects.length; }
    else { return 0; }
}
```

Immaginiamo, ora, di voler calcolare il valore medio di oggetti di altro tipo: dobbiamo scrivere un altro metodo. Ecco quello che elabora oggetti **Country**:

```
public static double average(Country[] objects)
{
    double sum = 0;
    for (Country obj : objects)
    {
        sum = sum + obj.getArea();
    }
    if (objects.length > 0) { return sum / objects.length; }
    else { return 0; }
}
```

L'algoritmo che calcola il valore medio è, evidentemente, lo stesso in tutti i casi, ma cambiano i dettagli della misurazione dei valori di cui fare la media: ci piacerebbe progettare un *unico* metodo che fornisca questo servizio.

C'è, però, un problema: ogni classe usa, in generale, un metodo di nome diverso per consentire l'ispezione del valore di cui si vuol fare la media. Nella classe **BankAccount** usiamo il metodo **getBalance**, mentre nella classe **Country** usiamo il metodo **getArea**.

Supponete che le diverse classi possano accordarsi sull'esistenza di un metodo, **getMeasure**, che fornisca la misura da usare nell'analisi dei dati: per i conti bancari **getMeasure** restituisce il saldo, per le nazioni restituisce la superficie, e così via.

In questo modo potremmo realizzare un unico metodo che calcoli:

```
sum = sum + obj.getMeasure();
```

Ma l'accordo sul nome del metodo risolve soltanto una parte del problema, perché, in Java, dobbiamo dichiarare anche il tipo della variabile **obj**. Ovviamente non possiamo semplicemente scrivere:

```
BankAccount oppure Country oppure . . . obj; // non si può
```

Dobbiamo inventare un nuovo tipo che descriva qualsiasi classe i cui oggetti possano essere "misurati": nel prossimo paragrafo vedremo come farlo.

### 10.1.2 Dichiaraone un tipo interfaccia

In Java un tipo interfaccia dichiara i metodi che possono essere invocati con una variabile di quel tipo.

In Java, un **tipo interfaccia** (*interface type*) viene utilizzato per specificare un elenco di operazioni necessarie. La dichiarazione è simile a quella di una classe: si elencano i metodi richiesti dall'interfaccia, senza, però, fornire l'implementazione di tali metodi. Ecco, ad esempio, la definizione di un tipo interfaccia che chiamiamo **Measurable**, cioè "misurabile":

```
public interface Measurable
{
    double getMeasure();
}
```

L'interfaccia **Measurable** richiede un solo metodo, **getMeasure**, ma, in generale, ne può richiedere più d'uno.

**Sintassi di Java****10.1 Dichiarazione di interfaccia****Sintassi**

```
public interface NomeInterfaccia
{
    intestazioni dei metodi
}
```

**Esempio**

I metodi di un'interfaccia sono automaticamente pubblici.

```
public interface Measurable
{
    double getMeasure();
}
```

Non viene fornita alcuna implementazione.

Un'interfaccia è simile a una classe, ma ci sono parecchie differenze importanti:

- Un'interfaccia non ha variabili di esemplare.
- I metodi di un'interfaccia devono essere *astratti*, cioè non hanno un'implementazione, oppure, a partire da Java 8, possono essere metodi *static* o *default* (sezioni Note per Java 8 10.1 e 10.2)
- Tutti i metodi di un'interfaccia sono automaticamente pubblici.
- Un'interfaccia non ha costruttori: le interfacce non sono classi e non si possono costruire oggetti di tipo interfaccia.

A questo punto abbiamo un tipo di dato che indica la possibilità di ottenere un valore (o una “misura”) corrispondente a un oggetto, quindi siamo in grado di realizzare un metodo *average* riutilizzabile:

```
public static double average(Measurable[] objects)
{
    double sum = 0;
    for (Measurable obj : objects)
    {
        sum = sum + obj.getMeasure();
    }
    if (objects.length > 0) { return sum / objects.length; }
    else { return 0; }
}
```

Questo metodo è utilizzabile con oggetti di qualsiasi classe che sia conforme all'interfaccia *Measurable*. Nel prossimo paragrafo vedremo cosa deve fare una classe per rendere “misurabili” i propri oggetti.

Va osservato che l'interfaccia *Measurable* non è un tipo definito nella libreria standard: è stato creato specificatamente per questo libro, come semplice caso di studio per la nozione di interfaccia.

### 10.1.3 Implementare un tipo interfaccia

Per indicare che una classe implementa un'interfaccia si usa la parola riservata `implements`.

Il metodo `average` visto nel paragrafo precedente è in grado di elaborare oggetti di qualsiasi classe che implementi l'interfaccia `Measurable`. Una classe **implementa** o realizza **un'interfaccia** se la dichiara in una clausola `implements`, in questo modo:

```
public class BankAccount implements Measurable
```

Di conseguenza, la classe deve implementare i metodi astratti richiesti dall'interfaccia:

```
public class BankAccount implements Measurable
{
    ...
    public double getMeasure()
    {
        return balance;
    }
}
```

Note che la classe deve dichiarare il metodo con accesso `public`, anche se per l'interfaccia tale dichiarazione non è necessaria: tutti i metodi di un'interfaccia sono pubblici.

Dopo aver dichiarato che la classe `BankAccount` implementa l'interfaccia `Measurable`, gli oggetti di tipo `BankAccount` sono anche di tipo `Measurable`:

```
Measurable obj = new BankAccount(); // va bene
```

Una variabile di tipo `Measurable` può contenere un riferimento a un oggetto che sia esemplare di una classe qualsiasi che implementa l'interfaccia `Measurable`.

## Sintassi di Java

### 10.2 Implementazione di interfaccia

#### Sintassi

```
public class NomeClasse implements NomeInterfaccia1, NomeInterfaccia2, ...
{
    variabili di esemplare
    metodi
}
```

#### Esempio

```
public class BankAccount implements Measurable
{
    ...
    public double getMeasure()
    {
        return balance;
    }
    ...
}
```

Variabili di esemplare di `BankAccount`

Altri metodi di `BankAccount`

Elenco di tutte le interfacce realizzate dalla classe.

Questo metodo fornisce l'implementazione del metodo dichiarato nell'interfaccia.

Analogamente, è semplice modificare la classe `Country` perché implementi l'interfaccia `Measurable`:

```
public class Country implements Measurable
{
    ...
    public double getMeasure()
    {
        return area;
    }
}
```

Il programma che conclude questo paragrafo usa un unico metodo `average` (definito nella classe `Data`) per calcolare il valore medio di alcuni conti bancari e di alcune nazioni.

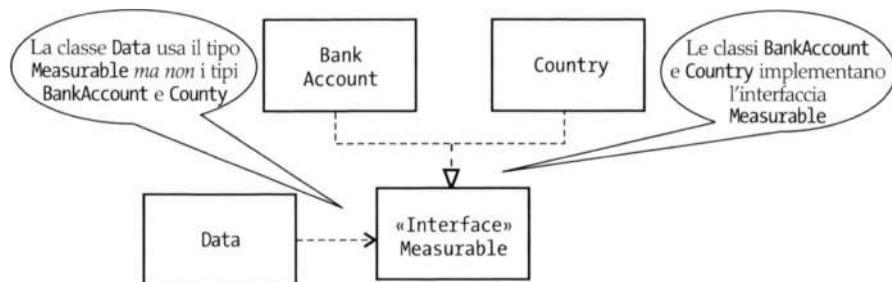
Si tratta di un utilizzo tipico delle interfacce: definendo l'interfaccia `Measurable` abbiamo reso riutilizzabile il metodo `average`.

La Figura 1 mostra le relazioni che esistono tra la classe `Data`, l'interfaccia `Measurable` e le classi che la implementano. Si noti che la classe `Data` dipende solamente dall'interfaccia `Measurable` e non è accoppiata alle classi `BankAccount` e `Country`.

Nella notazione UML le interfacce vengono contrassegnate dall'indicazione «interface», mentre una freccia tratteggiata con punta triangolare segnala la relazione che esiste tra una classe e un'interfaccia da essa implementata. Occorre fare molta attenzione alla punta delle frecce: una linea tratteggiata con la freccia a V aperta indica, invece, una relazione di dipendenza (cioè la classe ne “usa” un'altra).

**Figura 1**

Diagramma UML della classe `Data` e delle classi che implementano l'interfaccia `Measurable`



### File Data.java

```
1  public class Data
2  {
3      /**
4          Calcola la media delle misure degli oggetti dati.
5          @param objects un array di oggetti Measurable
6          @return la media delle misure
7      */
8      public static double average(Measurable[] objects)
9      {
10         double sum = 0;
11         for (Measurable obj : objects)
12         {
13             sum = sum + obj.getMeasure();
14         }
15     }
16 }
```

I tipi interfaccia vengono utilizzati per rendere il codice maggiormente riutilizzabile.

```

15     if (objects.length > 0) { return sum / objects.length; }
16     else { return 0; }
17   }
18 }
```

### File MeasurableTester.java

```

1 /**
2  * Questo programma usa le classi "misurabili" BankAccount e Country.
3 */
4 public class MeasurableTester
5 {
6   public static void main(String[] args)
7   {
8     // invocazione di average con un array di oggetti BankAccount
9     Measurable[] accounts = new Measurable[3];
10    accounts[0] = new BankAccount(0);
11    accounts[1] = new BankAccount(10000);
12    accounts[2] = new BankAccount(2000);
13
14    double averageBalance = Data.average(accounts);
15    System.out.println("Average balance: " + averageBalance);
16    System.out.println("Expected: 4000");
17
18    // invocazione di average con un array di oggetti Country
19    Measurable[] countries = new Measurable[3];
20    countries[0] = new Country("Uruguay", 176220);
21    countries[1] = new Country("Thailand", 513120);
22    countries[2] = new Country("Belgium", 30510);
23
24    double averageArea = Data.average(countries);
25    System.out.println("Average area: " + averageArea);
26    System.out.println("Expected: 239950");
27  }
28 }
```

### Esecuzione del programma

```

Average balance: 4000.0
Expected: 4000
Average area: 239950.0
Expected: 239950
```

#### 10.1.4 Confronto tra ereditarietà e interfacce

Nel Capitolo 9 avete visto come usare l'ereditarietà per costruire un modello di una gerarchia di classi correlate, come possono esserlo i diversi tipi di domande che compongono un questionario: le domande con risposta a scelta multipla e le domande con spazi da riempire sono due esempi del più generico concetto di “domanda”, ciascuno con le proprie caratteristiche specifiche.

Le interfacce consentono di rappresentare una relazione un po' diversa. Considerate, ad esempio, le classi `BankAccount` e `Country` viste nel paragrafo precedente: entrambe implementano l'interfaccia `Measurable`, ma non hanno altre caratteristiche in comune.

Essere “misurabile” è soltanto un aspetto di ciò che significa essere un conto bancario o una nazione, anche se è comunque utile dichiarare questo comportamento comune, perché consente di scrivere codice che sfrutta tale affinità, ad esempio progettando un metodo che calcoli un valore medio.

Una classe può implementare più di un’interfaccia, ad esempio:

```
public class Country implements Measurable, Named
```

dove l’interfaccia `Named` è così definita:

```
public interface Named
{
    String getName();
}
```

Al contrario, una classe può estendere un’unica superclasse.

Un’interfaccia descrive il comportamento che deve essere esibito da chi la implementa e prima della versione 8 di Java un’interfaccia non poteva definire alcuna implementazione. Ora, invece, è possibile definire in un’interfaccia un’implementazione “predefinita” o *di default* e, di conseguenza, la distinzione tra interfacce e classi astratte (viste nella sezione Argomenti avanzati 9.3) è più sottile: una differenza rilevante che rimane è il fatto che un tipo interfaccia, diversamente da una classe astratta, è *privo di informazioni di stato*, cioè non ha variabili di esemplare.

In generale, si progetta un’interfaccia quando si scrive codice che elabori in modo molto simile oggetti di classi diverse. Ad esempio, un programma di disegno potrebbe dover disegnare oggetti graficamente diversi, come segmenti, immagini, testo e così via: in tale situazione sarà utile un’interfaccia, ad esempio `Drawable`, che definisca il metodo `draw`, da invocare per disegnare la forma. Un altro esempio può essere una simulazione di traffico stradale, che consente il movimento di persone, automobili, cani, palloni e così via: si potrebbe definire l’interfaccia `Moveable`, con i metodi `move` (per spostare un’entità) e `getPosition` (per ispezionarne la posizione).



## Auto-valutazione

1. Immaginate di voler utilizzare il metodo `average` per calcolare lo stipendio medio di un array di oggetti `Employee`. Quale condizione deve essere soddisfatta dalla classe `Employee`?
2. Perché il metodo `average` non può avere una variabile parametro di tipo `Object[]`?
3. Perché non possiamo usare il metodo `average` per calcolare la lunghezza media di oggetti di tipo `String`?
4. Cosa c’è di sbagliato in questo frammento di codice?  

```
Measurable meas = new Measurable();
System.out.println(meas.getMeasure());
```
5. Cosa c’è di sbagliato in questo frammento di codice?  

```
Measurable meas = new Country("Uruguay", 176220);
System.out.println(meas.getName());
```

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi E10.1, E10.2 e E10.3, al termine del capitolo.



## Errori comuni 10.1

### Dimenticarsi di dichiarare pubblici i metodi implementati

I metodi di un'interfaccia non vengono dichiarati `public`, perché lo sono per impostazione predefinita. Tuttavia, i metodi di una classe non sono pubblici se ciò non viene specificato esplicitamente, perché la loro modalità di accesso predefinita è quella “di pacchetto”, come abbiamo visto nel Capitolo 8. Dimenticare la parola riservata `public` nella realizzazione di un metodo relativo a un'interfaccia è un errore comune:

```
public class BankAccount implements Measurable
{
    ...
    double getMeasure() // deve essere public
    {
        return balance;
    }
}
```

Di conseguenza, il compilatore segnala che il metodo ha una modalità di accesso più ristretta, “di pacchetto” anziché pubblica. La soluzione consiste nel dichiarare il metodo con attributo `public`.



## Errori comuni 10.2

### Cercare di creare un esemplare di un tipo interfaccia

È possibile dichiarare una variabile il cui tipo sia un'interfaccia:

```
Measurable meas;
```

Ma *non si può* costruire un oggetto di un tipo interfaccia:

```
Measurable meas = new Measurable(); // ERRORE
```

Le interfacce non sono classi e non esistono oggetti il cui tipo sia un'interfaccia. Se una variabile di tipo interfaccia fa riferimento a un oggetto, questo deve essere un esemplare di una classe che implementa tale interfaccia:

```
Measurable meas = new BankAccount(); // così va bene
```



## Argomenti avanzati 10.1

### Costanti nelle interfacce

Le interfacce non possono avere variabili di esemplare, ma al loro interno si possono dichiarare **costanti**. Quando dichiarate una costante in un'interfaccia, potete (e dovreste) omettere le parole riservate `public static final`, perché in un'interfaccia tutte le variabili sono automaticamente `public static final`. Ad esempio:

```
public interface Named
{
    String NO_NAME = "(NONE)";
    ...
}
```

Con questa definizione la costante `Named.NO_NAME` può essere utilizzata per indicare l'assenza di un nome.

Non capita tanto spesso di dover definire costanti in un tipo interfaccia. In particolare, bisognerebbe evitare di definire più costanti tra loro correlate (come `int NORTH = 1, int NORTHEAST = 2, e così via`) perché in tali casi è preferibile usare un tipo enumerativo, come visto nella sezione Argomenti avanzati 5.4.

## Note per Java 8 10.1

---

### Metodi statici nelle interfacce

Prima di Java 8 tutti i metodi delle interfacce dovevano essere astratti, mentre la versione Java 8 consente di definire nelle interfacce metodi statici che funzionano esattamente come i metodi statici definiti nelle classi. Un metodo statico definito in un'interfaccia non opera su un oggetto e il suo scopo dovrebbe essere in qualche modo correlato a quello dell'interfaccia in cui è contenuto.

Ad esempio, sarebbe decisamente sensato definire il metodo `average` visto nel Paragrafo 10.1 all'interno dell'interfaccia `Measurable`:

```
public interface Measurable
{
    double getMeasure(); // un metodo astratto
    static double average(Measurable[] objects) // un metodo static
    {
        . . . // la stessa implementazione vista nel Paragrafo 10.1
    }
}
```

Per invocare tale metodo si scrive il nome dell'interfaccia che lo contiene:

```
double meanArea = Measurable.average(countries);
```

## Note per Java 8 10.2

---

### Metodi di default

Un **metodo predefinito** o di default (*default method*) in un'interfaccia è un metodo non statico di cui viene definita anche l'implementazione. Una classe che implementa l'interfaccia erediterà il comportamento predefinito del metodo oppure lo potrà sovrscrivere: il fatto che in un'interfaccia venga predefinita un'implementazione per un metodo alleggerisce il lavoro necessario per realizzare una classe che la implementi.

Ad esempio, l'interfaccia `Measurable` potrebbe dichiarare il metodo `getMeasure` come predefinito, in questo modo:

```
public interface Measurable
{
    default double getMeasure() { return 0; }
}
```

Se una classe implementa questa interfaccia e non sovrascrive il metodo `getMeasure`, eredita questo comportamento predefinito.

Questo esempio specifico non è particolarmente utile: solitamente non si vuole che tutti gli oggetti abbiano misura uguale a zero. Ecco, invece, un esempio più interessante, dove un metodo di default invoca un altro metodo dell'interfaccia:

```
public interface Measurable
{
    double getMeasure(); // un metodo astratto
    default boolean smallerThan(Measurable other)
    {
        return getMeasure() < other.getMeasure();
    }
}
```

Il metodo `smallerThan` verifica se un oggetto ha una misura inferiore a quella di un altro, azione utile per disporre oggetti in ordine di misura crescente.

Una classe che voglia implementare l'interfaccia `Measurable` deve soltanto definire il metodo `getMeasure`, ereditando automaticamente il metodo `smallerThan`: un meccanismo che può essere molto utile. Ad esempio, l'interfaccia `Comparator` descritta nella sezione Argomenti avanzati 13.5 ha un metodo astratto ma più di una dozzina di metodi predefiniti.



## Note per Java 8 10.3

### Conflitto tra metodi di default

Anche se è un caso piuttosto raro, è possibile che una classe erediti metodi predefiniti da due interfacce diverse e che questi siano in conflitto tra loro, oppure che erediti un metodo predefinito che sia in conflitto con uno dei propri metodi. Due regole servono proprio a dirimere questi conflitti:

1. *Le classi vincono.* Quando una classe estende un'altra classe e implementa anche un'interfaccia, ereditando uno stesso metodo da entrambe, la sottoclasse eredita il metodo dalla superclasse, ignorando quello predefinito ereditato dall'interfaccia.
2. *Le interfacce rimangono in conflitto.* Quando una classe implementa due interfacce che hanno uno stesso metodo predefinito, la sua sovrascrittura nella classe è obbligatoria.

Per comprendere appieno queste regole, analizziamo l'esempio che segue:

```
public class Person
{
    public String name() { return firstName() + " " + lastName(); }
    ...
}

public interface Named
```

```

{
    default String name() { return "(NONE)"; }
}

public class User extends Person implements Named
{
    // eredita name() da Person
    ...
}

```

In questo caso il metodo definito nella superclasse vince sul metodo definito nell’interfaccia. Se, però, `Person` è un’interfaccia, la situazione è diversa:

```

public interface Person
{
    default String name() { return firstName() + " " + lastName(); }
    ...
}

```

Immaginiamo, ora, che una classe implementi entrambe le interfacce:

```
public class User implements Person, Named { . . . }
```

Questa classe *deve* sovrascrivere il metodo `name`, in modo che abbia un comportamento adeguato al contesto in cui si trova: i dettagli dipendono dal motivo per cui il progettista della classe `User` ha deciso di implementare l’interfaccia `Named`. Supponiamo, ad esempio, che esista un metodo che verifica se in un array di tipo `Named[]` esistono duplicati e che il programma lo voglia invocare per assicurarsi che i nomi degli utenti di un sistema informatico siano unici: in tal caso, il metodo `name` della classe `User` dovrebbe restituire il nome dell’utente.

## 10.2 Programmare con le interfacce

Nel paragrafo precedente avete visto come si possa realizzare un semplice servizio che accetta come parametro un’interfaccia: siamo riusciti a fornire al servizio, come argomento, oggetti di classi diverse e il servizio è stato in grado di invocare un metodo dell’interfaccia. Nei paragrafi che seguono imparerete le regole che governano la programmazione con interfacce in Java.

### 10.2.1 Conversione da classe a interfaccia

Guardate attentamente questa invocazione di metodo nel programma del paragrafo precedente:

```
double averageBalance = Data.average(accounts);
```

dove `accounts` è un array di oggetti `BankAccount`. Il metodo `average`, però, si aspetta di ricevere un array i cui elementi siano di tipo `Measurable`:

```
public double average(Measurable[] objects)
```

Si possono effettuare conversioni dal tipo di una classe al tipo di un'interfaccia che sia implementata dalla classe.

La conversione dal tipo `BankAccount` al tipo `Measurable` è lecita. In generale, si può effettuare una conversione dal tipo di una classe al tipo di una delle interfacce implementate dalla classe. Ad esempio:

```
BankAccount account = new BankAccount(1000);
Measurable meas = account; // va bene
```

Una variabile di tipo `Measurable` può anche riferirsi a un oggetto della classe `Country` vista nel paragrafo precedente, dato che anch'essa implementa l'interfaccia `Measurable`.

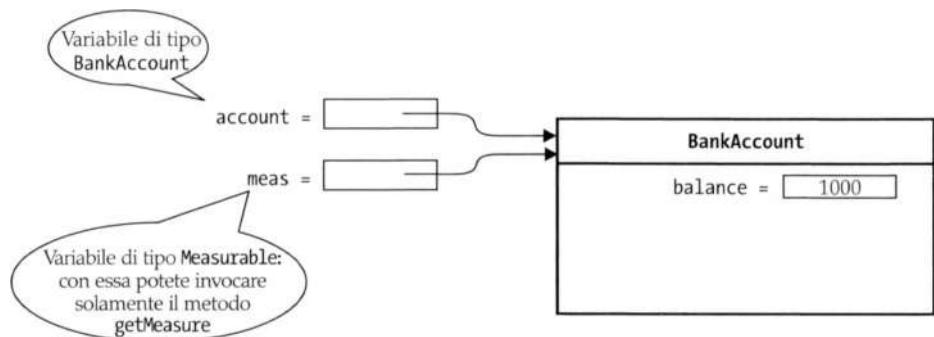
```
Country uruguay = new Country("Uruguay", 176220);
Measurable meas = uruguay; // anche questo va bene
```

Però la classe `Rectangle` della libreria standard non implementa l'interfaccia `Measurable`, per cui il seguente enunciato di assegnazione è sbagliato:

```
Measurable meas = new Rectangle(5, 10, 20, 30); // ERRORE
```

**Figura 2**

Variabili di tipo "riferimento a classe" e "riferimento a interfaccia"



### 10.2.2 Invocare metodi con variabili interfaccia

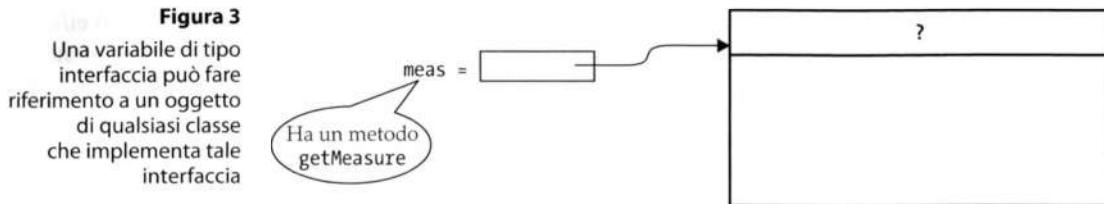
Supponiamo, ora, che la variabile `meas` sia stata inizializzata con un riferimento a un oggetto che sia esemplare di una classe che implementa l'interfaccia `Measurable`. Non sappiamo di quale classe l'oggetto sia esemplare, ma sappiamo che quella classe implementa tutti i metodi dell'interfaccia, quindi li possiamo invocare:

```
double result = meas.getMeasure();
```

Ragioniamo con più attenzione sull'invocazione del metodo `getMeasure`. Quale metodo `getMeasure` viene invocato? Le classi `BankAccount` e `Country` ne forniscono due diverse implementazioni. Come può essere invocato il metodo giusto se l'invocante non sa nemmeno a quale classe appartenga l'oggetto a cui fa riferimento `meas`?

Il polimorfismo entra di nuovo in azione (una prima trattazione del polimorfismo è stata fatta nel Paragrafo 9.4). La macchina virtuale Java trova il metodo corretto da invocare individuando per prima cosa la classe di cui l'oggetto è effettivamente esemplare, per poi invocare il metodo di quella classe avente il nome in esame: se, ad esempio, `meas` fa riferimento a un oggetto `BankAccount`, allora viene invocato il metodo `getMeasure` della classe `BankAccount`; se, invece, `meas` fa riferimento a un oggetto `Country`, allora viene invocato il metodo `getMeasure` della classe `Country`.

Le invocazioni di metodi mediante una variabile di tipo interfaccia sono polimorfiche: il metodo da invocare viene determinato durante l'esecuzione del programma.



### 10.2.3 Conversione da interfaccia a classe

A volte può accadere di memorizzare un oggetto in una variabile riferimento di tipo interfaccia e, poi, di avere la necessità di riconvertirlo al suo tipo originale. Analizziamo questo metodo che restituisce l'oggetto avente la misura maggiore tra i due ricevuti come parametri:

```
public static Measurable larger(Measurable obj1, Measurable obj2)
{
    if (obj1.getMeasure() > obj2.getMeasure())
    {
        return obj1;
    }
    else
    {
        return obj2;
    }
}
```

Il metodo `larger` restituisce l'oggetto avente la misura maggiore *sotto forma di riferimento di tipo Measurable*. Non c'è alternativa: il metodo non conosce il tipo effettivo degli oggetti che ha ricevuto. Proviamo a usare il metodo:

```
Country uruguay = new Country("Uruguay", 176220);
Country thailand = new Country("Thailand", 513120);
Measurable max = larger(uruguay, thailand);
```

Per convertire un riferimento di tipo interfaccia in un riferimento di tipo classe serve un *cast*.

Ora, cosa potete fare con il riferimento `max`? Voi sapete che si riferisce a un oggetto di tipo `Country`, ma il compilatore non lo sa, per cui, ad esempio, non potete invocare il metodo `getName`:

```
String countryName = max.getName(); // ERRORE
```

Questa invocazione è un errore, perché il tipo `Measurable` non ha un metodo `getName`.

Tuttavia, essendo assolutamente certi che `max` si riferisca a un oggetto di tipo `Country`, potete usare la notazione di forzatura (**cast**) per convertirlo al suo tipo originario:

```
Country maxCountry = (Country) max;
String name = maxCountry.getName();
```

Se vi siete sbagliati e l'oggetto in realtà non è una nazione, si verificherà un'eccezione durante l'esecuzione.



## Auto-valutazione

6. Si può usare un cast (BankAccount) `meas` per convertire una variabile `meas` di tipo `Measurable` in un riferimento di tipo `BankAccount`?
7. Se le classi `BankAccount` e `Country` implementano entrambe l'interfaccia `Measurable`, si può convertire un riferimento di tipo `Country` in un riferimento di tipo `BankAccount`?
8. Perché è impossibile costruire un oggetto di tipo `Measurable`?
9. Perché si può comunque dichiarare una variabile di tipo `Measurable`?
10. Cosa visualizza il seguente frammento di codice? Perché è un esempio di polimorfismo?

```
Measurable[] data = { new BankAccount(10000), new Country("Belgium", 30510) };
System.out.println(average(data));
```

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi R10.3, R10.4 e R10.5, al termine del capitolo.



## Esempi completi 10.1

### Analizzare sequenze di numeri

In questo Esempio completo analizziamo le proprietà di sequenze di numeri. Una sequenza di numeri può essere una serie di misure, di prezzi, di valori casuali o di valori matematici (come, ad esempio, i numeri primi). Sono molte le proprietà interessanti da analizzare. Ad esempio, potete cercare schemi ripetitivi o regole nascoste, oppure verificare se una sequenza è veramente casuale.

**Problema.** Analizziamo la distribuzione dell'ultima cifra dei valori della sequenza. Per una sequenza assegnata, vogliamo generare una tabella come questa:

```
0: 105
1: 94
2: 81
3: 112
4: 89
5: 103
6: 103
7: 100
8: 108
9: 105
```

Per poter gestire sequenze di qualunque tipo, dichiariamo un'interfaccia dotata di un unico metodo:

```
public interface Sequence
{
    int next();
}
```

La classe `LastDigitDistribution` analizza sequenze: usa un array di dieci contatori, aggiornati dal suo metodo `process`, che riceve un riferimento di tipo `Sequence` e il numero di valori da elaborare.

```
public void process(Sequence seq, int valuesToProcess)
{
    for (int i = 1; i <= valuesToProcess, i++)
    {
        int value = seq.next();
        int lastDigit = value % 10;
        counters[lastDigit]++;
    }
}
```

Si noti che il metodo non ha alcuna nozione in merito alla natura dei valori appartenenti alla sequenza.

Per analizzare una specifica sequenza occorre progettare una classe che implementi l'interfaccia `Sequence`. Ecco due esempi: la sequenza dei numeri che sono quadrati perfetti (1 4 9 16 25 ...) e una sequenza di numeri interi casuali.

```
public class SquareSequence implements Sequence
{
    private int n;

    public int next()
    {
        n++;
        return n * n;
    }
}

public class RandomSequence implements Sequence
{
    public int next()
    {
        return (int) (Integer.MAX_VALUE * Math.random());
    }
}
```

La classe che segue realizza l'intero processo di analisi. Notate come si evidenzi uno schema ripetitivo (*pattern*) nelle ultime cifre della sequenza di quadrati perfetti.

```
public class SequenceDemo
{
    public static void main(String[] args)
    {
        LastDigitDistribution dist1 = new LastDigitDistribution();
        dist1.process(new SquareSequence(), 1000);
        dist1.display();
        System.out.println();

        LastDigitDistribution dist2 = new LastDigitDistribution();
        dist2.process(new RandomSequence(), 1000);
    }
}
```

```
        dist2.display();
    }
}
```

### Esecuzione del programma

```
0: 100
1: 200
2: 0
3: 0
4: 200
5: 100
6: 200
7: 0
8: 0
9: 200

0: 105
1: 94
2: 81
3: 112
4: 89
5: 103
6: 103
7: 100
8: 108
9: 105
```

Nel pacchetto dei file scaricabili per questo libro, la cartella `worked_example_1` del Capitolo 10 contiene il codice sorgente completo della classe.

## 10.3 L'interfaccia Comparable

Implementate l'interfaccia Comparable in modo che gli oggetti delle vostre classi possano essere confrontati tra loro, ad esempio in un metodo di ordinamento.

Nei paragrafi precedenti abbiamo definito l'interfaccia `Measurable` e abbiamo progettato il metodo `average` che funziona con qualsiasi classe che implementi tale interfaccia. In questo paragrafo conoscerete l'interfaccia `Comparable` della libreria standard di Java.

L'interfaccia `Measurable` è utile per misurare singoli oggetti, mentre l'interfaccia `Comparable` è più complessa, dato che i confronti coinvolgono necessariamente due oggetti. L'interfaccia dichiara soltanto il metodo `compareTo` e l'invocazione

```
a.compareTo(b)
```

deve restituire un numero negativo se `a` precede `b`, zero se `a` e `b` sono uguali e un numero positivo se `b` precede `a`.

L'interfaccia `Comparable` ha un unico metodo:

```
public interface Comparable
{
    int compareTo(Object otherObject);
}
```

Ad esempio, la classe `BankAccount` può implementare `Comparable` in questo modo:

```
public class BankAccount implements Comparable
{
    ...
    public int compareTo(Object otherObject)
    {
        BankAccount other = (BankAccount) otherObject;
        if (balance < other.balance) { return -1; }
        if (balance > other.balance) { return 1; }
        return 0;
    }
}
```

Questo metodo `compareTo` confronta due conti bancari sulla base del loro saldo. Si osservi che la variabile parametro del metodo `compareTo` è di tipo `Object`. Per trasformarla in un riferimento di tipo `BankAccount`, usiamo un cast:

```
BankAccount other = (BankAccount) otherObject;
```

Dato che ora la classe `BankAccount` implementa l'interfaccia `Comparable`, si può ordinare un array di conti bancari usando il metodo `Arrays.sort`:

```
BankAccount[] accounts = new BankAccount[3];
accounts[0] = new BankAccount(10000);
accounts[1] = new BankAccount(2);
accounts[2] = new BankAccount(10000);
Arrays.sort(accounts);
```

Ora l'array `accounts` è ordinato per saldo crescente.



## Auto-valutazione

11. Come si può ordinare per superficie crescente un array di oggetti `Country`?
12. Si può usare il metodo `Arrays.sort` per ordinare un array di oggetti `String`? Controllate la documentazione API della classe `String`.
13. Si può usare il metodo `Arrays.sort` per ordinare un array di oggetti `Rectangle`? Controllate la documentazione API della classe `Rectangle`.
14. Scrivete un metodo, `max`, che trovi il maggiore tra due oggetti di tipo `Comparable`.
15. Scrivete un'invocazione del metodo progettato nella risposta alla domanda precedente che calcoli il maggiore tra due conti bancari, per poi visualizzare il suo saldo.

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi E10.10 e E10.30, al termine del capitolo.



## Suggerimenti per la programmazione 10.1

### Confrontare numeri interi e numeri in virgola mobile

Quando si scrive un metodo di confronto bisogna restituire un numero negativo per segnalare che il primo oggetto precede il secondo, zero se i due oggetti sono uguali e un

numero positivo se il secondo oggetto precede il primo. Avete già visto come implementare questa decisione con due diramazioni, ma quando si confrontano numeri interi *non negativi* c'è un'alternativa più semplice, basta sottrarre i due valori:

```
public class Person implements Comparable
{
    private int id; // deve essere >= 0
    ...
    public int compareTo(Object otherObject)
    {
        Person other = (Person) otherObject;
        return id - other.id;
    }
}
```

La differenza è negativa se `id < other.id`, è zero se i valori sono uguali ed è positiva negli altri casi.

Questo trucco non funziona se i numeri interi possono essere negativi, perché il calcolo della loro differenza può produrre una situazione di *overflow* (si veda l'Esercizio R10.1). Il metodo `Integer.compare`, però, funziona sempre:

```
return Integer.compare(id, other.id); // anche con numeri negativi
```

I numeri in virgola mobile non si possono confrontare per sottrazione (si veda l'Esercizio R10.2). Si usa, invece, il metodo `Double.compare`:

```
public class BankAccount implements Comparable
{
    ...
    public int compareTo(Object otherObject)
    {
        BankAccount other = (BankAccount) otherObject;
        return Double.compare(balance, other.balance);
    }
}
```

## Argomenti avanzati 10.2

---

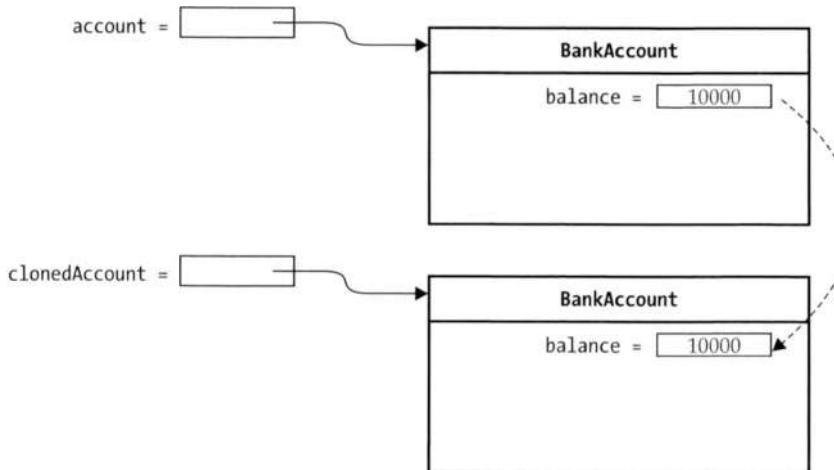
### Il metodo `clone` e l'interfaccia `Cloneable`

Sapete già che copiando il riferimento a un oggetto si ottengono semplicemente due riferimenti al medesimo oggetto:

```
BankAccount account = new BankAccount(10000);
BankAccount account2 = account;
account2.deposit(5000);
// ora sia account sia account2 puntano a un conto avente saldo 15000
```

Che cosa potete fare se volete effettivamente creare una copia di un oggetto? Questo è lo scopo del metodo `clone`: deve restituire un *nuovo* oggetto, avente lo stato identico a quello di un oggetto esistente (osservate la Figura 4).

**Figura 4**  
Clonazione di un oggetto

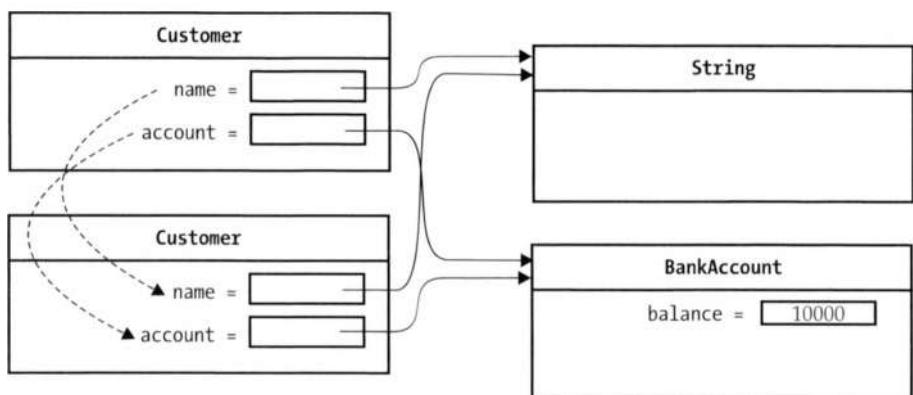


Ecco come invocarlo:

```
BankAccount clonedAccount = (BankAccount) account.clone();
```

Il tipo del riferimento che viene restituito dal metodo `clone` è `Object`, quindi, quando invocate il metodo, dovete usare un cast per segnalare al compilatore che il riferimento restituito da `account.clone()` è, in effetti, di tipo `BankAccount`.

**Figura 5**  
Il metodo `Object.clone`  
fa una copia superficiale



Per la realizzazione del metodo `clone` nelle vostre classi, `Object.clone` è un buon punto di partenza: tale metodo crea un nuovo oggetto dello stesso tipo dell'oggetto originario e copia automaticamente nell'oggetto clonato le variabili di esemplare dell'oggetto originario. Ecco un primo tentativo di realizzazione del metodo `clone` per la classe `BankAccount`:

```
public class BankAccount
{
    ...
    public Object clone()
    {
        ...
    }
}
```

```

{
    // incompleto
    Object clonedAccount = super.clone();
    return clonedAccount;
}
}

```

Questo metodo, però, va usato con cautela, perché si limita a delegare il problema della clonazione, senza risolverlo completamente. In particolare, se un oggetto contiene un riferimento a un altro oggetto, il metodo `Object.clone` crea una copia di tale riferimento, non un clone dell'oggetto stesso. La figura mostra come funziona il metodo `Object.clone` con un oggetto di tipo `Customer` che ha due riferimenti, uno a un oggetto di tipo `String` e un altro a un oggetto di tipo `BankAccount`. Come potete vedere, il metodo `Object.clone` copia i riferimenti nell'oggetto `Customer` clonato, anziché clonare gli oggetti a cui questi si riferiscono: una copia di questo tipo viene detta **copia superficiale** (*shallow copy*).

C'è un motivo che induce il metodo `Object.clone` a non clonare sistematicamente tutti i sotto-oggetti: in alcune situazioni questo non è necessario. Ad esempio, se un oggetto contiene un riferimento a una stringa, non vi è alcun pregiudizio nel copiare tale riferimento, dal momento che il contenuto delle stringhe, in Java, non può mai essere modificato. Il metodo `Object.clone` fa la cosa giusta se un oggetto contiene solo numeri, valori booleani o stringhe; bisogna, però, usarlo con cautela nel caso in cui un oggetto contenga riferimenti a oggetti modificabili.

Per questo motivo nel metodo `Object.clone` sono state predisposte due protezioni, per garantire che non venga utilizzato accidentalmente. Per prima cosa, il metodo è dichiarato `protected` (consultate Argomenti avanzati 9.5): questo vi impedisce di invocare `x.clone()` per sbaglio, se la classe a cui appartiene `x` non ha sovrascritto `clone` come metodo pubblico.

Quale seconda precauzione, `Object.clone` controlla che l'oggetto da clonare implementi l'interfaccia `Cloneable`; in caso contrario, lancia un'eccezione. In pratica, il metodo `Object.clone` assomiglia a questo:

```

public class Object
{
    protected Object clone() throws CloneNotSupportedException
    {
        if (this instanceof Cloneable)
        {
            // copia le variabili di esemplare
            ...
        }
        else
        {
            throw new CloneNotSupportedException();
        }
        ...
    }
}

```

Purtroppo tutte queste protezioni implicano che i legittimi invocanti di `Object.clone()` paghino un prezzo, ovvero che debbano intercettare l'eccezione (si veda il Capitolo 11) anche se la loro classe implementa `Cloneable`.

```

public class BankAccount implements Cloneable
{
    ...
    public Object clone()
    {
        try
        {
            return super.clone();
        }
        catch (CloneNotSupportedException e)
        {
            // non può accadere perché implementiamo Cloneable,
            // ma dobbiamo intercettarla comunque
            return null;
        }
    }
}

```

Se un oggetto contiene un riferimento a un altro oggetto modificabile, dovete invocare `clone` con tale riferimento. Per esempio, supponete che la classe `Customer` abbia una variabile di esemplare di tipo `BankAccount`. In questo caso, potete realizzare `Customer.clone` nel modo seguente:

```

public class Customer implements Cloneable
{
    private String name;
    private BankAccount account;
    ...
    public Object clone()
    {
        try
        {
            Customer cloned = (Customer) super.clone();
            cloned.account = (BankAccount) account.clone();
            return cloned;
        }
        catch (CloneNotSupportedException e)
        {
            // non può accadere perché implementiamo Cloneable
            return null;
        }
    }
}

```

In generale, l'implementazione del metodo `clone` richiede queste fasi:

- Dichiarare che la classe implementa l'interfaccia `Cloneable`.
- Nel metodo `clone`, invocare `super.clone()`, catturando `CloneNotSupportedException` se la superclasse è `Object`.
- Clonare le variabili di esemplare che fanno riferimento a oggetti modificabili.

## 10.4 Usare interfacce di smistamento (callback)

In questo paragrafo introdurremo il concetto di “smistamento” o “richiamata” (*callback*), mostrando come possa rendere il metodo `average` ancora più flessibile, e vedremo come si realizzi un tale smistamento in Java mediante le interfacce.

Per capire il motivo per cui vogliamo migliorare ancora il metodo `average`, considerate queste importanti limitazioni dovute all'utilizzo dell'interfaccia `Measurable`.

- Potete aggiungere l'implementazione dell'interfaccia `Measurable` soltanto a classi che sono sotto il vostro controllo. Se volete elaborare un insieme di oggetti di tipo `Rectangle`, non potete fare in modo che la classe `Rectangle` implementi un'altra interfaccia oltre a quelle che già realizza: è una classe di libreria, che non potete modificare.
- Potete “misurare” un oggetto in un unico modo: se volete analizzare un insieme di automobili sia in base alla velocità che in base al prezzo, siete bloccati.

Ripensiamo, quindi, al metodo `average`: misura oggetti, richiedendo che siano di tipo `Measurable`. La responsabilità della misurazione ricade sugli oggetti stessi: da questo derivano le limitazioni che abbiamo notato.

Sarebbe meglio se potessimo fornire al metodo `average` i dati di cui fare la media e, separatamente, un metodo che misuri gli oggetti. In questo modo, elaborando rettangoli potremmo fornire un metodo che calcoli l'area di un rettangolo, mentre elaborando automobili potremmo fornire un metodo che ispezioni il prezzo di un'automobile.

Un metodo di questo tipo viene chiamato ***callback*** e costituisce un meccanismo per confezionare e trasferire un blocco di codice che possa essere invocato più tardi.

In alcuni linguaggi di programmazione esiste la possibilità di dichiarare esplicitamente tali `callback`, sotto forma di blocchi di codice o di nomi di metodi, ma Java è un linguaggio orientato agli oggetti, per cui dobbiamo trasformare questo meccanismo in un oggetto, partendo dalla dichiarazione di un'interfaccia per il `callback`:

```
public interface Measurer
{
    double measure(Object anObject);
}
```

Il metodo `measure` misura un oggetto e restituisce il valore ottenuto mediante tale misurazione. In questa definizione usiamo la proprietà, che hanno tutti gli oggetti, di poter essere convertiti nel tipo `Object`.

Il codice che effettua l'invocazione del metodo di smistamento riceve un oggetto di una classe che implementa questa interfaccia. Nel nostro caso, il metodo `average` migliorato riceve un oggetto di tipo `Measurer`:

```
public static double average(Object[] objects, Measurer meas)
{
    double sum = 0;
    for (Object obj : objects)
```

Il meccanismo di *callback* consente di specificare codice che verrà eseguito in un secondo momento.

```

{
    sum = sum + meas.measure(obj);
}
if (objects.length > 0) { return sum / objects.length; }
else { return 0; }
}

```

Il metodo `average` richama semplicemente il metodo `measure` ogni volta che deve misurare un oggetto qualsiasi.

Infine, implementando l'interfaccia `Measurer` si progetta un misuratore per specifici oggetti. Ad esempio, ecco come misurare l'area di rettangoli. Definite la classe:

```

public class AreaMeasurer implements Measurer
{
    public double measure(Object anObject)
    {
        Rectangle aRectangle = (Rectangle) anObject;
        double area = aRectangle.getWidth() * aRectangle.getHeight();
        return area;
    }
}

```

Noteate che il metodo `measure` ha una variabile parametro di tipo `Object`, anche se questo particolare misuratore vuole misurare soltanto rettangoli. I tipi dei parametri del metodo devono corrispondere a quelli dichiarati nel metodo `measure` dell'interfaccia `Measurer`, per cui il parametro di tipo `Object` deve essere convertito in un `Rectangle` con un cast:

```
Rectangle aRectangle = (Rectangle) anObject;
```

Cosa potete fare con un oggetto di tipo `AreaMeasurer`? Vi serve per calcolare l'area di rettangoli. Costruite un oggetto di tipo `AreaMeasurer` e passatelo al metodo `average`:

```

Measurer areaMes = new AreaMeasurer();
Rectangle[] rects = {
    new Rectangle(5, 10, 20, 30),
    new Rectangle(10, 20, 30, 40)
};
double averageArea = average(rects, areaMes);

```

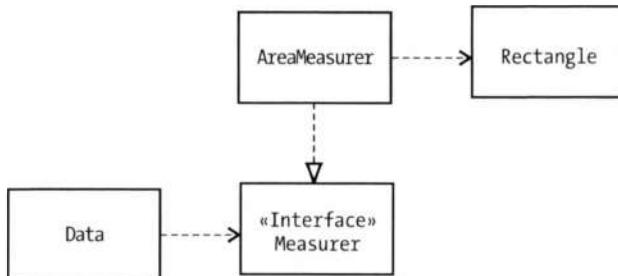
Il metodo `average` chiederà all'oggetto `AreaMeasurer` di misurare i rettangoli.

La Figura 6 mostra il diagramma UML delle classi e delle interfacce usate in questa soluzione. Come nella Figura 1, la classe `Data` (che contiene il metodo `average`) è disaccoppiata dalla classe `Rectangle`, di cui elabora oggetti. Tuttavia, diversamente da quanto accadeva nella Figura 1, la classe `Rectangle` non è più accoppiata a un'altra classe: per elaborare rettangoli, forniamo una piccola classe “ausiliaria”, `AreaMeasurer`, che ha solamente lo scopo di aiutare il metodo `average` a misurare gli oggetti.

Ecco il programma completo.

**Figura 6**

Diagramma UML della classe Data e dell'interfaccia Measurer



### File Measurer.java

```

1  /**
2   * Descrive classi i cui esemplari possono misurare altri oggetti.
3  */
4  public interface Measurer
5  {
6      /**
7       * Calcola la misura di un oggetto.
8       * @param anObject l'oggetto da misurare
9       * @return la misura
10    */
11   double measure(Object anObject);
12 }
  
```

### File AreaMeasurer.java

```

1 import java.awt.Rectangle;
2
3 /**
4  * Gli oggetti di questa classe misurano rettangoli in base alla loro area.
5  */
6 public class AreaMeasurer implements Measurer
7 {
8     public double measure(Object anObject)
9     {
10         Rectangle aRectangle = (Rectangle) anObject;
11         double area = aRectangle.getWidth() * aRectangle.getHeight();
12         return area;
13     }
14 }
  
```

### File Data.java

```

1 public class Data
2 {
3     /**
4      * Calcola la media delle misure di un insieme di oggetti.
5      * @param objects un array di oggetti
6      * @param meas il misuratore degli oggetti
7      * @return il valore medio delle misure
8      */
  
```

```

9   public static double average(Object[] objects, Measurer meas)
10  {
11      double sum = 0;
12      for (Object obj : objects)
13      {
14          sum = sum + meas.measure(obj);
15      }
16      if (objects.length > 0) { return sum / objects.length; }
17      else { return 0; }
18  }
19 }
```

### File MeasurerTester.java

```

1 import java.awt.Rectangle;
2
3 /**
4  * Questo programma illustra l'utilizzo di un Measurer.
5 */
6 public class MeasurerTester
7 {
8     public static void main(String[] args)
9     {
10         Measurer areaMeas = new AreaMeasurer();
11
12         Rectangle[] rects = new Rectangle[]
13         {
14             new Rectangle(5, 10, 20, 30),
15             new Rectangle(10, 20, 30, 40),
16             new Rectangle(20, 30, 5, 15)
17         };
18
19         double averageArea = Data.average(rects, areaMeas);
20         System.out.println("Average area: " + averageArea);
21         System.out.println("Expected: 625");
22     }
23 }
```

### Esecuzione del programma

Average area: 625.0  
Expected: 625



### Auto-valutazione

16. Immaginate di voler utilizzare il metodo `average` del Paragrafo 10.1 per trovare la lunghezza media di oggetti `String`. Perché ciò non è possibile?
17. Come si può utilizzare il metodo `average` di questo paragrafo per trovare la lunghezza media di oggetti `String`?
18. Perché il metodo `measure` dell'interfaccia `Measurer` ha un parametro in più del metodo `getMeasure` dell'interfaccia `Measurable`?
19. Scrivete un metodo, `max`, con tre parametri che trovi il maggiore tra due oggetti, usando un oggetto `Measurer` per confrontarli.

20. Scrivete un'invocazione del metodo progettato nella risposta alla domanda precedente che calcoli il maggiore tra due rettangoli, per poi visualizzare la sua larghezza e la sua altezza.

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R10.9, E10.8 e E10.9, al termine del capitolo.



## Note per Java 8 10.4

### Espressioni lambda

Nel paragrafo precedente avete visto come usare le interfacce per specificare varianti nel comportamento: il metodo `average` ha bisogno di misurare qualsiasi oggetto e lo fa invocando il metodo `measure` dell'oggetto `Measurer` che ha ricevuto.

Sfortunatamente, chi vuole invocare il metodo `average` ha molto lavoro da fare: deve definire una classe che implementi l'interfaccia `Measurer`, per poi costruirne un esemplare. La versione 8 di Java consente l'uso di una comoda scorciatoia, a patto che l'interfaccia abbia *un unico metodo astratto*. Una tale interfaccia viene chiamata **interfaccia funzionale**, perché il suo scopo è quello di definire una sola funzione, e l'interfaccia `Measurer` ne è un esempio.

Per specificare tale unica funzione si può usare una **espressione lambda**, cioè un'espressione che definisce in modo compatto i parametri ricevuti da un metodo e il valore che questo restituisce. Ecco un esempio:

```
(Object obj) -> ((BankAccount) obj).getBalance()
```

Questa espressione definisce un metodo che, dato un oggetto, lo converte tramite un cast in un riferimento di tipo `BankAccount` e ne restituisce il saldo.

Il termine “espressione lambda” deriva da una consuetudine della notazione matematica, che usa la lettera greca lambda ( $\lambda$ ) invece del simbolo  $\rightarrow$  (che, a sua volta, vuole ricordare graficamente una freccia diretta da sinistra a destra). In altri linguaggi di programmazione una tale espressione viene chiamata *espressione funzionale*.

Un'espressione lambda non può stare da sola, deve essere assegnata a una variabile il cui tipo sia un'interfaccia funzionale:

```
Measurer accountMeas = (Object obj) -> ((BankAccount) obj).getBalance();
```

A questo punto accadono tre cose:

1. Viene definita una classe che implementa l'interfaccia funzionale e il suo unico metodo astratto viene implementato usando l'espressione lambda.
2. Viene costruito un oggetto, esemplare di tale classe.
3. Alla variabile `accountMeas` viene assegnato un riferimento a tale oggetto.

Un'espressione lambda può anche essere fornita come argomento a un metodo: in questo caso la variabile parametro del metodo viene inizializzata con l'oggetto costruito durante la procedura appena delineata. Analizziamo, ad esempio, l'invocazione seguente:

```
double averageBalance = average(accounts,
    (Object obj) -> ((BankAccount) obj).getBalance());
```

Esattamente come nel caso precedente, viene costruito un oggetto, esemplare di una classe che implementa l'interfaccia `Measurer`. Tale oggetto viene poi utilizzato per inizializzare la variabile parametro `meas` del metodo `average`. Infatti, tale variabile parametro è definita di tipo `Measurer`:

```
public static double average(Object[] objects, Measurer meas)
{
    . . .
    sum = sum + meas.measure(obj);
    . . .
}
```

Il metodo `average` invoca il metodo `measure` con la variabile `meas` e questo provoca l'esecuzione del corpo dell'espressione lambda.

Nella sua forma più semplice, un'espressione lambda contiene un elenco di parametri, seguito dall'espressione che viene calcolata a partire da tali parametri. Se l'elaborazione da compiere è più complessa, si può scrivere un corpo del metodo nel modo consueto, racchiudendolo tra parentesi graffe e aggiungendo un enunciato `return`:

```
Measurer areaMeas = (Object obj) ->
{
    Rectangle r = (Rectangle) obj;
    return r.getWidth() * r.getHeight();
};
```

In linea teorica è forse più facile comprendere le espressioni lambda se le si immagina come una notazione particolarmente comoda per realizzare il meccanismo di *callback*. Prendiamo in esame un generico metodo che ha la necessità di invocare un frammento di codice che varia da un'invocazione all'altra. Questo risultato si può ottenere in questo modo:

1. Chi progetta il metodo definisce un'interfaccia che descrive l'obiettivo del codice che deve essere eseguito: un'interfaccia dotata di un unico metodo.
2. Il metodo riceve un parametro il cui tipo è quell'interfaccia e ne invoca l'unico metodo ogni volta che ha bisogno di invocare il frammento di codice variabile.
3. Chi invoca il metodo fornisce come argomento un'espressione lambda il cui corpo è proprio il codice che deve essere eseguito durante questa invocazione.

Vedrete ulteriori esempi di utilizzo delle espressioni lambda come gestori di eventi (nel Paragrafo 10.5) e come comparatori (nel Paragrafo 13.8).

## Argomenti avanzati 10.3

### Tipi interfaccia generici

Nel Paragrafo 10.3 avete visto come utilizzare la versione “semplice” del tipo interfaccia `Comparable`, ma, in effetti, l'interfaccia `Comparable` è un tipo parametrico, come il tipo `ArrayList`:

```
public interface Comparable<T>
{
    int compareTo(T other);
}
```

Il parametro di tipo, `T`, rappresenta il tipo degli oggetti che saranno accettati per fare confronti dal metodo `compareTo` di una classe che implementa questa interfaccia e solitamente tale tipo coincide con la classe stessa. Ad esempio, la classe `BankAccount` potrebbe implementare l'interfaccia `Comparable<BankAccount>`, in questo modo:

```
public class BankAccount implements Comparable<BankAccount>
{
    ...
    public int compareTo(BankAccount other)
    {
        return Double.compare(balance, other.balance);
    }
}
```

L'uso del parametro di tipo comporta un vantaggio: non c'è bisogno di un cast per convertire una variabile parametro di tipo `Object` nel tipo desiderato.

Analogamente, l'interfaccia `Measurer` può essere migliorata rendendola generica:

```
public interface Measurer<T>
{
    double measure(T anObject);
}
```

Il parametro di tipo specifica il tipo del parametro del metodo `measure`. Di nuovo, questo rende possibile evitare il cast da `Object` quando si implementa l'interfaccia:

```
public class AreaMeasurer implements Measurer<Rectangle>
{
    public double measure(Rectangle anObject)
    {
        return anObject.getWidth() * anObject.getHeight();
    }
}
```

## 10.5 Classi interne

La classe `AreaMeasurer` del paragrafo precedente è veramente banale: ne abbiamo bisogno soltanto perché il metodo `average` richiede un oggetto di una classe che implementi l'interfaccia `Measurer`. Quando avete una classe che serve a uno scopo molto limitato, come questo, potete dichiararla all'interno del metodo che ne ha bisogno:

```
public class MeasurerTester
{
    public static void main(String[] args)
    {
        class AreaMeasurer implements Measurer
```

```

{
    ...
}

Measurer areaMes = new AreaMeasurer();
double averageArea = Data.average(rects, areaMes);

...
}
}

```

Una classe interna viene dichiarata all'interno di un'altra classe.

Le classi interne sono solitamente utilizzate per classi con scopo molto limitato, che non hanno bisogno di essere visibili in altre zone del programma.

Una classe dichiarata all'interno di un'altra classe, come la classe `AreaMeasurer` di questo esempio, viene detta **classe interna** (*inner class*). Questa disposizione segnalerà al lettore del vostro programma che la classe `AreaMeasurer` non ha interesse al di fuori di questo metodo. Poiché una classe interna a un metodo non è una caratteristica accessibile pubblicamente, non c'è bisogno di documentarla in maniera estesa.

Si può anche dichiarare una classe all'interno di un'altra classe, ma al di fuori dei metodi di quest'ultima: in questo modo la classe interna sarà visibile a tutti i metodi della classe che la contiene.

```

public class MeasurerTester
{
    class AreaMeasurer implements Measurer
    {
        ...
    }

    public static void main(String[] args)
    {
        ...
        Measurer areaMes = new AreaMeasurer();
        double averageArea = Data.average(rects, areaMes);
        ...
    }
}

```

Quando compilate i file sorgenti di un programma che usa classi interne, guardate ai file di classe che vengono generati nella cartella del programma: vedrete che le classi interne vengono memorizzate in file con nomi curiosi, come `MeasurerTester$1AreaMeasurer.class`. I nomi esatti non sono importanti: ciò che importa è che il compilatore traduce una classe interna in un normale file di classe.



## Auto-valutazione

21. Perché si usa una classe interna invece di una classe normale?
22. Quando definireste una classe all'interno di un'altra classe ma all'esterno di tutti i metodi di quest'ultima?
23. Quanti file di classe vengono generati compilando il programma `MeasurerTester` visto in questo paragrafo?

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi E10.11 e E10.13, al termine del capitolo.

## Argomenti avanzati 10.4

---

### Classi anonime

Un'entità è *anonima* quando non possiede un nome. In un programma, qualcosa che si usa una volta sola generalmente non ha bisogno di un nome. Per esempio, se la nazione memorizzata nella variabile `belgium` non viene usata in altre zone del metodo, potete sostituirla con:

```
Country belgium = new Country("Belgium", 30510);
countries.add(belgium);
```

con il seguente:

```
countries.add(new Country("Belgium", 30510));
```

L'oggetto `new Country("Belgium", 30510)` è un **oggetto anonimo**. Ai programmati piacciono gli oggetti anonimi, perché non devono sforzarsi di trovare un nome per la variabile che ne memorizza il riferimento: se avete mai dovuto affrontare una decisione sofferta per capire se il riferimento a una moneta dovesse chiamarsi `c`, `dime` oppure `aCoin`, potete comprendere questo atteggiamento.

Spesso con le classi interne ci troviamo in una situazione simile: dopo aver costruito un unico oggetto di tipo `AreaMeasurer`, tale classe non viene più usata. In Java, è possibile dichiarare **classi anonime**, se tutto ciò di cui avete bisogno è un singolo esemplare della classe.

```
public static void main(String[] args)
{
    // costruzione di un oggetto di una classe anonima
    Measurer m = new Measurer()
        // la dichiarazione della classe inizia qui
    {
        public double measure(Object anObject)
        {
            Rectangle aRectangle = (Rectangle) anObject;
            return aRectangle.getWidth() * aRectangle.getHeight();
        }
    };
    . .
    double result = Data.average(rectangles, m);
    . .
}
```

Questo codice significa: costruisci un oggetto di una classe che implementa l'interfaccia `Measurer`, definendo il metodo `measure` come indicato. Questo stile era piuttosto utilizzato prima dell'introduzione delle espressioni lambda, che sono presenti in Java a partire dalla versione 8: oggi è più semplice usare, appunto, un'espressione lambda, per cui in questo libro non useremo classi anonime.

## 10.6 Oggetti semplificati

Un oggetto *semplicizzato* ("mock object") fornisce gli stessi servizi di un altro oggetto, ma in modo semplificato.

Quando state realizzando un programma costituito da più classi, spesso ne volete collaudare alcune prima di averlo portato a termine: a questo scopo è molto efficace fare uso di **oggetti semplificati** (*mock object*), che forniscono gli stessi servizi previsti da un altro oggetto, ma in modo estremamente più semplice.

Prendiamo in esame un'applicazione che gestisce un registro scolastico, con i risultati di varie prove di più studenti. Per questo usiamo la classe `GradeBook` avente metodi come:

```
public void addScore(int studentId, double score)
public double getAverageScore(int studentId)
public void save(String filename)
```

Consideriamo, ora, la classe `GradingProgram`, che elabora oggetti di tipo `GradeBook`, invocandone metodi: vorremmo poterla collaudare prima di avere una classe `GradeBook` pienamente funzionante.

Per poterlo fare dichiariamo un'interfaccia avente gli stessi metodi della classe `GradeBook`; spesso, per convenzione, si usa la lettera `I` maiuscola come prefisso per il nome di tale interfaccia.

```
public interface IGradeBook
{
    void addScore(int studentId, double score);
    double getAverageScore(int studentId);
    void save(String filename);
    ...
}
```

La classe semplificata e la classe completa implementano la medesima interfaccia.

Il programma `GradingProgram` deve usare *soltanto* questa interfaccia, senza fare mai riferimento alla classe `GradeBook`; quest'ultima, ovviamente, implementa l'interfaccia `IGradeBook`, ma, come già detto, potrebbe non essere ancora completamente definita.

Nel frattempo, prima di terminare lo sviluppo della classe `GradeBook`, produciamo una sua realizzazione semplificata, facendo ipotesi drastiche: ad esempio, memorizzare i dati in un file non è un'azione davvero necessaria per collaudare l'interfaccia utilizzata dall'utente; inoltre, possiamo temporaneamente limitarci alla gestione di un solo studente.

```
public class MockGradeBook implements IGradeBook
{
    private ArrayList<Double> scores;

    public void addScore(int studentId, double score)
    {
        // ignora studentId
        scores.add(score);
    }

    public double getAverageScore(int studentId)
    {
        double total = 0;
        for (double x : scores) { total = total + x; }
        return total / scores.size();
    }
}
```

```
    }
    public void save(String filename)
    {
        // non fa nulla
    }
    ...
}
```

A questo punto possiamo costruire un esemplare di `MockGradeBook` e usarlo nella classe `GradingProgram`, riuscendo così a collaudarla immediatamente. Quando, poi, saremo pronti per collaudare la vera classe `GradeBook`, useremo un suo esemplare. Evitate, comunque, di cancellare la classe semplificata: sarà utile per il collaudo regressivo.



### Auto-valutazione

24. Perché è necessario che la classe semplificata e la classe effettiva implementino la medesima interfaccia?
25. Perché l'utilizzo degli oggetti semplificati è particolarmente efficace quando le classi `GradeBook` e `GradingProgram` sono sviluppate da due programmatore diversi?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi P10.19 e P10.20, al termine del capitolo.

## 10.7 Gestione di eventi

In questo paragrafo e nei successivi torniamo sulla programmazione grafica: vedrete come usare i tipi interfaccia nella progettazione di interfacce grafiche per l'interazione con l'utente.

Nelle applicazioni scritte finora i dati vengono forniti in ingresso dall'utente sotto il controllo del *programma*, che chiede all'utente di inserire i dati in un ordine specifico: per esempio, può chiedere di immettere per prima cosa un nome, seguito da un importo in dollari. I programmi che usate quotidianamente, però, non funzionano in questo modo: in un'applicazione dotata di una interfaccia grafica, il controllo è in mano all'*utente*, che può usare tanto il mouse quanto la tastiera e può intervenire sui molti elementi grafici dell'interfaccia, in qualsiasi ordine desideri. Ad esempio, l'utente può inserire informazioni in campi di testo, aprire menu a discesa, premere pulsanti e trascinare barre di scorrimento, in qualsiasi ordine, e il programma deve rispondere a tali comandi, in qualunque ordine arrivino. Chiaramente, dover gestire molti possibili dati che giungano in ordine casuale è molto più difficile che obbligare semplicemente l'utente a fornire i dati secondo un ordine prestabilito.

In questi paragrafi imparerete a scrivere, in Java, programmi capaci di reagire a eventi generati dall'utente attraverso l'interfaccia grafica del programma, ad esempio selezionando una voce in un menu o premendo un pulsante del mouse. La raccolta di strumenti Java per la gestione delle finestre (*Java windowing toolkit*) prevede un meccanismo molto articolato che permette a un programma di specificare gli eventi a cui è interessato, oltre a indicare quali oggetti devono essere avvertiti quando si verifica uno di tali eventi.

Gli eventi dell'interfaccia utente comprendono pressioni di tasti, movimenti del mouse e pressioni di suoi pulsanti, selezioni di voci in menu e così via.

## 10.7.1 Ricezione di eventi

Quando l'utente di un programma grafico digita caratteri sulla tastiera o utilizza il mouse in un punto qualunque all'interno di una delle finestre del programma, il gestore Java delle finestre invia una notifica al programma per segnalare che si è verificato un **evento**. Il gestore delle finestre genera un numero enorme di eventi: per esempio, ogni volta che il mouse percorre una minima distanza all'interno di una finestra, viene generato un evento di tipo "movimento del mouse", mentre in seguito a ogni pressione di un pulsante del mouse vengono generati due eventi, "mouse premuto" e "mouse rilasciato". Inoltre, quando l'utente seleziona un pulsante grafico o una voce di un menu, vengono generati eventi a un livello di astrazione più elevato.

La maggior parte dei programmi non vuole essere sommersa da eventi inutili. Pensate, ad esempio, a ciò che succede quando viene selezionata con il mouse una voce di un menu: il mouse si posiziona sulla voce di menu, poi viene premuto il pulsante del mouse, che, infine, viene rilasciato. Piuttosto che ricevere grandi quantità di eventi del mouse non rilevanti per la propria elaborazione, un programma può specificare che gli interessano solamente le selezioni di voci di menu, ignorando tutti i sottostanti eventi relativi al mouse. Se, invece, l'interazione dell'utente con il mouse serve a disegnare forme grafiche su un canovaccio virtuale, sarà necessario tenere traccia con grande attenzione di tutti i singoli eventi del mouse.

Ogni programma deve indicare quali eventi gradisce ricevere, installando opportuni oggetti che assumono il ruolo di **ricevitori di eventi** (*event listener*). Ciascun oggetto che funge da ricevitore è esemplare di una classe progettata da voi e i suoi metodi contengono le istruzioni che vanno eseguite quando accadono quei particolari eventi che si intendono ricevere.

Per installare un ricevitore dovete conoscere la **sorgente dell'evento** (*event source*), che è il componente dell'interfaccia grafica che genera quel particolare evento. Un oggetto che funge da ricevitore di eventi va aggiunto alle sorgenti di evento appropriate, dopodiché, quando accade l'evento che interessa, la sua sorgente invoca gli opportuni metodi di tutti i ricevitori a essa connessi.

Tutto ciò può certamente sembrare un po' astruso, quindi analizzeremo in dettaglio un programma estremamente semplice che visualizza un messaggio ogni volta che viene premuto un pulsante, come si può vedere nella Figura 7. I ricevitori interessati agli eventi di un pulsante grafico devono essere esemplari di una classe che implementa l'interfaccia `ActionListener`:

```
public interface ActionListener
{
    void actionPerformed(ActionEvent event);
}
```

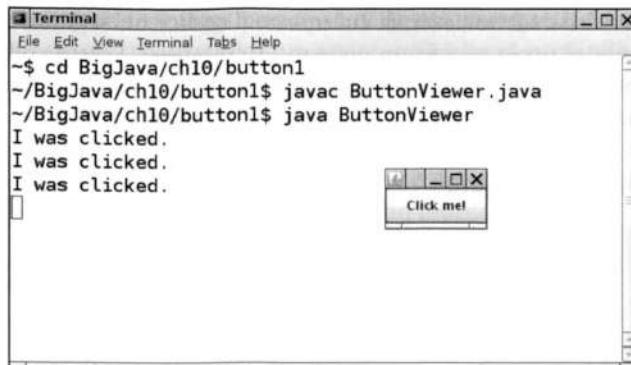
Questa particolare interfaccia ha un unico metodo, `actionPerformed`. Il vostro compito consiste nel progettare una classe il cui metodo `actionPerformed` contenga le istruzioni che volete che vengano eseguite ogni volta che viene premuto il pulsante.

Un ricevitore di eventi è esemplare di una classe progettata dal programmatore dell'applicazione e i suoi metodi descrivono le azioni da compiere quando si verifica un particolare tipo di evento.

Le sorgenti di eventi generano segnalazioni relative agli eventi: quando ne avviene uno, lo segnalano a tutti i ricevitori interessati.

**Figura 7**

Programma con un ricevitore di azioni



Ecco un esempio molto semplice di una classe di questo tipo.

### File ClickListener.java

```

1 import java.awt.event.ActionEvent;
2 import java.awt.event.ActionListener;
3
4 /**
5      Un ricevitore di azioni che visualizza un messaggio.
6 */
7 public class ClickListener implements ActionListener
8 {
9     public void actionPerformed(ActionEvent event)
10    {
11        System.out.println("I was clicked");
12    }
13 }
```

Abbiamo ignorato la variabile parametro `event` del metodo `actionPerformed`: contiene ulteriori dettagli relativi all'evento, come l'istante in cui esso è avvenuto.

Dopo aver dichiarato la classe per il ricevitore di eventi, dobbiamo costruirne un esemplare e associarlo al pulsante grafico:

```
ClickListener listener = new ClickListener();
button.addActionListener(listener);
```

Ogni volta che il pulsante viene premuto, il gestore Java dell'ambiente grafico invoca il metodo

```
listener.actionPerformed(event);
```

e, di conseguenza, viene visualizzato il messaggio.

Potete pensare al metodo `actionPerformed` come a un ulteriore esempio di *callback*, simile al metodo `measure` dell'interfaccia `Measurer`. Il gestore Java dell'ambiente grafico invoca il metodo `actionPerformed` ogni volta che viene premuto il pulsante, così come il metodo `Data.average` invoca il metodo `measure` ogni volta che ha bisogno di misurare un oggetto.

Usate componenti di tipo JButton  
per realizzare pulsanti grafici  
e associate un ActionListener  
a ogni pulsante.

La classe `ButtonViewer`, di cui trovate il codice nel seguito, costruisce un frame con un pulsante, al quale associa un oggetto `ClickListener`. Potete collaudare questo programma aprendo una finestra di console, eseguendovi `ButtonViewer`, premendo il pulsante grafico con il mouse e osservando i messaggi visualizzati.

### File `ButtonViewer.java`

```

1 import java.awt.event.ActionListener;
2 import javax.swing.JButton;
3 import javax.swing.JFrame;
4
5 /**
6  * Questo programma mostra come si installa un ricevitore di azioni.
7 */
8 public class ButtonViewer
9 {
10     private static final int FRAME_WIDTH = 100;
11     private static final int FRAME_HEIGHT = 60;
12
13     public static void main(String[] args)
14     {
15         JFrame frame = new JFrame();
16         JButton button = new JButton("Click me!");
17         frame.add(button);
18
19         ActionListener listener = new ClickListener();
20         button.addActionListener(listener);
21
22         frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
23         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24         frame.setVisible(true);
25     }
26 }
```

#### 10.7.2 Classi interne come ricevitori di eventi

Nel paragrafo precedente avete visto che il codice che deve essere eseguito in risposta alla selezione di un pulsante grafico viene inserito in una classe che funge da ricevitore di eventi. Una classe di questo tipo viene molto spesso realizzata come classe interna, in questo modo:

```

JButton button = new JButton(...);

// questa classe interna viene definita nel metodo
// in cui si trova la variabile che contiene il pulsante
class MyListener implements ActionListener
{
    . . .

}

ActionListener listener = new MyListener();
button.addActionListener(listener);
```

Due sono i vantaggi principali di questa strategia. Innanzitutto, banalmente, le classi che ricevono eventi tendono a essere molto brevi e, usando questo stile, la classe interna si viene a trovare nel posto esatto in cui serve al suo scopo, senza creare confusione nella restante parte del progetto. Inoltre, una classe interna ha un'interessante caratteristica: i suoi metodi possono accedere alle variabili dichiarate nei blocchi che la contengono. Da questo punto di vista, la dichiarazione di un metodo in una classe interna si comporta in modo simile a quanto avviene in un blocco annidato.

I metodi di una classe interna possono accedere alle variabili dell'ambito di visibilità circostante.

Questa caratteristica è molto utile nella realizzazione di gestori di eventi, perché consente alla classe interna di accedere alle variabili di cui necessita senza che ci sia bisogno di passarle come parametri a un costruttore della classe o a uno dei suoi metodi.

Vediamo un esempio, nel quale immaginiamo di voler accreditare gli interessi a un conto bancario ogni volta che viene premuto un pulsante.

```
 JButton button = new JButton("Add Interest");
 final BankAccount account = new BankAccount(INITIAL_BALANCE);

 // questa classe interna viene dichiarata all'interno del
 // metodo in cui si trovano le variabili account e button
 class AddInterestListener implements ActionListener
 {
     public void actionPerformed(ActionEvent event)
     {
         // il metodo del ricevitore accede alla variabile
         // account visibile nel blocco circostante
         double interest = account.getBalance() * INTEREST_RATE / 100;
         account.deposit(interest);
     }
 }

 ActionListener listener = new AddInterestListener();
 button.addActionListener(listener);
```

Le variabili locali a cui si accede da un metodo di una classe interna non devono essere modificate dopo essere state inizializzate.

C'è, però, un vincolo tecnico da conoscere: nelle versioni di Java precedenti alla versione 8, una classe interna poteva accedere a variabili *locali* dell'ambito di visibilità circostante solo se erano state dichiarate *final*, mentre a partire dalla versione 8 la variabile deve solamente essere *effettivamente costante*, cioè si deve comportare come una variabile *final* (senza essere modificata dopo l'inizializzazione), ma non deve essere necessariamente dichiarata *final*. Nel nostro esempio la variabile *account* si riferisce sempre al medesimo conto bancario: a beneficio di lettori che usano Java nella versione 7 o precedente, l'abbiamo dichiarata *final*.

Una classe interna può anche accedere alle variabili *di esemplare* della classe circostante, nuovamente con una restrizione: la variabile di esemplare deve appartenere all'oggetto che ha costruito l'esemplare di classe interna. Se l'esemplare di classe interna è stato costruito all'interno di un metodo statico, può accedere alle sole variabili statiche circostanti.

Ecco, infine, il codice sorgente del programma.

### File InvestmentViewer1.java

```
 1 import java.awt.event.ActionEvent;
 2 import java.awt.event.ActionListener;
 3 import javax.swing.JButton;
 4 import javax.swing.JFrame;
```

```
5
6  /**
7   * Questo programma illustra il funzionamento di una classe
8   * interna che accede a una variabile di un blocco circostante.
9  */
10 public class InvestmentViewer1
11 {
12     private static final int FRAME_WIDTH = 120;
13     private static final int FRAME_HEIGHT = 60;
14
15     private static final double INTEREST_RATE = 10;
16     private static final double INITIAL_BALANCE = 1000;
17
18     public static void main(String[] args)
19     {
20         JFrame frame = new JFrame();
21
22         // il pulsante che fa partire l'elaborazione
23         JButton button = new JButton("Add Interest");
24         frame.add(button);
25
26         // l'applicazione accredita gli interessi a questo conto bancario
27         final BankAccount account = new BankAccount(INITIAL_BALANCE);
28
29         class AddInterestListener implements ActionListener
30         {
31             public void actionPerformed(ActionEvent event)
32             {
33                 // il metodo del ricevitore accede alla variabile
34                 // account del blocco circostante
35                 double interest = account.getBalance() * INTEREST_RATE / 100;
36                 account.deposit(interest);
37                 System.out.println("balance: " + account.getBalance());
38             }
39         }
40
41         ActionListener listener = new AddInterestListener();
42         button.addActionListener(listener);
43
44         frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
45         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
46         frame.setVisible(true);
47     }
48 }
```

### Esecuzione del programma

```
balance: 1100.0
balance: 1210.0
balance: 1331.0
balance: 1464.1
```



## Auto-valutazione

26. Nel programma `ButtonViewer`, quali sono gli oggetti che rappresentano la sorgente di eventi e il ricevitore di eventi?
27. Perché è lecito assegnare un oggetto di tipo `ClickListener` a una variabile di tipo `ActionListener`?
28. In quali situazioni il metodo `actionPerformed` deve essere invocato dal programmatore?
29. Perché a volte un metodo di una classe interna ha bisogno di accedere a una variabile che si trova in un ambito di visibilità circostante?
30. Se una classe interna accede a una variabile locale che si trova in un ambito di visibilità circostante, quale speciale regola occorre applicare?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R10.16, R10.22 e E10.17, al termine del capitolo.



## Errori comuni 10.3

### Implementare un metodo modificando il tipo dei suoi parametri

Quando implementate un'interfaccia dovete dichiarare ciascun metodo *esattamente* come specificato nell'interfaccia stessa: fare piccole modifiche accidentali ai tipi dei parametri è un errore piuttosto frequente. Ecco un classico esempio:

```
class MyListener implements ActionListener
{
    public void actionPerformed()
        // ahi, abbiamo dimenticato il parametro ActionEvent
    {
        ...
    }
}
```

Per quanto riguarda il compilatore, a questa classe manca la definizione del metodo

```
public void actionPerformed(ActionEvent event)
```

Dovete leggere con cura il messaggio d'errore, facendo attenzione ai tipi dei parametri: scoprirete così il vostro errore.



## Errori comuni 10.4

### Tentare di invocare i metodi dei ricevitori

Alcuni studenti cercano di scrivere codice che invoca esplicitamente i metodi dei ricevitori di eventi:

```
ActionEvent event = new ActionEvent(. . .); // non fate!
listener.actionPerformed(event);
```

Non si dovrebbe mai invocare un metodo di un ricevitore di eventi: tale metodo viene invocato dal gestore Java delle finestre quando l'utente del programma ha fatto in modo che l'evento accada (ad esempio, premendo il pulsante del mouse).



## Note per Java 8 10.4

### Le espressioni lambda per la gestione degli eventi

La sezione Note per Java 8 10.4 ha illustrato l'uso delle espressioni lambda come esemplari di classi che implementano un'interfaccia funzionale, cioè un'interfaccia dotata di un unico metodo astratto. Tra queste interfacce figurano certamente i gestori di eventi, come `ActionListener`.

Ad esempio, invece di definire la classe `ClickListener`, per poi associarne un esemplare a un pulsante grafico, gli si può semplicemente associare il ricevitore, in questo modo:

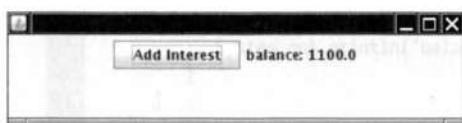
```
button.addActionListener(  
    (ActionEvent event) -> System.out.println("I was clicked."));
```

## 10.8 Costruire applicazioni dotate di pulsanti

In questo paragrafo vedrete come si delinea un'applicazione grafica dotata di pulsanti. Aggiungeremo un pulsante al nostro semplice programma che visualizza investimenti: ogni volta che viene premuto il pulsante vengono accreditati sul conto bancario gli interessi maturati e viene visualizzato il saldo aggiornato, come si può vedere nella Figura 8.

**Figura 8**

Un'applicazione dotata di pulsante grafico



Per prima cosa costruiamo un oggetto di tipo `JButton`, fornendo al costruttore l'etichetta del pulsante:

```
JButton button = new JButton("Add Interest");
```

Abbiamo anche bisogno di un componente dell'interfaccia grafica che visualizzi un messaggio: il saldo attuale del conto bancario. Un componente di questo tipo è un'*etichetta* (*label*) e al costruttore di `JLabel` viene fornita la stringa contenente il messaggio che deve essere visualizzato inizialmente, in questo modo:

```
JLabel label = new JLabel("balance = " + account.getBalance());
```

Usate un contenitore di tipo `JPanel` per raggruppare insieme più componenti dell'interfaccia grafica.

La finestra principale della nostra applicazione contiene sia il pulsante sia l'etichetta, ma non possiamo aggiungere banalmente i due componenti direttamente al frame, perché verrebbero posizionati uno sopra l'altro, coprendosi. La soluzione consiste nell'utilizzo di un **pannello**, cioè un contenitore per componenti dell'interfaccia utente, per poi

aggiungere al frame proprio il pannello, dopo aver aggiunto a quest'ultimo i due componenti:

```
JPanel panel = new JPanel();
panel.add(button);
panel.add(label);
frame.add(panel);
```

Le azioni conseguenti alla pressione di un pulsante grafico vanno specificate mediante classi che implementano l'interfaccia ActionListener.

A questo punto siamo pronti per la parte difficile: il ricevitore di eventi che gestisce le pressioni del pulsante. Come nel paragrafo precedente, è necessario definire una classe che implementi l'interfaccia `ActionListener`, inserendo l'azione desiderata all'interno del suo metodo `actionPerformed`. La nostra classe di gestione degli eventi accredita gli interessi sul conto e ne visualizza il saldo aggiornato:

```
class AddInterestListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        double interest = account.getBalance() * INTEREST_RATE / 100;
        account.deposit(interest);
        label.setText("balance = " + account.getBalance());
    }
}
```

Dobbiamo fare attenzione a un piccolo dettaglio tecnico: il metodo `actionPerformed` usa le variabili `account` e `label`, che sono variabili locali del metodo `main` del programma di visualizzazione di investimenti bancari, non variabili di esemplare della classe `AddInterestListener`. Quindi, se usiamo una versione di Java precedente alla versione 8, dobbiamo dichiarare `final` le variabili `account` e `label`, in modo che il metodo `actionPerformed` vi possa accedere.

Mettiamo ora insieme la varie parti:

```
public static void main(String[] args)
{
    ...
    JButton button = new JButton("Add Interest");
    final BankAccount account = new BankAccount(INITIAL_BALANCE);
    final JLabel label = new JLabel("balance = " + account.getBalance());

    class AddInterestListener implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            double interest = account.getBalance() * INTEREST_RATE / 100;
            account.deposit(interest);
            label.setText("balance = " + account.getBalance());
        }
    }

    ActionListener listener = new AddInterestListener();
    button.addActionListener(listener);
    ...
}
```

Con un po' di esperienza imparerete a leggere codice di questo tipo come se fosse un testo: "Quando viene premuto il pulsante, aggiungi gli interessi e imposta il testo dell'etichetta".

Ecco il programma completo, che mostra come si aggiungono più componenti a un frame, usando un pannello, e come si progettano ricevitori di eventi sotto forma di classi interne.

### File InvestmentViewer2.java

```
1 import java.awt.event.ActionEvent;
2 import java.awt.event.ActionListener;
3 import javax.swing.JButton;
4 import javax.swing.JFrame;
5 import javax.swing.JLabel;
6 import javax.swing.JPanel;
7
8 /**
9  * Questo programma visualizza la crescita di un investimento.
10 */
11 public class InvestmentViewer2
12 {
13     private static final int FRAME_WIDTH = 400;
14     private static final int FRAME_HEIGHT = 100;
15
16     private static final double INTEREST_RATE = 10;
17     private static final double INITIAL_BALANCE = 1000;
18
19     public static void main(String[] args)
20     {
21         JFrame frame = new JFrame();
22
23         // il pulsante che fa partire l'elaborazione
24         JButton button = new JButton("Add Interest");
25
26         // l'applicazione accredita gli interessi su questo conto bancario
27         final BankAccount account = new BankAccount(INITIAL_BALANCE);
28
29         // l'etichetta che visualizza i risultati
30         final JLabel label = new JLabel("balance = " + account.getBalance());
31
32         // il pannello che contiene i componenti dell'interfaccia grafica
33         JPanel panel = new JPanel();
34         panel.add(button);
35         panel.add(label);
36         frame.add(panel);
37
38         class AddInterestListener implements ActionListener
39         {
40             public void actionPerformed(ActionEvent event)
41             {
42                 double interest = account.getBalance() * INTEREST_RATE / 100;
43                 account.deposit(interest);
44                 label.setText("balance = " + account.getBalance());
45             }
46         }
47 }
```

```
48     ActionListener listener = new AddInterestListener();
49     button.addActionListener(listener);
50
51     frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
52     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
53     frame.setVisible(true);
54 }
55 }
```

## Auto-valutazione

31. Come si fa a visualizzare il messaggio "balance: . . ." alla sinistra del pulsante "Add Interest"?
32. Perché non è necessario dichiarare final anche la variabile button?

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi E10.18, E10.19 e E10.20, al termine del capitolo.

## Errori comuni 10.5

---

### Dimenticarsi di associare un ricevitore

Se, eseguendo il vostro programma, scoprirete che i pulsanti sembrano non funzionare, controllate bene di aver associato il ricevitore per gli eventi dei pulsanti; lo stesso vale per gli altri componenti dell'interfaccia utente. È un errore che capita fin troppo spesso: progettare la classe ricevitore e la relativa azione di gestione degli eventi senza associarne un esemplare alla sorgente dell'evento stesso.

## Suggerimenti per la programmazione 10.2

---

### Non usate un contenitore come ricevitore di eventi

In questo libro usiamo classi interne per i ricevitori di eventi dell'interfaccia grafica: questa soluzione funziona con molti diversi tipi di eventi, e, una volta padroneggiata la tecnica, non dovrete pensarci più. Inoltre, molti ambienti di sviluppo generano automaticamente codice sorgente per l'interfaccia utente usando classi interne, per cui è bene acquisire familiarità con esse.

Tuttavia, molti programmati evitano di usare classi per ricevitori di eventi e, invece, trasformano un contenitore (come un pannello o un frame) in un ricevitore. Ecco un esempio tipico, in cui il metodo actionPerformed è stato aggiunto alla classe che si occupa della visualizzazione, cioè il visualizzatore stesso implementa l'interfaccia ActionListener:

```
public class InvestmentViewer
    implements ActionListener // questo approccio è sconsigliato
{
    public InvestmentViewer()
    {
        JButton button = new JButton("Add Interest");
        button.addActionListener(this);
        ...
    }
}
```

```

    }

    public void actionPerformed(ActionEvent event)
    {
        ...
    }
    ...
}

```

Ora il metodo `actionPerformed` fa parte della classe `InvestmentViewer`, anziché trovarsi in una classe ricevitore separata, e il ricevitore viene installato usando `this`.

Questa tecnica ha due problemi rilevanti. Prima di tutto, separa la dichiarazione dei pulsanti dalle azioni relative ai pulsanti stessi, che, nel codice, risultano distanti. Secondariamente, non è facilmente scalabile: se la classe di visualizzazione ha due pulsanti, ciascuno dei quali genera eventi, il metodo `actionPerformed` deve scoprire la sorgente di ogni evento, usando codice noioso e introducendo, quindi, una nuova fonte di errori.

## 10.9 Eventi di temporizzazione

In questo paragrafo studieremo gli eventi di temporizzazione e mostreremo come essi consentano di realizzare semplici animazioni.

La classe `Timer` del pacchetto `javax.swing` genera una sequenza di eventi, separati da intervalli di tempo tutti uguali fra loro (potete immaginare un temporizzatore come un pulsante invisibile che viene premuto automaticamente e periodicamente). Ciò è utile ogni volta che volete disporre di oggetti aggiornati a intervalli regolari: ad esempio, in un'animazione potreste voler aggiornare una scena dieci volte al secondo, ridisegnando l'immagine per dare l'illusione del movimento.

Quando usate un temporizzatore dovete fornire al suo costruttore la frequenza degli eventi e un oggetto di una classe che implementi l'interfaccia `ActionListener` e che contenga l'azione desiderata all'interno del proprio metodo `actionPerformed`. Infine, fate partire il temporizzatore.

Un temporizzatore genera eventi a intervalli di tempo fissi.

```

class MyListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        // azione che verrà eseguita a ogni evento di temporizzazione
        ...
    }
}

MyListener listener = new MyListener();
Timer t = new Timer(interval, listener);
t.start();

```

A partire da questo momento, il temporizzatore invoca il metodo `actionPerformed` dell'oggetto `listener` a intervalli regolari, che durano un numero di millisecondi uguale a `interval`.

Il nostro programma esemplificativo mostrerà un rettangolo in movimento. Per prima cosa ci serve una classe, `RectangleComponent`, il cui metodo `moveRectangleBy` sposta il rettangolo della quantità specificata.

### File RectangleComponent.java

```
1 import java.awt.Graphics;
2 import java.awt.Graphics2D;
3 import java.awt.Rectangle;
4 import javax.swing.JComponent;
5
6 /**
7     Componente che visualizza un rettangolo che può essere spostato.
8 */
9 public class RectangleComponent extends JComponent
10 {
11     private static final int BOX_X = 100;
12     private static final int BOX_Y = 100;
13     private static final int BOX_WIDTH = 20;
14     private static final int BOX_HEIGHT = 30;
15
16     private Rectangle box;
17
18     public RectangleComponent()
19     {
20         // il rettangolo che viene disegnato dal metodo paintComponent
21         box = new Rectangle(BOX_X, BOX_Y, BOX_WIDTH, BOX_HEIGHT);
22     }
23
24     public void paintComponent(Graphics g)
25     {
26         Graphics2D g2 = (Graphics2D) g;
27         g2.draw(box);
28     }
29
30     /**
31      Sposta il rettangolo della quantità specificata.
32      @param dx l'entità dello spostamento nella direzione x
33      @param dy l'entità dello spostamento nella direzione y
34   */
35     public void moveRectangleBy(int dx, int dy)
36     {
37         box.translate(dx, dy);
38         repaint();
39     }
40 }
```

Il metodo `repaint` chiede a un componente di ridisegnare se stesso: invocarlo ogni volta che modificate le forme grafiche che vengono disegnate dal metodo `paintComponent`.

Notate l'invocazione di `repaint` nel metodo `moveRectangleBy`: è necessaria per essere certi che il componente venga ridisegnato dopo aver modificato lo stato del rettangolo. Ricordate sempre che l'oggetto che rappresenta il componente non contiene i pixel che vengono visualizzati, ma contiene solamente un oggetto di tipo `Rectangle`, il quale a sua volta contiene quattro numeri. L'invocazione di `translate` aggiorna i valori delle coordinate del rettangolo, mentre l'invocazione di `repaint` provoca, a sua volta, l'invocazione del metodo

`paintComponent`, che, infine, ridisegna il componente, facendo in modo che il rettangolo appaia nella posizione aggiornata.

Il metodo `actionPerformed` del ricevitore di eventi di temporizzazione ha il semplice compito di invocare `component.moveRectangleBy(1, 1)`: ciò sposta il rettangolo di un pixel verso il basso e verso destra. Dal momento che il metodo `actionPerformed` viene invocato dieci volte al secondo, il movimento del rettangolo all'interno del frame appare fluido.

### File RectangleFrame.java

```

1 import java.awt.event.ActionEvent;
2 import java.awt.event.ActionListener;
3 import javax.swing.JFrame;
4 import javax.swing.Timer;
5
6 /**
7  * Questo frame contiene un rettangolo che si sposta.
8 */
9 public class RectangleFrame extends JFrame
10 {
11     private static final int FRAME_WIDTH = 300;
12     private static final int FRAME_HEIGHT = 400;
13
14     private RectangleComponent scene;
15
16     class TimerListener implements ActionListener
17     {
18         public void actionPerformed(ActionEvent event)
19         {
20             scene.moveRectangleBy(1, 1);
21         }
22     }
23
24     public RectangleFrame()
25     {
26         scene = new RectangleComponent();
27         add(scene);
28
29         setSize(FRAME_WIDTH, FRAME_HEIGHT);
30
31         ActionListener listener = new TimerListener();
32
33         final int DELAY = 100; // millisecondi tra due eventi
34         Timer t = new Timer(DELAY, listener);
35         t.start();
36     }
37 }
```

### File RectangleViewer.java

```

1 import javax.swing.JFrame;
2
3 /**
4  * Questo programma visualizza un rettangolo in movimento.
5 */
```

```

6 public class RectangleViewer
7 {
8     public static void main(String[] args)
9     {
10         JFrame frame = new RectangleFrame();
11         frame.setTitle("An animated rectangle");
12         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13         frame.setVisible(true);
14     }
15 }
```



## Auto-valutazione

33. Perché un temporizzatore ha bisogno di un ricevitore di eventi?
34. Cosa succederebbe se nel metodo `moveRectangleBy` mancasse l'invocazione del metodo `repaint`?

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi E10.27 e E10.28, al termine del capitolo.



## Errori comuni 10.6

### Dimenticarsi di ridisegnare

Quando i vostri gestori di eventi modificano i dati di un componente disegnato, occorre fare attenzione. Dopo che avete modificato i dati, il componente non viene automaticamente ridisegnato in base a tali nuovi dati: occorre invocare il metodo `repaint` del componente all'interno del gestore di eventi o nei metodi modificatori del componente stesso. In conseguenza di ciò, al momento opportuno verrà invocato il metodo `paintComponent` del componente, che riceverà un opportuno oggetto di tipo `Graphics`. Notate, però, che non dovreste mai invocare direttamente il metodo `paintComponent`.

Questo problema riguarda solamente i componenti disegnati da voi. Quando modificate le proprietà di uno dei componenti standard di Swing, come `JLabel`, il componente viene ridisegnato automaticamente.

## 10.10 Eventi del mouse

Per catturare gli eventi del mouse  
usate un ricevitore di eventi del mouse  
(*mouse listener*).

Se scrivete un programma che mostra dei disegni e volete che l'utente possa manipolarli usando il mouse, dovete elaborare eventi del mouse più complessi delle semplici pressioni di pulsanti o degli eventi periodici generati da un temporizzatore.

Un ricevitore di eventi del mouse deve realizzare l'interfaccia `MouseListener`, che contiene questi cinque metodi:

```

public interface MouseListener
{
    void mousePressed(MouseEvent event);
        // Chiamato quando un pulsante del mouse
        // è stato premuto su un componente
    void mouseReleased(MouseEvent event);
```

```

    // Chiamato quando un pulsante del mouse
    // è stato rilasciato su un componente
    void mouseClicked(MouseEvent event);
        // Chiamato quando un pulsante del mouse
        // è stato premuto e rilasciato in rapida
        // successione su un componente ("click")
    void mouseEntered(MouseEvent event);
        // Chiamato quando il mouse entra in un componente
    void mouseExited(MouseEvent event);
        // Chiamato quando il mouse esce da un componente
}

```

I metodi `mousePressed` e `mouseReleased` vengono invocati ogni volta che un pulsante del mouse viene, rispettivamente, premuto o rilasciato. Se un pulsante viene premuto e rilasciato in rapida successione, senza che il mouse si sia spostato, allora viene invocato anche il metodo `mouseClicked`. I metodi `mouseEntered` e `mouseExited` possono essere utilizzati per visualizzare in modo speciale un componente dell'interfaccia utente quando il puntatore del mouse si trova al suo interno.

Il metodo usato più frequentemente è `mousePressed`: solitamente gli utenti si aspettano che le proprie azioni vengano elaborate non appena il pulsante del mouse viene premuto.

Per aggiungere a un componente un ricevitore di eventi del mouse si invoca il metodo `addMouseListener`:

```

public class MyMouseListener implements MouseListener
{
    // implementazione dei cinque metodi
}

MouseListener listener = new MyMouseListener()
component.addMouseListener(listener);

```

Nel nostro nuovo programma di esempio, ogni volta che l'utente preme e rilascia il pulsante del mouse (cioè fa "click") sulla finestra, vogliamo che il rettangolo si sposti e si posizioni nel punto in cui si trova il mouse. Per prima cosa modifichiamo la classe `RectangleComponent`, aggiungendole un metodo `moveRectangleTo` che sposti il rettangolo in una nuova posizione.

### File RectangleComponent2.java

```

1 import java.awt.Graphics;
2 import java.awt.Graphics2D;
3 import java.awt.Rectangle;
4 import javax.swing.JComponent;
5
6 /**
7     Componente che visualizza un rettangolo che può essere spostato.
8 */
9 public class RectangleComponent2 extends JComponent
10 {
11     private static final int BOX_X = 100;
12     private static final int BOX_Y = 100;
13     private static final int BOX_WIDTH = 20;

```

```
14 private static final int BOX_HEIGHT = 30;
15
16 private Rectangle box;
17
18 public RectangleComponent2()
19 {
20     // il rettangolo che viene disegnato dal metodo paintComponent
21     box = new Rectangle(BOX_X, BOX_Y, BOX_WIDTH, BOX_HEIGHT);
22 }
23
24 public void paintComponent(Graphics g)
25 {
26     Graphics2D g2 = (Graphics2D) g;
27     g2.draw(box);
28 }
29
30 /**
31     Sposta il rettangolo alla posizione specificata.
32     @param x la coordinata x della nuova posizione
33     @param y la coordinata y della nuova posizione
34 */
35 public void moveRectangleTo(int x, int y)
36 {
37     box.setLocation(x, y);
38     repaint();
39 }
40 }
```

Notate l'invocazione di `repaint` nel metodo `moveRectangleTo`: come abbiamo visto nel paragrafo precedente, è necessaria per essere certi che, dopo aver modificato lo stato del rettangolo, il componente ridisegni se stesso, visualizzando il rettangolo nella sua nuova posizione.

A questo punto aggiungiamo al componente un ricevitore di eventi del mouse: ogni volta che viene premuto un pulsante del mouse, il ricevitore sposta il rettangolo nella posizione in cui si trova mouse.

```
class MousePressListener implements MouseListener
{
    public void mousePressed(MouseEvent event)
    {
        int x = event.getX();
        int y = event.getY();
        component.moveRectangleTo(x, y);
    }

    // metodi che non fanno niente
    public void mouseReleased(MouseEvent event) {}
    public void mouseClicked(MouseEvent event) {}
    public void mouseEntered(MouseEvent event) {}
    public void mouseExited(MouseEvent event) {}
}
```

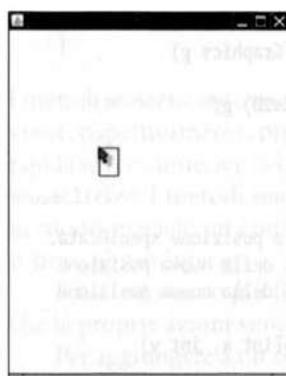
Accade spesso che un particolare ricevitore di eventi specifichi un'azione in corrispondenza di uno o due metodi soltanto, anche se i cinque metodi dell'interfaccia vanno comunque

realizzati tutti: i metodi inutilizzati vengono semplicemente realizzati sotto forma di metodi che non fanno nulla.

Ora procedete ed eseguite il programma RectangleViewer2: ogni volta che fate “click” con il mouse all’interno del frame, l’angolo superiore sinistro del rettangolo si sposta e si posiziona nel punto in cui si trova il puntatore del mouse, come si può vedere nella Figura 9.

**Figura 9**

Un “click” del mouse sposta il rettangolo



### File RectangleFrame2.java

```

1 import java.awt.event.MouseListener;
2 import java.awt.event.MouseEvent;
3 import javax.swing.JFrame;
4
5 /**
6  * Questo frame contiene un rettangolo che si sposta.
7 */
8 public class RectangleFrame2 extends JFrame
9 {
10     private static final int FRAME_WIDTH = 300;
11     private static final int FRAME_HEIGHT = 400;
12
13     private RectangleComponent2 scene;
14
15     class MousePressListener implements MouseListener
16     {
17         public void mousePressed(MouseEvent event)
18         {
19             int x = event.getX();
20             int y = event.getY();
21             scene.moveRectangleTo(x, y);
22         }
23
24         // metodi che non fanno niente
25         public void mouseReleased(MouseEvent event) {}
26         public void mouseClicked(MouseEvent event) {}
27         public void mouseEntered(MouseEvent event) {}
28         public void mouseExited(MouseEvent event) {}

```

```
29 }
30
31 public RectangleFrame2()
32 {
33     scene = new RectangleComponent2();
34     add(scene);
35
36     MouseListener listener = new MousePressListener();
37     scene.addMouseListener(listener);
38
39     setSize(FRAME_WIDTH, FRAME_HEIGHT);
40 }
41 }
```

### File RectangleViewer2.java

```
1 import javax.swing.JFrame;
2
3 /**
4     Visualizza un rettangolo che viene spostato dal mouse.
5 */
6 public class RectangleViewer2
7 {
8     public static void main(String[] args)
9     {
10         JFrame frame = new RectangleFrame2();
11         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12         frame.setVisible(true);
13     }
14 }
```



### Auto-valutazione

35. Perché nella classe RectangleComponent2 il metodo `moveRectangleBy` è stato sostituito dal metodo `moveRectangleTo`?
36. Perché la classe `MousePressListener` deve avere cinque metodi?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R10.19 e E10.29, al termine del capitolo.



### Argomenti avanzati 10.5

#### Eventi della tastiera

Se scrivete il programma per un gioco al calcolatore, probabilmente vorrete elaborare anche gli eventi della tastiera, ad esempio per gestire le frecce direzionali: basta aggiungere al componente che usate per disegnare la scena di gioco un ricevitore di eventi della tastiera (*key listener*). L'interfaccia `KeyListener` ha tre metodi ma, come nel caso degli eventi del mouse, sarete principalmente interessati agli eventi che vengono posti in relazione con la pressione di un tasto (`keyPressed`) e potrete lasciare vuoti gli altri due metodi. Una classe usata come ricevitore di eventi della tastiera potrebbe essere come questa:

```

class MyKeyListener implements KeyListener
{
    public void keyPressed(KeyEvent event)
    {
        String key = KeyStroke.getKeyStrokeForEvent(event).toString();
        key = key.replace("pressed ", "");
        // elabora il tasto premuto
        . . .
    }

    // metodi che non fanno niente
    public void keyReleased(KeyEvent event) {}
    public void keyTyped(KeyEvent event) {}
}

```

L'invocazione `KeyStroke.getKeyStrokeForEvent(event).toString()` trasforma l'oggetto evento in una descrizione testuale del tasto premuto, ad esempio "pressed LEFT" se è stato premuto il tasto della freccia verso sinistra. Nella seconda riga di codice del metodo `keyPressed` abbiamo eliminato il prefisso "pressed ": la parte rimanente è una stringa come "LEFT" oppure "A" e descrive il tasto che è stato premuto. Nella documentazione API della classe `KeyStroke` potete trovare un elenco con i nomi di tutti i tasti.

Come sempre, ricordatevi di associare il ricevitore di eventi alla sorgente corrispondente:

```

KeyListener listener = new MyKeyListener();
scene.addKeyListener(listener);

```

Per poter ricevere eventi della tastiera, un componente deve anche invocare:

```

scene.setFocusable(true);
scene.requestFocus();

```

## Argomenti avanzati 10.6

### Adattatori per eventi

Nel paragrafo precedente avete visto come installare un ricevitore in una sorgente di eventi del mouse e come vengono invocati i metodi di tale ricevitore quando si verifica un evento. Solitamente un programma non è interessato a tutte le segnalazioni potenzialmente destinate a ricevitori: per esempio, un programma può essere interessato soltanto ai "click" del mouse, senza che abbia rilevanza il fatto che questi siano, in realtà, composti da una coppia di eventi di pressione e di rilascio di un pulsante del mouse. Naturalmente il programmatore potrebbe progettare un ricevitore che definisca tutti i metodi a cui non è interessato come metodi "che non fanno nulla", come in questo esempio:

```

class MouseClickListener implements MouseListener
{
    public void mouseClicked(MouseEvent event)
    {
        // reagisce al click del mouse
        . . .
    }
}

```

```
// quattro metodi che non fanno nulla  
public void mouseEntered(MouseEvent event) {}  
public void mouseExited(MouseEvent event) {}  
public void mousePressed(MouseEvent event) {}  
public void mouseReleased(MouseEvent event) {}  
}
```

Per evitarcì tutto questo lavoro, qualche anima buona ha creato la classe `MouseAdapter` che implementa l'interfaccia `MouseListener` in modo che tutti i metodi non facciano nulla. Potete *estendere* tale classe, ereditando i metodi “nullafacenti” e sovrascrivendo soltanto quei metodi che vi interessano, in questo modo:

```
class MouseClickListener extends MouseAdapter  
{  
    public void mouseClicked(MouseEvent event)  
    {  
        // reagisce al click del mouse  
        . . .  
    }  
}
```

Esiste anche la classe `KeyAdapter` che implementa l'interfaccia `KeyListener` con tre metodi che non fanno nulla.



## Computer e società 10.1

### Codice aperto (*open source*) e software libero

La maggior parte delle aziende produttrici di software considerano il codice sorgente alla stregua di un segreto industriale. Dopo tutto, se i clienti o i concorrenti avessero accesso al codice sorgente, potrebbero studiarlo e progettare programmi simili senza ricompensare economicamente il venditore originario. Ai clienti non piace il codice sorgente segreto per gli stessi motivi: se un'azienda fallisce o decide di non fornire più supporto a un determinato programma, i suoi utenti sono lasciati in balia di se stessi e non saranno in grado di correggere gli errori presenti nel software, né di adattarlo a un nuovo sistema operativo. Fortunatamente, però, molti pacchetti software vengono

distribuiti con licenze di tipo “open source” (*codice aperto*, cioè non segreto) che concedono agli utilizzatori il diritto di ispezionare, modificare e ridistribuire il codice sorgente di un programma.

L'accesso al codice sorgente non è sufficiente per garantire che il software possa soddisfare le esigenze dei suoi utilizzatori. Alcune aziende hanno creato software che spia i propri utenti oppure limita l'accesso a contenuti multimediali (libri, musica o video) acquistati in precedenza. Se tale software viene eseguito su un server o su un dispositivo *embedded* (cioè “integrato” in un hardware dedicato), l'utente non ne può modificare il comportamento. In un suo articolo, <http://gnu.org/philosophy/free-software-even-more-important.en.html>, Richard Stallman,

un famoso scienziato informatico e vincitore del premio MacArthur “genius”, descrive il “movimento per il software libero” che sostiene il diritto degli utilizzatori del software di poter controllare il suo comportamento. Si tratta di una posizione etica che va al di là dell'utilizzo di software a codice aperto per i soli motivi legati al risparmio.

Stallman ha fondato il progetto GNU (<http://gnu.org/gnu/the-gnu-project.html>), che ha l'obiettivo di creare una versione interamente aperta e libera di un sistema operativo compatibile con UNIX: il sistema operativo GNU. Tutti i programmi del progetto GNU vengono distribuiti sotto la licenza GNU GPL (General Public License), che consente di copiare il codice, di modificarlo e di distribuire i programmi in forma

originale o modificata, anche facendoli pagare secondo le richieste del mercato, con l'unico vincolo che tali versioni dovranno sottostare comunque alla licenza GPL. In sintesi, bisogna distribuire, insieme ai programmi modificati, anche il relativo codice sorgente e consentire a chiunque di agire nello stesso modo, modificando ulteriormente il codice. La licenza GPL costituisce un contratto sociale: gli utilizzatori del software godono della libertà di utilizzarlo e di modificarlo, ma, in cambio, sono obbligati a condividere i miglioramenti che apportano.

Alcune aziende che vendono software hanno ripetutamente attaccato la licenza GPL, accusandola di essere "virale" e di "minare alle fondamenta il settore del software commerciale", ma altre hanno adottato una strategia più sfumata, producendo software libero oppure open source, facendosi pagare per attività di supporto o per progettare estensioni proprietarie. Ad esempio,

il Java Development Kit è disponibile con GPL, ma aziende che necessitano di aggiornamenti di sicurezza per versioni meno recenti o di altre forme di supporto pagano Oracle.

A volte il software open source non è "pulito" come il software commerciale, perché molti dei programmati coinvolti sono volontari, che sono interessati a risolvere propri problemi e non a progettare un prodotto che sia di facile utilizzo anche per altre persone. Il software open source ha avuto grande successo so-

prattutto in quei settori che sono di interesse per i programmati, come il sistema operativo Linux, i server web e gli strumenti di programmazione.

La comunità di sviluppo di progetti open source può essere molto competitiva e creativa. È piuttosto frequente vedere progetti che competono tra loro, anche copiandosi idee l'un l'altro, diventando così tutti migliori in poco tempo. Il fatto che siano coinvolti molti programmati e che tutti abbiano il codice sorgente completo a disposizione, consente spesso di individuare e correggere gli errori molto rapidamente. Eric Raymond ha descritto lo sviluppo open source in un famoso articolo, "The Cathedral and the Bazaar" (<http://catb.org/~esr/writings/cathedral-bazaar/>), del quale è celebre la frase "Given enough eyeballs, all bugs are shallow" (con un numero sufficiente di occhi, tutti gli errori vengono a galla).



Richard Stallman, un pioniere del movimento per il codice aperto e libero

## Riepilogo degli obiettivi di apprendimento

### Per rendere disponibile un servizio a più classi si usano le interfacce

- In Java un tipo interfaccia dichiara i metodi che possono essere invocati con una variabile di quel tipo.
- Per indicare che una classe implementa un'interfaccia si usa la parola riservata `implements`.
- I tipi interfaccia vengono utilizzati per rendere il codice maggiormente riutilizzabile.

### Come si effettuano conversioni tra classi e interfacce

- Si possono effettuare conversioni dal tipo di una classe al tipo di un'interfaccia che sia implementata dalla classe.
- Le invocazioni di metodi mediante una variabile di tipo interfaccia sono polimorfiche: il metodo da invocare viene determinato durante l'esecuzione del programma.
- Per convertire un riferimento di tipo interfaccia in un riferimento di tipo classe serve un *cast*.

### L'interfaccia Comparable della libreria standard

- Implementate l'interfaccia Comparable in modo che gli oggetti delle vostre classi possano essere confrontati tra loro, ad esempio in un metodo di ordinamento.

### Uso delle interfacce per realizzare *callback*

- Il meccanismo di *callback* consente di specificare codice che verrà eseguito in un secondo momento.

### Con le classi interne si limita l'ambito di visibilità di classi ausiliarie

- Una classe interna viene dichiarata dentro un'altra classe.
- Le classi interne sono solitamente utilizzate per classi con scopo molto limitato, che non hanno bisogno di essere visibili in altre zone del programma.

### Gli oggetti semplificati (*mock*) vengono usati nel collaudo

- Un oggetto *semplificato* ("mock object") fornisce gli stessi servizi di un altro oggetto, ma in modo semplificato.
- La classe semplificata e la classe completa implementano la medesima interfaccia.

### Eventi e loro ricevitori per la programmazione di interfacce grafiche

- Gli eventi dell'interfaccia utente comprendono pressioni di tasti, movimenti del mouse e pressioni di suoi pulsanti, selezioni di voci in menu e così via.
- Un ricevitore di eventi è esemplare di una classe progettata dal programmatore dell'applicazione e i suoi metodi descrivono le azioni da compiere quando si verifica un particolare tipo di evento.
- Le sorgenti di eventi generano segnalazioni relative agli eventi: quando ne avviene uno, lo segnalano a tutti i ricevitori interessati.
- Usate componenti di tipo JButton per realizzare pulsanti grafici e associate un ActionListener a ogni pulsante.
- I metodi di una classe interna possono accedere alle variabili dell'ambito di visibilità circostante.
- Le variabili locali a cui si accede da un metodo di una classe interna non devono essere modificate dopo essere state inizializzate.

### Progettazione di applicazioni grafiche con pulsanti

- Usate un contenitore di tipo JPanel per raggruppare insieme più componenti dell'interfaccia grafica.
- Le azioni conseguenti alla pressione di un pulsante grafico vanno specificate mediante classi che implementano l'interfaccia ActionListener.

### Temporizzatori per realizzare animazioni

- Un temporizzatore genera eventi a intervalli di tempo fissi.
- Il metodo repaint chiede a un componente di ridisegnare se stesso: invocatelo ogni volta che modificate le forme grafiche che vengono disegnate dal metodo paintComponent.

### Programmi che elaborano eventi del mouse

- Per catturare gli eventi del mouse usate un ricevitore di eventi del mouse (*mouse listener*).

## Elementi di libreria presentati in questo capitolo

|                               |                            |
|-------------------------------|----------------------------|
| java.awt.Component            | mouseClicked               |
| addKeyListener                | mouseEntered               |
| addMouseListener              | mouseExited                |
| repaint                       | mousePressed               |
| setFocusable                  | mouseReleased              |
| java.awt.Container            | java.lang.Comparable<T>    |
| add                           | compareTo                  |
| java.awt.Dimension            | double compare             |
| java.awt.Rectangle            | java.lang.Integer          |
| setLocation                   | compare                    |
| java.awt.event.ActionListener | javax.swing.AbstractButton |
| actionPerformed               | addActionListener          |
| java.awt.event.KeyEvent       | javax.swing.JButton        |
| java.awt.event.KeyListener    | javax.swing.JLabel         |
| keyPressed                    | javax.swing.JPanel         |
| keyReleased                   | javax.swing.KeyStroke      |
| keyTyped                      | getKeyStrokeForEvent       |
| java.awt.event.MouseEvent     | javax.swing.Timer          |
| getX                          | start                      |
| getY                          | stop                       |
| java.awt.event.MouseListener  |                            |

## Esercizi di riepilogo e approfondimento

- ★★ **R10.1.** La variabile `a` di tipo `int` contiene il valore due miliardi e alla variabile `b`, anch'essa di tipo `int`, viene assegnato il valore dell'espressione `-a`. Qual è il risultato di `a - b`? E di `b - a`? Qual è il risultato di `Integer.compare(a, b)`? E di `Integer.compare(b, a)`?
- ★★ **R10.2.** La variabile `a` di tipo `double` contiene il valore 0.6 e la variabile `b`, anch'essa di tipo `double`, contiene il valore 0.3. Qual è il risultato di `(int)(a - b)`? E di `(int)(b - a)`? Qual è il risultato di `Double.compare(a, b)`? E di `Double.compare(b, a)`?
- \* **R10.3.** Supponete che `C` sia una classe che implementa le interfacce `I` e `J`. Quali dei seguenti assegnamenti richiede un cast?

```
C c = . . .;
I i = . . .;
J j = . . .;
```

- a. `c = i;`
- b. `j = c;`
- c. `i = j;`

- \* **R10.4.** Supponete che `C` sia una classe che implementa le interfacce `I` e `J` e che `i` sia dichiarata e inizializzata in questo modo:

```
I i = new C();
```

Quali dei seguenti assegnamenti lancia un'eccezione?

- a. `C c = (C) i;`
- b. `J j = (J) i;`
- c. `i = (I) null;`

- \* **R10.5.** Supponete che `Sandwich` sia una classe che implementa l'interfaccia `Edible` e che siano date queste dichiarazioni di variabili:

```
Sandwich sub = new Sandwich();
Rectangle cerealBox = new Rectangle(5, 10, 20, 30);
Edible e = null;
```

Quali dei seguenti assegnamenti sono leciti?

- a. `e = sub;`
- b. `sub = e;`
- c. `sub = (Sandwich) e;`
- d. `sub = (Sandwich) cerealBox;`
- e. `e = cerealBox;`
- f. `e = (Edible) cerealBox;`
- g. `e = (Rectangle) cerealBox;`
- h. `e = (Rectangle) null;`

- \*\* **R10.6.** Le classi `Rectangle2D.Double`, `Ellipse2D.Double` e `Line2D.Double` implementano l'interfaccia `Shape`. La classe `Graphics2D` dipende dall'interfaccia `Shape`, ma non dalle classi che descrivono i rettangoli, le ellissi e i segmenti. Tracciate un diagramma UML che illustri questa situazione.
- \*\* **R10.7.** Supponete che `r` contenga un riferimento a `new Rectangle(5, 10, 20, 30)`. Quali di questi assegnamenti sono validi? Controllate la documentazione API della libreria per verificare quali interfacce siano implementate dalla classe `Rectangle`.

- a. `Rectangle a = r;`
- b. `Shape b = r;`
- c. `String c = r;`
- d. `ActionListener d = r;`
- e. `Measurable e = r;`
- f. `Serializable f = r;`
- g. `Object g = r;`

- \*\* **R10.8.** Classi come `Rectangle2D.Double`, `Ellipse2D.Double` e `Line2D.Double` implementano l'interfaccia `Shape`, la quale ha un metodo

```
Rectangle getBounds()
```

che restituisce il rettangolo minimo che racchiude la forma. Esaminate l'invocazione:

```
Shape s = . . . ;
Rectangle r = s.getBounds();
```

e spiegate perché questo è un esempio di polimorfismo.

- \*\* **R10.9.** Supponete di dover elaborare un array di dipendenti per trovarne il salario medio. Spiegate cosa dovete fare per usare il metodo `Data.average` visto nel Paragrafo 10.1, che elabora oggetti di tipo `Measurable`. Cosa dovete fare per usarne la seconda implementazione, vista nel Paragrafo 10.4? Quale soluzione è più semplice?
- \* **R10.10.** Cosa succede se cercate di usare un array di oggetti `String` con il metodo `Data.average` visto nel Paragrafo 10.1?
- \*\* **R10.11.** Come si può usare il metodo `Data.average` visto nel Paragrafo 10.4 per calcolare la lunghezza media di un insieme di stringhe?

- \*\* **R10.12.** Cosa succede se si fornisce un array di stringhe come argomento al metodo `Data.average` visto nel Paragrafo 10.4?
- \*\* **R10.13.** Considerate queste classi, una di primo livello e una interna. A quali variabili può accedere il metodo `f`?

```

public class T
{
    private int t;

    public void m(final int x, int y)
    {
        int a;
        final int b;

        class C implements I
        {
            public void f()
            {
                ...
            }
        }

        final int c;
        ...
    }
}

```

- \*\* **R10.14.** Cosa succede quando una classe interna cerca di accedere a una variabile locale che assume più valori diversi? Provate e date una spiegazione di ciò che osservate.
- \*\*\* **R10.15 (grafica).** Come riorganizzereste il programma `InvestmentViewer1` se dovreste rendere `AddInterestListener` una classe di primo livello (cioè una classe non interna)?
- \* **R10.16 (grafica).** Cos'è un oggetto che rappresenta un evento? Cos'è una sorgente di evento? Cos'è un ricevitore di eventi?
- \* **R10.17 (grafica).** Dal punto di vista di un programmatore, qual è la differenza più importante fra l'interfaccia utente di un'applicazione per console e quella di un'applicazione grafica?
- \* **R10.18 (grafica).** Spiegate la differenza fra un oggetto `ActionEvent` e un oggetto `MouseEvent`.
- \*\* **R10.19 (grafica).** Spiegate perché l'interfaccia `ActionListener` ha un solo metodo, mentre `MouseListener` ne ha cinque.
- \*\* **R10.20 (grafica).** Una classe può essere un'origine di evento per più tipi di evento? Se sì, fornite un esempio.
- \*\* **R10.21 (grafica).** Quali informazioni contiene un oggetto `ActionEvent`? Quali informazioni aggiuntive sono contenute in un oggetto `MouseEvent`?
- \*\*\* **R10.22 (grafica).** Perché utilizziamo classi interne per i ricevitori di eventi? Se Java non consentisse di usare classi interne, potremmo ugualmente realizzare ricevitori di eventi? In quale modo?
- \*\* **R10.23 (grafica).** Qual è la differenza fra i metodi `paintComponent` e `repaint`.
- \* **R10.24 (grafica).** Qual è la differenza fra un frame e un pannello?

## Esercizi di programmazione

- \*\* **E10.1.** Aggiungete alla classe `Data` un metodo che restituisca l'oggetto avente la misura maggiore, con questa firma:

```
public static Measurable max(Measurable[] objects)
```

- \* **E10.2.** Progettate una classe `Quiz` che implementi l'interfaccia `Measurable`. Un questionario ha un punteggio e un voto in lettere (come `B+`). Usate la classe `Data` dell'Esercizio E10.1 per elaborare un array di questionari, visualizzando il punteggio medio e il questionario con il punteggio più alto, sia in lettere che in numeri.
- \* **E10.3.** Progettate una classe `Person`: una persona è caratterizzata da un nome e un'altezza in centimetri. Usate la classe `Data` dell'Esercizio E10.1 per elaborare un array di oggetti `Person`, visualizzando l'altezza media delle persone e il nome della persona più alta.
- \*\* **E10.4 (per Java 8).** Aggiungete all'interfaccia `Measurable` i metodi statici `largest` e `smallest`, progettati in modo da restituire, rispettivamente, l'oggetto con misura massima e minima in un array di oggetti di tipo `Measurable`.
- \*\*\* **E10.5 (per Java 8).** Aggiungete all'interfaccia `Sequence` vista nella sezione Esempi completi 10.1 i seguenti metodi statici:

```
static Sequence powersOf(int n)
static Sequence multiplesOf(int n)
```

Il primo metodo deve restituire la stessa sequenza prodotta dalla classe `SquareSequence` vista in quella sezione, mentre il secondo ha un comportamento analogo, ma la sequenza prodotta contiene i multipli anziché le potenze.

- \*\* **E10.6 (per Java 8).** Aggiungete all'interfaccia `Sequence` vista nella sezione Esempi completi 10.1 un metodo predefinito che restituisca un array contenente i primi `n` valori della sequenza:

```
default int[] values(int n)
```

- \*\* **E10.7 (per Java 8).** Fate in modo che, nell'interfaccia `Sequence` vista nella sezione Esempi completi 10.1, il metodo `process` diventi un metodo predefinito.
- \*\* **E10.8.** Aggiungete alla classe `Data` un metodo che restituisca l'oggetto avente la misura maggiore, secondo quanto misurato dal misuratore ricevuto come parametro:

```
public static Object max(Object[] objects, Measurer m)
```

- \* **E10.9.** Usando un diverso oggetto `Measurer`, trovate in un insieme di oggetti `Rectangle` quello con il perimetro maggiore.
- \* **E10.10.** Modificate la classe `Coin` vista nel Capitolo 8 in modo che implementi l'interfaccia `Comparable`.
- \* **E10.11.** Risolvete nuovamente l'Esercizio E10.9, facendo in modo che l'oggetto di tipo `Measurer` sia esemplare di una classe interna definita nel metodo `main`.
- \* **E10.12.** Risolvete nuovamente l'Esercizio E10.9, facendo in modo che l'oggetto di tipo `Measurer` sia esemplare di una classe interna definita all'esterno del metodo `main`.
- \*\* **E10.13.** Realizzate una classe `Bag` che rappresenti una borsa per la spesa e memorizzi gli articoli acquistati sotto forma di stringhe. Gli articoli possono essere ripetuti e devono essere presenti

metodi per aggiungere un articolo e per contare quante volte un dato articolo è stato inserito nella borsa:

```
public void add(String itemName)
public int count(String itemName)
```

Gli articoli devono essere memorizzati in una lista di tipo `ArrayList<Item>`, dove `Item` è una classe interna con due variabili di esemplare: il nome dell'articolo e la sua quantità.

- \*\* **E10.14.** Realizzate una classe `Grid` che memorizzi misure all'interno di una griglia rettangolare. La griglia ha un dato numero di righe e di colonne e si può aggiungere una stringa descrittiva a ciascuna sua posizione. Progettate i seguenti metodi e costruttori:

```
public Grid(int numRows, int numColumns)
public void add(int row, int column, String description)
public String getDescription(int row, int column)
public ArrayList<Location> getDescribedLocations()
```

dove `Location` è una classe interna che incapsula il numero di riga e il numero di colonna di una posizione della griglia.

- \*\*\* **E10.15.** Risolvete di nuovo l'esercizio precedente immaginando che la griglia sia illimitata. Il costruttore non ha parametri e le variabili parametro dei metodi `add` e `getDescription`, che sono numeri di riga e di colonna, possono essere numeri interi qualsiasi.
- \*\*\* **E10.16 (grafica).** Scrivete un metodo `randomShape` che generi casualmente oggetti che implementano l'interfaccia `Shape` della libreria standard di Java: un miscuglio di rettangoli, ellissi e segmenti, con posizioni casuali. Invocatelo 10 volte e disegnate tutte le forme.
- \* **E10.17 (grafica).** Migliorate il programma `ButtonViewer` in modo che visualizzi il messaggio “I was clicked *n* times!” ogni volta che viene premuto il pulsante: il valore di *n* deve aumentare a ogni pressione.
- \*\* **E10.18 (grafica).** Migliorate il programma `ButtonViewer` in modo che abbia due pulsanti, ciascuno dei quali visualizzi il messaggio “I was clicked *n* times!” ogni volta che viene premuto. Ogni pulsante deve avere il proprio contatore.
- \*\* **E10.19 (grafica).** Migliorate il programma `ButtonViewer` in modo che abbia due pulsanti etichettati come A e B, ciascuno dei quali, quando premuto, visualizzi il messaggio “Button x was clicked!”, dove x è A o B.
- \*\* **E10.20 (grafica).** Realizzate il programma `ButtonViewer` dell'esercizio precedente usando una sola classe come ricevitore di eventi.
- \* **E10.21 (grafica).** Migliorate il programma `ButtonViewer` in modo che visualizzi l'istante di tempo in cui è avvenuta la pressione del pulsante.
- \*\*\* **E10.22 (grafica).** Realizzate la classe `AddInterestListener` del programma `InvestmentViewer1` in modo che sia una classe normale, non una classe interna. Suggerimento: usate un riferimento al conto bancario, aggiungendo al ricevitore un costruttore che lo inizializzi.
- \*\*\* **E10.23 (grafica).** Realizzate la classe `AddInterestListener` del programma `InvestmentViewer2` in modo che sia una classe normale, non una classe interna. Suggerimento: usate riferimenti al conto bancario e all'etichetta, aggiungendo al ricevitore un costruttore che li inizializzi.
- \*\* **E10.24 (per Java 8).** Realizzate di nuovo il programma del Paragrafo 10.7.2 specificando il ricevitore mediante un'espressione lambda.

- ★★ **E10.25 (per Java 8).** Realizzate di nuovo il programma del Paragrafo 10.8 specificando il ricevitore mediante un'espressione lambda.
- ★★ **E10.26 (per Java 8).** Realizzate di nuovo il programma del Paragrafo 10.9 specificando il ricevitore mediante un'espressione lambda.
- ★★ **E10.27 (grafica).** Scrivete un programma che usi un temporizzatore per visualizzare l'ora esatta una volta ogni secondo. *Suggerimento:* il codice seguente visualizza l'ora esatta e la classe Date si trova nel pacchetto java.util:

```
Date now = new Date();
System.out.println(now);
```

- ★★★ **E10.28 (grafica).** Modificate la classe RectangleComponent del programma di animazione visto nel Paragrafo 10.9 in modo che il rettangolo rimbalzi ai bordi del componente, invece di uscire.
- ★ **E10.29 (grafica).** Modificate la classe RectangleComponent2 del programma di animazione visto nel Paragrafo 10.10 in modo che ogni “click” del mouse provochi l’aggiunta di un nuovo rettangolo al componente visualizzatore. *Suggerimento:* usate un ArrayList<Rectangle> e disegnate tutti i rettangoli all’interno del metodo paintComponent.
- ★ **E10.30.** Progettate una classe Person che implementi l’interfaccia Comparable. Le persone vanno confrontate sulla base del loro nome. Chiedete all’utente di fornire dieci nomi e generate dieci oggetti Person, poi, usando il metodo compareTo, individuate la prima e l’ultima persona e visualizzatene i nomi.

**Sul sito web dedicato al libro si trova una raccolta di progetti di programmazione più complessi.**



# 11

## Ingresso/uscita e gestione delle eccezioni



### Obiettivi del capitolo

---

- Essere in grado di leggere e scrivere file di testo
- Elaborare gli argomenti forniti sulla riga dei comandi
- Imparare a lanciare e a catturare eccezioni
- Saper realizzare programmi che propagano eccezioni a controllo obbligatorio

In questo capitolo imparerete a leggere e scrivere file: un'abilità veramente importante per l'elaborazione di dati reali. Come applicazione di questo, vedrete come cifrare i dati. L'ultima parte del capitolo spiega come i programmi possano segnalare problemi, come la mancanza di file o l'acquisizione di dati non corretti, e riprendere l'elaborazione, usando il meccanismo di gestione delle eccezioni del linguaggio Java.

## 11.1 Leggere e scrivere file di testo

Iniziamo questo capitolo parlando della frequente necessità di leggere e scrivere file contenenti informazioni di tipo testuale (“file di testo”), come i file di codice sorgente Java o i file HTML oppure, ancora, i file che vengono creati con un semplice editor di testi come Windows Notepad.

Per leggere file di testo usate la classe Scanner.

Il modo più semplice per leggere un file di testo prevede l’utilizzo della classe `Scanner`, che avete già visto per la lettura di dati in ingresso provenienti dalla tastiera. Per leggere dati da un file presente sul disco, la classe `Scanner` si affida a un’altra classe, `File`, che descrive file e cartelle presenti in un disco (la classe `File` ha molti metodi di cui non parliamo in questo libro, ad esempio per cancellare un file o per modificarne il nome).

Per prima cosa si costruisce un oggetto di tipo `File`, fornendo il nome del file da leggere:

```
File inputFile = new File("input.txt");
```

Successivamente si utilizza tale oggetto per costruire un oggetto `Scanner`:

```
Scanner in = new Scanner(inputFile);
```

Questo oggetto `Scanner` legge il testo contenuto nel file `input.txt`: per acquisire dati potete usare i consueti metodi della classe, come `next`, `nextLine`, `nextInt` e `nextDouble`.

Ad esempio, il ciclo seguente può essere utilizzato per elaborare numeri presenti in un file di testo:

```
while (in.hasNextDouble())
{
    double value = in.nextDouble();
    . . . // elabora value
}
```

Per scrivere dati in un file si costruisce un oggetto di tipo `PrintWriter`, fornendo il nome del file, come in questo esempio:

```
PrintWriter out = new PrintWriter("output.txt");
```

Se il file in cui scrivere esiste già, viene svuotato prima di scrivervi nuovi dati. Se il file non esiste, viene creato un file vuoto.

La classe `PrintWriter` costituisce un miglioramento della classe `PrintStream`, che già conoscete perché `System.out` ne è un esemplare. Con un oggetto di tipo `PrintWriter` potete usare i consueti metodi `print`, `println` e `printf`:

```
out.println("Hello, World!");
out.printf("Total: %.2f\n", total);
```

Quando avete finito di elaborare un file, chiudetelo.

Quando avete terminato di elaborare un file, accertatevi di *chiudere* l’oggetto `Scanner` o `PrintWriter`:

```
in.close();
out.close();
```

Se il vostro programma termina l'esecuzione senza aver chiuso un oggetto di tipo `PrintWriter`, può darsi che non tutti i dati siano stati realmente scritti nel file sul disco.

Il programma seguente mette all'opera questi concetti: legge un file di testo contenente numeri e li scrive in un altro file, incolonnati e seguiti dalla loro somma.

Se, ad esempio, il file d'ingresso contiene questi dati:

```
32 54 67.5 29 35 80  
115 44.5 100 65
```

allora il programma produce il seguente file:

```
32.00  
54.00  
67.50  
29.00  
35.00  
80.00  
115.00  
44.50  
100.00  
65.00  
Total: 622.00
```

C'è, però, un ulteriore problema con cui confrontarsi. Se il file da cui lo Scanner deve leggere non esiste, nel momento in cui lo Scanner viene costruito si verifica l'eccezione `FileNotFoundException`: il compilatore esige che gli diciamo esplicitamente come vogliamo che il nostro programma reagisca in una tale situazione. Analogamente, il costruttore di `PrintWriter` genera questa eccezione se non è in grado di scrivere nel file (cosa che può succedere quando il nome del file non è valido oppure l'utente non possiede le autorizzazioni necessarie per creare un file nella posizione specificata). Nel nostro semplice programma, se si verifica tale eccezione vogliamo porre termine all'esecuzione del metodo `main`. Per questo è sufficiente aggiungere al metodo `main` una dichiarazione `throws`, in questo modo:

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

Nel paragrafo 11.4 vedremo come gestire in modo più professionale le eccezioni.

Le classi `File`, `PrintWriter` e `FileNotFoundException` sono contenute nel pacchetto `java.io`.

### File Total.java

```
1 import java.io.File;  
2 import java.io.FileNotFoundException;  
3 import java.io.PrintWriter;  
4 import java.util.Scanner;  
5  
6 /**  
7  * Questo programma legge un file contenente numeri e li scrive in  
8  * un altro file, incolonnati e seguiti dal loro totale.  
9 */  
10 public class Total
```

```

11  {
12      public static void main(String[] args) throws FileNotFoundException
13      {
14          // chiede i nomi dei file di input e di output
15
16          Scanner console = new Scanner(System.in);
17          System.out.print("Input file: ");
18          String inputFileNome = console.next();
19          System.out.print("Output file: ");
20          String outputFileNome = console.next();
21
22          // costruisce gli oggetti Scanner e PrintWriter per leggere e scrivere
23
24          File inputFile = new File(inputFileNome);
25          Scanner in = new Scanner(inputFile);
26          PrintWriter out = new PrintWriter(outputFileNome);
27
28          // Acquisisce i dati in ingresso e li scrive in uscita
29
30          double total = 0;
31
32          while (in.hasNextDouble())
33          {
34              double value = in.nextDouble();
35              out.printf("%15.2f\n", value);
36              total = total + value;
37          }
38
39          out.printf("Total: %8.2f\n", total);
40
41          in.close();
42          out.close();
43      }
44  }

```



### Auto-valutazione

- Cosa succede se al programma `Total` viene fornito lo stesso nome per i file di ingresso e di uscita? Se non siete sicuri della risposta, provate.
- Cosa succede se al programma `Total` viene fornito come nome del file d'ingresso il nome di un file inesistente?
- Immaginate di voler aggiungere il totale alla fine del file già esistente, invece di scriverne uno nuovo. La domanda di auto-valutazione numero 1 dice che non è possibile specificare semplicemente lo stesso nome per i file di ingresso e di uscita. Come si può risolvere questo problema? Delineate una soluzione mediante pseudocodice.
- Come modifichereste il programma `Total` per fare in modo che scriva la media dei valori acquisiti, invece della loro somma?
- Come modifichereste il programma `Total` per fare in modo che scriva i valori su due colonne, in questo modo?

|               |       |
|---------------|-------|
| 32.00         | 54.00 |
| 67.50         | 29.00 |
| 35.00         | 80.00 |
| 115.00        | 44.50 |
| 100.00        | 65.00 |
| Total: 622.00 |       |

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R11.1, R11.2 e E11.1, al termine del capitolo.



## Errori comuni 11.1

### Barre rovesciate (*backslash*) nei nomi di file

Quando specificate sotto forma di stringa letterale il nome di un file che contiene caratteri “barra rovesciata” (come nei sistemi operativi Windows), dovete inserire ciascuna barra rovesciata *due volte*:

```
File inputFile = new File("c:\\\\homework\\\\input.dat");
```

Ricordate che una sola barra rovesciata all’interno di stringhe racchiuse da virgolette è un **carattere di escape**, che viene combinato con il carattere seguente per assumere un significato speciale, come per esempio `\n`, che rappresenta l’operazione “andare a capo”. La combinazione `\\` definisce una singola barra rovesciata.

Quando, però, il nome di un file è fornito al programma dall’utente, la barra rovesciata non va digitata due volte.



## Errori comuni 11.2

### Costruire uno Scanner per leggere una stringa

Quando costruite un oggetto `PrintWriter` usando una stringa come parametro di costruzione, poi scriverà in un file:

```
PrintWriter out = new PrintWriter("output.txt");
```

Questo, però, non funziona con un oggetto `Scanner`. L’enunciato

```
Scanner in = new Scanner("input.txt"); // ERRORE ?
```

*non* apre un file, ma, semplicemente, analizza il contenuto della stringa: l’invocazione `in.next()` restituisce la stringa `"input.txt"`. In alcuni casi (come nel Paragrafo 11.2.5) si tratta di una caratteristica utile.

Dovete soltanto ricordarvi di fornire al costruttore di `Scanner` un oggetto `File`:

```
Scanner in = new Scanner(new File("input.txt")); // così va bene
```



## Argomenti avanzati 11.1

### Leggere pagine Web

Con questa sequenza di enunciati potete leggere il contenuto di una pagina web:

```
String address = "http://horstmann.com/index.html";
URL pageLocation = new URL(address);
Scanner in = new Scanner(pageLocation.openStream());
```

Da questo punto in poi potete leggere il contenuto della pagina web usando lo Scanner nel modo consueto. Il costruttore di URL e il metodo openStream possono lanciare un'eccezione di tipo IOException, quindi dovete aggiungere al metodo main la clausola throws IOException (nel Paragrafo 11.4.3 troverete maggiori informazioni sulla clausola throws).

La classe URL si trova nel pacchetto java.net.

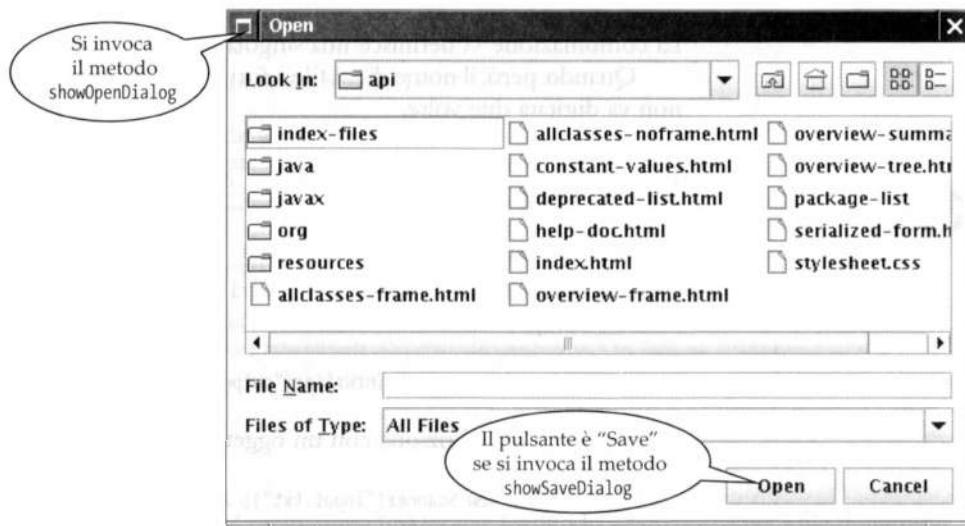


## Argomenti avanzati 11.2

### Finestre per la selezione di file

In un programma dotato di interfaccia grafica, spesso si vuole consentire all'utente di selezionare il nome di un file mediante una finestra di dialogo (*dialog box*), come quella realizzata dalla classe JFileChooser nel pacchetto Swing, dedicato proprio alle interfacce grafiche per l'interazione con l'utente.

Una finestra di dialogo di tipo JFileChooser



La classe JFileChooser ha molte opzioni per mettere a punto la visualizzazione della finestra di dialogo, ma nella sua forma di base l'utilizzo è piuttosto semplice: si costruisce un oggetto per la scelta del file (un esemplare di JFileChooser), quindi si invoca il suo metodo showOpenDialog o showSaveDialog. Entrambi i metodi visualizzano la stessa finestra di dialogo, ma il pulsante per selezionare un file si chiama, rispettivamente, "Open" ("Apri") o "Save" ("Salva").

Per un posizionamento migliore della finestra di dialogo sullo schermo potete specificare il componente dell'interfaccia utente al di sopra del quale volete aprire la finestra di dialogo; se non vi importa dove si apre la finestra di dialogo potete usare semplicemente null. I metodi showOpenDialog e showSaveDialog restituiscono JFileChooser.APPROVE\_OPTION se l'utente ha scelto un file, JFileChooser.CANCEL\_OPTION se l'utente ha annullato la selezione. Se è stato selezionato un file, per ottenere un oggetto File che lo descrive si invoca il metodo getFile. Ecco un esempio completo:

```
JFileChooser chooser = new JFileChooser();
Scanner in = null;
if (chooser.showOpenDialog(null) == JFileChooser.APPROVE_OPTION)
{
    File selectedFile = chooser.getSelectedFile();
    in = new Scanner(selectedFile);
    ...
}
```

## Argomenti avanzati 11.3



### Codifiche dei caratteri

Un **carattere** (*character*, come la lettera A, la cifra 0, la lettera accentata é, la lettera greca π, il simbolo ∫ o uno degli ideogrammi cinesi) viene codificato mediante una sequenza di byte, ciascuno dei quali, rappresentato come numero intero non negativo in base decimale, è un valore compreso tra 0 e 255.

Sfortunatamente non esiste un'unica codifica. Nel 1963, ASCII (American Standard Code for Information Interchange) definì una codifica per 128 caratteri, che comprende tutte le lettere maiuscole e minuscole dell'alfabeto latino, le cifre decimali e alcuni simboli, come + \* %, e li rappresenta come valori compresi tra 0 e 127. Ad esempio, il codice per la lettera A è 65.

Un numero di nazioni sempre maggiore sentì il bisogno di codificare il proprio alfabeto e progettò il proprio codice, spesso basandosi sulla codifica ASCII e usando i valori compresi nell'intervallo che va da 128 a 255 per i simboli caratteristici della propria lingua. Ad esempio, in Spagna la lettera é è stata codificata con il numero 233, ma in Grecia lo stesso codice denota la lettera t (iota minuscola). Come potete ben immaginare, se un turista spagnolo di nome José inviasse un messaggio di posta elettronica a un albergo greco, questa diversa codifica sarebbe un problema.

Per risolvere questi problemi, nel 1987 iniziò il progetto della codifica **Unicode**. Come descritto nella sezione Computer e società 4.2, questa codifica assegna un numero intero (non negativo) univoco a ciascun carattere utilizzato nel mondo, anche se esistono comunque più varianti della codifica binaria di questi numeri interi, la più diffusa delle quali è denominata UTF-8 e codifica ciascun carattere come una sequenza di byte, da uno a quattro. Ad esempio, la lettera A è ancora 65, come in ASCII, mentre la lettera é viene codificata come 195 169. I dettagli dello schema di codifica non sono importanti, ciò che importa è specificare, quando si legge e si scrive un file, che la codifica adottata è UTF-8.

Fino ad oggi i sistemi operativi Windows e Macintosh non hanno ancora adottato la codifica UTF-8 e Java utilizza lo stesso schema di codifica del sistema operativo: se non si specifica qualcosa di diverso, le classi `Scanner` e `PrintWriter` leggono e scrivono file usando tale codifica, una scelta efficace se i file contengono soltanto caratteri ASCII oppure se lettura e scrittura del file avvengono con la stessa codifica. Se, però, dovete elaborare file che contengono caratteri accentati, ideogrammi cinesi o simboli speciali, dovreste richiedere esplicitamente la codifica UTF-8, costruendo così uno `Scanner`:

```
Scanner in = new Scanner(file, "UTF-8");
```

e un `PrintWriter`:

```
PrintWriter out = new PrintWriter(file, "UTF-8");
```

Probabilmente vi starete chiedendo come mai Java non sia in grado di individuare automaticamente la codifica corretta. Per capirlo, prendiamo in esame la stringa José, che, in UTF-8, viene codificata come 74 111 115 195 169. I primi tre byte, che codificano Jos, appartengono all'intervallo della codifica ASCII e non pongono problemi, ma i due byte successivi, 195 169, potrebbero rappresentare la lettera é in UTF-8 oppure la coppia di caratteri Áj nella codifica tradizionale spagnola. L'oggetto Scanner non conosce lo spagnolo e non è in grado di decidere quale sia la codifica da scegliere.

Quindi, quando scambiate file con utenti di altre zone del mondo, dovreste sempre specificare esplicitamente la codifica.

## 11.2 Acquisire e scrivere testi

In questo paragrafo imparerete a elaborare dati di tipo testuale con il livello di complessità tipico dei problemi reali.

### 11.2.1 Acquisire parole

Il metodo next legge una stringa delimitata da caratteri di spaziatura.

Il metodo next della classe Scanner acquisisce la stringa successiva. Considerate questo ciclo:

```
while (in.hasNext())
{
    String input = in.next();
    System.out.println(input);
}
```

Fornendo in ingresso il testo:

```
Mary had a little lamb
```

questo ciclo visualizzerebbe ciascuna parola su una riga a sé stante:

```
Mary
had
a
little
lamb
```

Un testo, però, può contenere anche segni di punteggiatura e altri simboli. Il metodo next restituisce una sequenza di caratteri qualsiasi che non siano **caratteri di spaziatura (white space)**, che sono gli spazi veri e propri (*space*), i caratteri di tabulazione (*tab*) e i caratteri di “nuova riga” che separano le righe (*newline*). Ad esempio, queste sono considerate parole dal metodo next:

```
snow.
1729
C++
```

(osservate che il punto dopo `snow` è considerato parte della parola, perché non è un carattere di spaziatura).

Vediamo in dettaglio cosa accade quando viene invocato il metodo `next`. I caratteri in ingresso che sono caratteri di spaziatura vengono *consumati*, cioè vengono rimossi dal flusso di ingresso, senza entrare a far parte della parola che si sta costruendo. Il primo carattere che risulta essere diverso da un carattere di spaziatura diventa il primo carattere della parola. Si aggiungono ulteriori caratteri finché non si trova un carattere di spaziatura oppure si raggiunge la fine del file. Se, però, la fine del file viene raggiunta prima che sia stato aggiunto almeno un carattere alla parola in costruzione, si verifica `NoSuchElementException`.

A volte si vogliono leggere proprio le parole nel senso comune del termine, ignorando qualsiasi carattere che non sia una lettera. Per farlo bisogna invocare il metodo `useDelimiter` dell'oggetto `Scanner` che si sta usando:

```
Scanner in = new Scanner(. . .);
in.useDelimiter("[^A-Za-z]+");
```

In questo modo abbiamo specificato che lo schema di caratteri (detto *pattern*) che separa le parole è “qualunque sequenza di caratteri diversi dalle lettere” (si veda la sezione Argomenti avanzati 11.4). Con queste impostazioni, i segni di punteggiatura e le cifre numeriche vengono eliminate dalle parole restituite dal metodo `next`.

### 11.2.2 Acquisire caratteri

A volte si vuole leggere un carattere per volta, come vedrete nell'esempio riportato nel Paragrafo 11.3, dove eseguiremo la cifratura dei caratteri di un file. Per ottenere questo risultato si invoca il metodo `useDelimiter` dell'oggetto `Scanner` usando come parametro una stringa vuota:

```
Scanner in = new Scanner(. . .);
in.useDelimiter("");
```

Ogni successiva invocazione di `next` restituisce una stringa contenente un solo carattere, per cui possiamo elaborare i singoli caratteri di un file in questo modo:

```
while (in.hasNext())
{
    char ch = in.next().charAt(0);
    . . . // elabora ch
}
```

### 11.2.3 Classificare caratteri

La classe `Character` contiene metodi utili per classificare i caratteri.

Quando si acquisisce un carattere in ingresso o quando si analizzano i caratteri di una parola o di una riga, spesso si vuole sapere di che tipo di carattere si tratta: la classe `Character` contiene parecchi metodi utili a risolvere questo problema, ciascuno dei quali richiede un parametro di tipo `char` e restituisce un valore di tipo `boolean` (come si può vedere nella Tabella 1).

Ad esempio, l'invocazione

```
Character.isDigit(ch)
```

restituisce `true` se `ch` è una cifra compresa tra '`0`' e '`9`' oppure una cifra in un altro sistema di scrittura, come descritto nella sezione Computer e società 4.2, altrimenti restituisce `false`.

**Tabella 1**

Metodi per classificare caratteri

| Metodo                    | Esempi di caratteri     |
|---------------------------|-------------------------|
| <code>isDigit</code>      | 0, 1, 2                 |
| <code>isLetter</code>     | A, B, C, a, b, c        |
| <code>isUpperCase</code>  | A, B, C                 |
| <code>isLowerCase</code>  | a, b, c                 |
| <code>isWhiteSpace</code> | caratteri di spaziatura |

### 11.2.4 Acquisire righe

Il metodo `nextLine` legge un'intera riga.

Quando ogni riga di un file di testo costituisce un dato unitario (detto *record*), spesso è meglio leggere l'intera riga con il metodo `nextLine`:

```
String line = in.nextLine();
```

La successiva riga di testo (privata del carattere finale di “nuova riga”) viene assegnata alla stringa `line`: si può, così, disporre della riga per le elaborazioni richieste.

Il metodo `hasNextLine` restituisce `true` se c’è almeno un’ulteriore riga di testo da acquisire e `false` quando tutte le righe sono state lette: per essere sicuri che ci sia un’altra riga da elaborare, invocate il metodo `hasNextLine` prima del metodo `nextLine`.

Ecco un esempio tipico di elaborazione di righe di un file. Un file con dati di popolazione, estratto dal CIA Fact Book (<https://www.cia.gov/library/publications/the-world-factbook/>) contiene righe con questo formato:

```
China 1330044605
India 1147995898
United States 303824646
...
```

Dal momento che alcune nazioni hanno nomi composti da più parole, sarebbe scomodo leggere questo file usando il metodo `next`: dopo aver letto, ad esempio, la parola `United`, come farebbe il programma a sapere di dover leggere ancora una parola, prima di leggere il valore che esprime la popolazione?

Leggiamo, invece, ciascuna riga intera, memorizzandola in una stringa:

```
while (in.hasNextLine())
{
    String line = in.nextLine();
    . . . // elaborazione della stringa line
}
```

poi usiamo i metodi `isDigit` e `isWhitespace` della Tabella 1 per scoprire dove finisce il nome della nazione e dove inizia il numero.

Troviamo la prima cifra:

```
int i = 0;
while (!Character.isDigit(line.charAt(i))) { i++; }
```

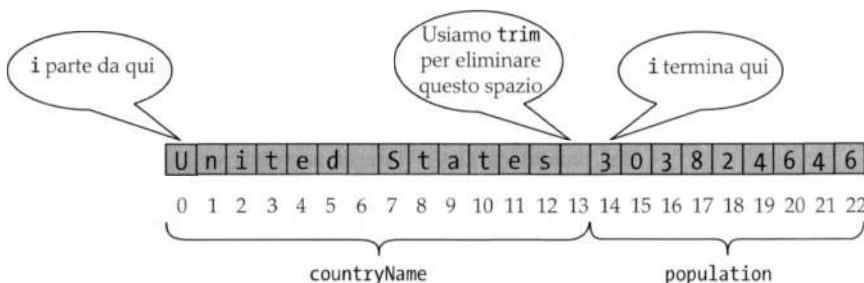
Quindi, estraiamo il nome della nazione e la relativa popolazione:

```
String countryName = line.substring(0, i);
String population = line.substring(i);
```

In questo modo, però, il nome della nazione termina con uno o più spazi, che eliminiamo usando il metodo `trim`:

```
countryName = countryName.trim();
```

Il metodo `trim` restituisce una copia della stringa originaria, eliminando tutti i caratteri di spaziatura iniziali e finali.



Abbiamo ancora un problema: la popolazione è memorizzata in una stringa, non in una variabile numerica. Vedremo nel Paragrafo 11.2.6 come convertire la stringa in un numero.

### 11.2.5 Analizzare una stringa

Nel paragrafo precedente avete visto come scomporre una stringa in sottostringhe cercando singoli caratteri specifici, ma a volte è più semplice seguire un approccio diverso. Per leggere i caratteri presenti in una stringa si può usare un oggetto `Scanner`:

```
Scanner lineScanner = new Scanner(line);
```

Questo `lineScanner` può essere utilizzato come qualsiasi altro `Scanner`, leggendo parole e numeri:

```
String countryName = lineScanner.next(); // leggiamo la prima parola
// aggiungiamo parole a countryName finché non troviamo un numero
while (!lineScanner.hasNextInt())
{
    countryName = countryName + " " + lineScanner.next();
}
int populationValue = lineScanner.nextInt();
```

### 11.2.6 Convertire stringhe in numeri

A volte capita di avere una stringa che contiene un numero, come la stringa `population` nel Paragrafo 11.2.4. Supponiamo, ad esempio, che la stringa sia costituita dalla sequenza di caratteri "303824646". Per ottenere il numero intero 303824646 usiamo il metodo `Integer.parseInt`:

```
int populationValue = Integer.parseInt(population);
// ora il valore di populationValue è il numero intero 303824646
```

Se una stringa contiene le cifre di un numero, se ne può ottenere il valore invocando `Integer.parseInt` oppure `Double.parseDouble`.

Per convertire una stringa contenente le cifre di un numero in virgola mobile nel suo valore si usa il metodo `Double.parseDouble`. Supponiamo, ad esempio, che la stringa `input` sia uguale a "3.95".

```
double price = Double.parseDouble(input);
// ora il valore di price è il numero 3.95
```

Quando si invocano i metodi `Integer.parseInt` oppure `Double.parseDouble` occorre fare attenzione: il loro parametro deve essere una stringa che contiene le cifre di un numero, senza altri caratteri, nemmeno di spaziatura! Nel nostro esempio sappiamo che non ci saranno spazi all'inizio della stringa, ma ce ne potrebbero essere alla fine, per cui è meglio invocare prima il metodo `trim`:

```
int populationValue = Integer.parseInt(population.trim());
```

Questo esempio prosegue nella sezione Consigli pratici 11.1.

### 11.2.7 Evitare errori nell'acquisizione di numeri

Avete già usato molte volte i metodi `nextInt` e `nextDouble` della classe `Scanner`, ma qui ci occuperemo di quello che succede in situazioni "anomale". Analizziamo l'invocazione:

```
int value = in.nextInt();
```

Il metodo `nextInt` riconosce numeri come 3 o -21. Se, però, nei dati letti dallo `Scanner` non c'è un numero, si verifica l'eccezione `InputMismatchException`. Consideriamo, ad esempio, la situazione in cui i dati in ingresso sono questi:

Invocando `nextInt` viene "consumato" lo spazio iniziale e viene letta la parola `21st`, che non rispetta il formato previsto per i numeri: in questa situazione nel metodo `nextInt` si verifica l'eccezione `InputMismatchException`, per "mancata corrispondenza" (*mismatch*) tra il formato previsto e il formato effettivamente letto.

Se, invece, quando si invocano i metodi `nextInt` o `nextDouble` non c'è alcun dato presente, si verifica l'eccezione `NoSuchElementException`. Per evitare eccezioni, per acquisire un numero intero si analizza il flusso di dati in ingresso con il metodo `hasNextInt`, ad esempio in questo modo:

```
if (in.hasNextInt())
{
    int value = in.nextInt();
    ...
}
```

Analogamente, prima di invocare `nextDouble` bisognerebbe invocare `hasNextDouble`.

### 11.2.8 Acquisire numeri, parole e righe

I metodi `nextInt`, `nextDouble` e `next` non consumano i caratteri di spaziatura che seguono il dato acquisito: ciò può rappresentare un problema se si alternano invocazioni di `nextInt`/`nextDouble`/`next` e di `nextLine`. Immaginate, infatti, di avere un file contenente nomi di nazioni e le loro popolazioni, in questo formato:

```
China
1330044605
India
1147995898
United States
303824646
```

e di leggerlo con questo frammento di codice:

```
while (in.hasNextLine())
{
    String countryName = in.nextLine();
    int population = in.nextInt();
    ... // elaborazione di countryName e population
}
```

Inizialmente in ingresso abbiamo:

```
C|h|i|n|a|\n1|3|4|0|0|4|4|6|0|5|\nI|n|d|i|a|\n
```

Dopo la prima invocazione di `nextLine`, rimane in ingresso:

```
1|3|4|0|0|4|4|6|0|5|\nI|n|d|i|a|\n
```

Dopo l'invocazione di `nextInt`, rimane in ingresso:

```
\nI|n|d|i|a|\n
```

Si osservi che l'invocazione di `nextInt` non consuma il carattere *newline*, quindi la seconda invocazione di `nextLine` restituisce una stringa vuota!

La soluzione consiste nell'invocare `nextLine` dopo aver letto il valore della popolazione:

```
String countryName = in.nextLine();
int population = in.nextInt();
in.nextLine(); // consuma il carattere newline
```

L'invocazione di `nextLine` consuma tutti i caratteri di spaziatura residui e il carattere `newline`.

### 11.2.9 Impaginare i dati in uscita

Spesso, quando si visualizzano numeri o stringhe, si vuole fare in modo che la loro impaginazione sia sotto il controllo del programmatore. Ad esempio, le quantità espresse in dollari vanno solitamente visualizzate con due cifre dopo il separatore decimale, in questo modo:

```
Cookies:      3.20
```

Fin dal Paragrafo 4.3.2 sapete come ottenere un'impaginazione di questo tipo, usando il metodo `printf`: in questo paragrafo presenteremo alcune ulteriori opzioni relative a tale metodo.

Supponiamo di voler visualizzare una tabella di articoli e prezzi, memorizzati in due array, in questo modo:

```
Cookies:      3.20
Linguine:      2.95
Clams:        17.29
```

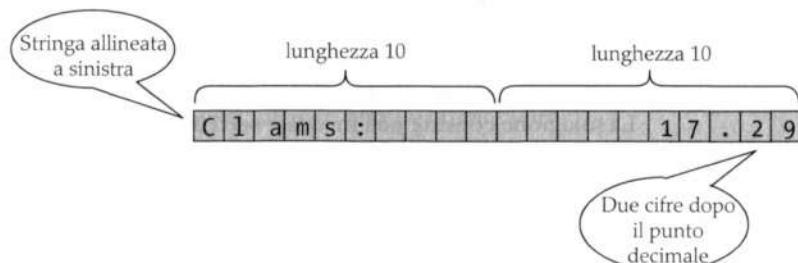
Osservate che le stringhe che rappresentano gli articoli devono essere incolonnate a sinistra, mentre i numeri che rappresentano i prezzi vanno incolonnati a destra. In mancanza di indicazioni, il metodo `printf` incolonna i valori a destra.

Per specificare che il contenuto di un campo deve essere incolonnato a sinistra, si aggiunge un trattino (*hyphen*, -) prima dell'ampiezza del campo stesso:

```
System.out.printf("%-10s%10.2f", items[i] + ":", prices[i]);
```

In questo esempio vediamo due diverse specifiche di formato:

- La specifica `%-10s` impagina una stringa con incolonnamento a sinistra. Alla stringa `prices[i] + ":"` vengono aggiunti spazi in modo che diventi lunga dieci caratteri (una procedura che viene chiamata *padding*). Il simbolo `-` specifica che la stringa deve essere posta a sinistra, seguita da un numero di spazi sufficiente a raggiungere la lunghezza di 10 caratteri.
- La specifica `%10.2f` impagina un numero in virgola mobile, anche questo in un campo largo dieci caratteri. In questo caso, però, gli spazi vengono aggiunti alla sinistra del valore.



Un elemento sintattico come `%-10s` o `%10.2f` è detto *specifica* o specificatore di formato e descrive come debba essere visualizzato un valore.

Una specifica di formato ha la seguente struttura:

- Il primo carattere è `%`.
- Poi, troviamo delle segnalazioni facoltative (dette *flag*) che modificano il formato stesso, come avviene, ad esempio, con il flag `-`, che indica l'incollonamento a sinistra. La Tabella 2 riassume i flag più utilizzati.
- A seguire troviamo l'ampiezza del campo, cioè il numero totale di caratteri che costituisce il campo visualizzato (compresi gli spazi inseriti per il *padding*), seguita da una precisione (facoltativa) per i numeri in virgola mobile.
- La specifica di formato termina con il *tipo di formato*, come `f` per i numeri in virgola mobile e `s` per le stringhe. Questi tipi non sono molti e la Tabella 3 mostra i principali.

**Tabella 2**  
Flag di formato

| Flag           | Significato                                     | Esempio                            |
|----------------|-------------------------------------------------|------------------------------------|
| <code>-</code> | Incollonamento a sinistra                       | <code>1.23</code> seguito da spazi |
| <code>0</code> | Mostra zeri a sinistra                          | <code>001.23</code>                |
| <code>+</code> | Mostra il segno + nei numeri positivi           | <code>+1.23</code>                 |
| <code>(</code> | Racchiude tra parentesi tonde i numeri negativi | <code>(-1.23)</code>               |
| <code>,</code> | Mostra il separatore delle migliaia             | <code>12,300</code>                |
| <code>^</code> | Usa lettere maiuscole                           | <code>1.23E+1</code>               |

**Tabella 3**  
Tipi di formato

| Codice         | Tipo                                                                                            | Esempio              |
|----------------|-------------------------------------------------------------------------------------------------|----------------------|
| <code>d</code> | Numero intero in base decimale                                                                  | <code>123</code>     |
| <code>f</code> | Numero in virgola fissa                                                                         | <code>12.30</code>   |
| <code>e</code> | Numero in notazione esponenziale                                                                | <code>1.23E+1</code> |
| <code>g</code> | Numero generico (viene usata la notazione esponenziale per valori molto grandi o molto piccoli) | <code>12.3</code>    |
| <code>s</code> | Stringa                                                                                         | <code>Tax:</code>    |



## Auto-valutazione

- Se il flusso d'ingresso contiene i caratteri `Hello, World!`, che valore assumono `word` e `input` dopo l'esecuzione di questi enunciati?  
`String word = in.nextLine();`  
`String input = in.nextLine();`
- Se il flusso d'ingresso contiene i caratteri `995.0 Fred`, che valore assumono `number` e `input` dopo l'esecuzione di questi enunciati?  
`int number = 0;`  
`if (in.hasNextInt()) { number = in.nextInt(); }`  
`String input = in.next();`
- Se il flusso d'ingresso contiene i caratteri `6E6 $6,995.00`, che valore assumono `x1` e `x2` dopo l'esecuzione di questi enunciati?

- Want to know more?*
- ```
double x1 = in.nextDouble();
double x2 = in.nextDouble();
```
9. Se un file contiene una sequenza di numeri, alcuni dei quali sono, però, assenti e vengono sostituiti dalla stringa **N/A** (*Not Available*, cioè “non disponibile”), come si possono acquisire i numeri e ignorare i segnaposto dei valori mancanti?
  10. Come si possono eliminare gli spazi dal nome della nazione, nell’esempio visto nel Paragrafo 11.2.4, senza usare il metodo `trim`?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi E11.4, E11.6 e E11.7, al termine del capitolo.



## Argomenti avanzati 11.4

### Espressioni canoniche (*regular expressions*)

Una **espressione canonica** (*regular expression*) descrive uno schema (*pattern*) di caratteri. Per esempio, i numeri hanno un formato semplice, contengono una o più cifre: l’espressione canonica che descrive numeri è `[0-9]+`. La notazione `[0-9]` indica “qualsiasi cifra compresa fra 0 e 9”, mentre il segno `+` significa “uno o più” di tali elementi.

Le funzioni di ricerca degli ambienti di programmazione professionali sono in grado di interpretare le espressioni canoniche e numerosi programmi di servizio usano espressioni canoniche per individuare corrispondenze testuali. Un programma molto diffuso che usa espressioni canoniche è `grep` (il cui nome è l’acronimo di “global regular expression print”, cioè visualizzazione di espressioni canoniche generalizzate). Potete eseguire `grep` da una finestra di comandi o all’interno di alcuni ambienti di compilazione: fa parte del sistema operativo UNIX, ma ne esistono versioni anche per Windows. Il programma richiede un’espressione canonica e il nome di uno o più file in cui cercare: durante la propria esecuzione visualizza un elenco delle righe che corrispondono all’espressione canonica.

Supponiamo di voler cercare tutti i “numeri magici” (introdotti nella sezione Suggerimenti per la programmazione 4.1) presenti in un file. Il comando seguente elenca tutte le righe del file `Homework.java` che contengono sequenze di cifre:

```
grep [0-9]+ Homework.java
```

Il risultato non è particolarmente utile, perché elenca anche le righe che contengono nomi di variabili, come `x1`. È chiaro che, invece, cerchiamo le sequenze di cifre che *non* sono immediatamente successive a una lettera:

```
grep [^A-Za-z][0-9]+ Homework.java
```

La notazione `[^A-Za-z]` indica “qualsiasi carattere che *non* è compreso nell’intervallo fra A e Z, né fra a e z”. Il risultato è molto migliore e mostra soltanto le righe che contengono effettivamente numeri.

Il metodo `useDelimiter` della classe `Scanner` accetta un’espressione canonica per descrivere i delimitatori, cioè i blocchi di testo che separano le “parole”. Come già visto in precedenza, usando `[^A-Za-z]` come schema, il delimitatore risulta definito come una sequenza di uno o più caratteri che non siano lettere.

La classe `String` ha due utili metodi che usano espressioni canoniche. Il metodo `split` suddivide una stringa in un array di stringhe, usando come delimitatori sequenze di caratteri definite da un'espressione canonica. Ad esempio

```
String[] tokens = line.split("\\s+");
```

scomponete `line` usando i caratteri di spaziatura come delimitatori. Il metodo `replaceAll` restituisce una stringa in cui tutte le occorrenze di un'espressione canonica sono state sostituite da una stringa data. Ad esempio, la stringa restituita da `word.replaceAll("[aeiou]", "")` è uguale a `word`, da cui sono state rimosse tutte le vocali.

Per avere maggiori informazioni in merito alle espressioni canoniche consultate uno dei tanti siti Internet che ne parlano, usando ad esempio come chiave di ricerca “regular expression tutorial”.

## Argomenti avanzati 11.5

### Acquisire un intero file

Nel paragrafo precedente avete visto come acquisire righe, parole e caratteri da un file. In alternativa, è possibile leggere l'intero file e memorizzarlo in un vettore di righe (`lines`) o in un'unica stringa (`content`), usando le classi `Files` e `Paths` del pacchetto `java.nio.file`, in questo modo:

```
String filename = . . .;
ArrayList<String> lines = Files.readAllLines(Paths.get(filename));
String content = new String(Files.readAllBytes(Paths.get(filename)));
```

## 11.3 Argomenti sulla riga dei comandi

In relazione al sistema operativo e al sistema di sviluppo di Java utilizzati, esistono metodi diversi per eseguire un programma: selezionando “Run” nell’ambiente di compilazione, attivando un’icona o digitando il nome del programma in una finestra di comandi. Quest’ultimo metodo è detto “invocazione del programma dalla riga dei comandi”: quando usate questo metodo dovete naturalmente digitare il nome del programma, ma potete anche aggiungere altre informazioni che il programma potrebbe utilizzare. Queste stringhe addizionali sono dette **argomenti sulla riga dei comandi** (*command line arguments*). Ad esempio, se mettete in esecuzione un programma in questo modo:

```
java ProgramClass -v input.dat
```

allora il programma `ProgramClass` riceve due argomenti dalla riga di comando: le stringhe `-v` e `"input.dat"`. Cosa fare con queste stringhe è una decisione che compete interamente al programma, anche se di solito le stringhe che iniziano con un trattino vengono interpretate come opzioni.

È bene che i vostri programmi consentano l'utilizzo di argomenti sulla riga di comando oppure è meglio che chiedano tali informazioni all'utente, magari con una interfaccia grafica? Per un utente occasionale un'interfaccia grafica è decisamente migliore, perché lo guida

e rende possibile l'utilizzo dell'applicazione senza bisogno di molte informazioni, ma per un utente consolidato, che usa il programma molto spesso, l'interfaccia a riga di comando ha un grande vantaggio: si può facilmente automatizzare. Dovendo elaborare centinaia di file ogni giorno, potrete trascorrere gran parte del vostro tempo a digitare i loro nomi all'interno di una finestra grafica di dialogo: usando, invece, un *file batch* o uno *shell script* (una caratteristica messa a disposizione dai sistemi operativi) potete mettere in esecuzione automaticamente un programma molte volte, con argomenti diversi sulla riga di comando.

I programmi che vengono eseguiti dalla riga di comando ricevono i relativi argomenti nel metodo main.

Gli argomenti forniti nella riga di comando giungono al programma all'interno del parametro args del metodo main.

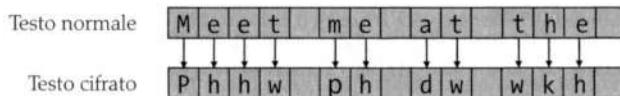
```
public static void main(String[] args)
```

Nel nostro esempio args è un array di lunghezza 2 e contiene queste stringhe:

```
args[0]:  "-v"
args[1]:  "input.dat"
```

Vediamo come scrivere un programma che *cifra* un file, cioè che ne modifica il contenuto in modo che risulti illeggibile a chiunque, tranne a chi conosce il metodo di decifrazione. Ignorando duemila anni di progressi nel campo della cifratura, useremo il metodo di Giulio Cesare, che prevedeva la sostituzione di A con D, B con E, e così via (Figura 1).

**Figura 1**  
Cifratura di Cesare



Il programma richiede i seguenti argomenti sulla riga di comando:

- Un *flag* opzionale, *-d*, per segnalare la richiesta di decifrazione anziché cifratura.
- Il nome del file da leggere.
- Il nome del file da scrivere.

Ad esempio:

```
java CaesarCipher input.txt encrypt.txt
```

cifra il file *input.txt*, memorizzando il risultato nel file *encrypt.txt*, mentre:

```
java CaesarCipher -d encrypt.txt output.txt
```

decifra il file *encrypt.txt*, memorizzando il risultato nel file *output.txt*.

### File CaesarCipher.java

```
1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.io.PrintWriter;
4 import java.util.Scanner;
```

```
5
6 /**
7  Questo programma cifra un file usando il cifrario di Cesare.
8 */
9 public class CaesarCipher
10 {
11     public static void main(String[] args) throws FileNotFoundException
12     {
13         final int DEFAULT_KEY = 3;
14         int key = DEFAULT_KEY;
15         String inFile = "";
16         String outFile = "";
17         int files = 0; // numero di argomenti sulla riga di comando che sono file
18
19         for (int i = 0; i < args.length; i++)
20         {
21             String arg = args[i];
22             if (arg.charAt(0) == '-')
23             {
24                 // è un'opzione sulla riga di comando
25
26                 char option = arg.charAt(1);
27                 if (option == 'd') { key = -key; }
28                 else { usage(); return; }
29             }
30             else
31             {
32                 // è il nome di un file
33
34                 files++;
35                 if (files == 1) { inFile = arg; }
36                 else if (files == 2) { outFile = arg; }
37             }
38         }
39         if (files != 2) { usage(); return; }
40
41         Scanner in = new Scanner(new File(inFile));
42         in.useDelimiter(""); // elabora singoli caratteri
43         PrintWriter out = new PrintWriter(outFile);
44
45         while (in.hasNext())
46         {
47             char from = in.next().charAt(0);
48             char to = encrypt(from, key);
49             out.print(to);
50         }
51         in.close();
52         out.close();
53     }
54
55 /**
56  Cifra lettere maiuscole e minuscole traslandole in
57  base a una chiave.
58  @param ch la lettera da cifrare
59  @param key la chiave di cifratura
60  @return la lettera cifrata
```

```

61  */
62  public static char encrypt(char ch, int key)
63  {
64      int base = 0;
65      if ('A' <= ch && ch <= 'Z') { base = 'A'; }
66      else if ('a' <= ch && ch <= 'z') { base = 'a'; }
67      else { return ch; } // non è una lettera
68      int offset = ch - base + key;
69      final int LETTERS = 26; // numero di lettere nell'alfabeto romano
70      if (offset >= LETTERS) { offset = offset - LETTERS; }
71      else if (offset < 0) { offset = offset + LETTERS; }
72      return (char) (base + offset);
73  }
74
75 /**
76     Visualizza un messaggio che descrive la modalità di utilizzo.
77 */
78 public static void usage()
79 {
80     System.out.println("Usage: java CaesarCipher [-d] infile outfile");
81 }
82 }

```



## Auto-valutazione

11. Se un programma viene invocato con la riga di comando `java CaesarCipher -d file1.txt`, quali sono gli elementi di args?
12. Tenete traccia, passo dopo passo, dell'esecuzione del programma `CaesarCipher`, così come viene invocato nella domanda precedente.
13. Se invocato con la riga di comando `java CaesarCipher file1.txt file2.txt -d`, il programma `CaesarCipher` funziona correttamente? Perché?
14. Cifrate la parola CAESAR usando il cifrario di Cesare.
15. Come si può modificare il programma `CaesarCipher` in modo che l'utente possa specificare una chiave di cifratura diversa da 3 con un'opzione `-k` come in questo esempio?

`java CaesarCipher -k15 input.txt output.txt`

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi R11.5, E11.10 e E11.11, al termine del capitolo.



## Consigli pratici 11.1

### Elaborare file di testo

L'elaborazione di file di testo contenenti dati reali è un problema che può rivelarsi sorprendentemente arduo: in questa sezione troverete una guida in più fasi che vi potrà essere d'aiuto, usando dati relativi alla popolazione delle nazioni del mondo.

**Problema.** Leggere due file di dati nazionali, `worldpop.txt` e `worldarea.txt` (presenti nel pacchetto dei file scaricabili per questo libro): il primo contiene dati sulla popolazione, nazione per nazione, mentre nel secondo si trovano i dati sulla superficie, in chilometri

quadrati; le nazioni sono elencate nello stesso ordine nei due file. Scrivere il file `world_pop_density.txt` che contenga i nomi delle nazioni e le relative densità di popolazione (abitanti per chilometro quadrato), con i nomi delle nazioni incolonnati a sinistra e i numeri incolonnati a destra:

Afghanistan	50.56
Akrotiri	127.64
Albania	125.91
Algeria	14.18
American Samoa	288.92
...	

**Fase 1** Capite il problema.

Come sempre, prima di progettare una soluzione per un problema bisogna averlo compreso chiaramente. Siete in grado di risolvere il problema a mano (eventualmente con file di dati di minori dimensioni)? Se non riuscite a farlo, dovete acquisire ulteriori informazioni in relazione al problema.

Un aspetto importante che dovete considerare è se sia possibile elaborare i dati nel momento in cui questi diventano disponibili oppure se li dovete prima memorizzare tutti. Ad esempio, se vi viene chiesto di visualizzare i dati secondo un determinato criterio di ordinamento, dovete prima acquisirli tutti, probabilmente inserendoli in un vettore. Spesso, invece, i dati possono essere elaborati “al volo” (*on the go*), senza memorizzarli.

Nel nostro esempio possiamo leggere da ciascun file una riga alla volta e calcolare la densità di popolazione per quella nazione, perché i file di dati contengono la popolazione e la superficie nello stesso ordine.

Lo pseudocodice seguente descrive l'elaborazione da compiere.

Finché ci sono altre righe da leggere

Leggi una riga da ciascun file.

Estrai il nome della nazione.

popolazione = numero che segue la nazione nella riga del primo file

superficie = numero che segue la nazione nella riga del secondo file

Se superficie != 0

    densità = popolazione / superficie

    Scrivi nel file il nome della nazione e la densità.

**Fase 2** Individuate i file da leggere e quelli da scrivere.

Questo dovrebbe essere ben chiaro dalla descrizione del problema. Nel nostro esempio ci sono due file da leggere (i dati sulla popolazione e quelli sulla superficie) e un file da scrivere (con le densità).

**Fase 3** Scegliete un meccanismo per acquisire i nomi dei file.

Ci sono tre opzioni:

- Codificare i nomi dei file all'interno del programma (come "worldpop.txt");
- Chiedere all'utente;

```

Scanner in = new Scanner(System.in);
System.out.print("Enter filename: ");
String inFile = in.nextLine();

```

- Usare argomenti sulla riga di comando.

Nel nostro esempio, per semplicità, abbiamo usato la prima opzione.

**Fase 4** Decidete se acquisire i dati per righe, per parole o per singoli caratteri.

Come regola empirica, acquisite righe quando i dati sono raggruppati per righe: questo avviene se, come nel nostro esempio, i dati vengono forniti in formato tabulare. Questa strategia è utile anche quando è necessario fare riferimento al numero di riga.

Quando si acquisiscono dati che sono distribuiti su più righe, è più semplice leggere una parola alla volta. In questo caso occorre ricordare che si perdono i caratteri di spaziatura.

L'acquisizione dei singoli caratteri è utile soprattutto quando il problema richiede, appunto, l'accesso ai singoli caratteri: tra gli esempi, possiamo citare l'analisi della frequenza dei caratteri in un testo, la cifratura o la trasformazione dei caratteri di tabulazione in spazi.

**Fase 5** Quando i dati sono organizzati per righe, estraete i dati richiesti.

Con il metodo `nextLine` è molto semplice leggere una riga di dati in ingresso: poi, bisogna estrarre i dati da ciascuna riga, ad esempio estraendo sottostringhe, come descritto nel Paragrafo 11.2.4.

Solitamente i confini delle sottostringhe verranno individuati usando metodi come `Character.isWhiteSpace` o `Character.isDigit`.

**Fase 6** Usate classi e metodi per realizzare i compiti maggiormente ripetuti.

L'acquisizione di dati da file prevede solitamente alcuni compiti ripetitivi, come il fatto di ignorare i caratteri di spaziatura o di estrarre numeri da stringhe. Trattandosi di operazioni molto noiose, è davvero utile isolarle dal resto del codice.

Nel nostro esempio, notiamo un compito che ricorre due volte: suddividere una riga per ottenere il nome di una nazione e il valore che lo segue. Per risolvere questo problema realizziamo la semplice classe `CountryValue`, usando la tecnica descritta nel Paragrafo 11.2.4..

Ecco, infine, il programma completo.

### File CountryValue.java

```

1 /**
2  * Describe un valore associato a una nazione.
3 */
4 public class CountryValue
5 {
6     private String country;
7     private double value;
8
9 }

```

```
10     Costruisce un CountryValue a partire da una riga.
11     @param line riga che contiene il nome di una nazione, seguito da un valore
12 */
13 public CountryValue(String line)
14 {
15     int i = 0; // trova la prima cifra
16     while (!Character.isDigit(line.charAt(i))) { i++; }
17     int j = i - 1; // trova la fine della parola precedente
18     while (Character.isWhitespace(line.charAt(j))) { j--; }
19     country = line.substring(0, j + 1); // estraе il nome della nazione
20     value = Double.parseDouble(line.substring(i).trim()); // estraе il valore
21 }
22 /**
23     Restituisce il nome della nazione.
24     @return il nome della nazione
25 */
26 public String getCountry() { return country; }
27 /**
28     Restituisce il valore associato.
29     @return il valore associato alla nazione
30 */
31 public double getValue() { return value; }
32 }
33 }
```

### File PopulationDensity.java

```
1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.io.PrintWriter;
4 import java.util.Scanner;
5
6 public class PopulationDensity
7 {
8     public static void main(String[] args) throws FileNotFoundException
9     {
10         // apri i file da leggere
11         Scanner in1 = new Scanner(new File("worldpop.txt"));
12         Scanner in2 = new Scanner(new File("worldarea.txt"));
13
14         // apri il file da scrivere
15         PrintWriter out = new PrintWriter("world_pop_density.txt");
16
17         // acquisisce righe da entrambi i file
18         while (in1.hasNextLine() && in2.hasNextLine())
19         {
20             CountryValue population = new CountryValue(in1.nextLine());
21             CountryValue area = new CountryValue(in2.nextLine());
22
23             // calcola e scrive la densità di popolazione
24             double density = 0;
25             if (area.getValue() != 0) // per evitare la divisione per zero
26             {
27                 density = population.getValue() / area.getValue();
28             }
29         }
30     }
31 }
```

```

29         out.printf("%-40s%15.2f\n", population.getCountry(), density);
30     }
31
32     in1.close();
33     in2.close();
34     out.close();
35   }
36 }
```



## Esempi completi 11.1

### Analizzare i nomi dei bambini

Negli Stati Uniti, la Social Security Administration pubblica, sul proprio sito web, gli elenchi dei nomi più diffusi tra quelli attribuiti ai nuovi nati: <http://www.ssa.gov/OACT/babynames/>. Chiedendo i 1000 nomi più diffusi in un determinato decennio, il browser visualizza il risultato sullo schermo, esattamente come si può vedere nella figura qui riportata.

Rank	Male			Female		
	Name	Number	Percent	Name	Number	Percent
1	Michael	462,065	2.2506	Jessica	302,962	1.5438
2	Christopher	361,250	1.7595	Ashley	301,702	1.5372
3	Matthew	351,477	1.7119	Emily	237,133	1.2082
4	Joshua	328,955	1.8022	Sarah	224,000	1.1413
5	Jacob	298,016	1.4515	Samantha	223,913	1.1408
6	Nicholas	275,222	1.3405	Amanda	190,901	0.9726
7	Andrew	272,600	1.3277	Brittany	190,779	0.9720
8	Daniel	271,734	1.3235	Elizabeth	172,383	0.8783
9	Tyler	262,218	1.2771	Taylor	168,977	0.8609
10	Joseph	260,365	1.2681	Megan	160,312	0.8168
11	Brandon	259,299	1.2629	Hannah	158,647	0.8083
12	David	253,193	1.2332	Kayla	155,844	0.7940

Per memorizzare questi dati in forma di testo, basta semplicemente selezionarli e, con una procedura di “copia e incolla”, inserirli in un file, come il file `babynames.txt` relativo ai nomi utilizzati negli Anni Novanta e reperibile nel pacchetto dei file scaricabili per questo libro.

Ogni riga del file contiene sette dati:

- La posizione (da 1 a 1000)
- Il nome, il numero di utilizzi e la percentuale di utilizzo del nome maschile che occupa tale posizione nella classifica
- Il nome, il numero di utilizzi e la percentuale di utilizzo del nome femminile che occupa tale posizione nella classifica

Per esempio, la riga seguente:

10 Joseph 260365 1.2681 Megan 160312 0.8168

mostra come il decimo nome maschile più diffuso sia stato, in quel decennio, Joseph, usato per una percentuale di bambini pari a 1.2681%, cioè per 260365 bambini, mentre il decimo nome femminile più diffuso è stato Megan. Perché Joseph è più diffuso di Megan? Sembra che i genitori usino un insieme di nomi femminili più ampio, facendo così in modo che ciascuno di essi sia meno frequente.

Il vostro compito è quello di verificare questa supposizione, determinando l'insieme dei nomi attribuiti al 50 per cento dei bambini e al 50 per cento delle bambine più in alto nella lista. Semplicemente, dovete visualizzare i nomi di bambini e bambine, insieme alla loro posizione in classifica, finché non avete raggiunto, separatamente per ciascuno dei due sessi, il 50 per cento di utilizzo complessivo.

#### Fase 1 Capite il problema.

Per elaborare ciascuna riga del file, per prima cosa leggiamo la posizione, quindi leggiamo tre valori per il nome maschile (nome, numero di utilizzi e percentuale), infine ripetiamo la procedura per il nome femminile. Per poter interrompere la procedura quando si raggiunge il 50 per cento, sommiamo le percentuali e ci fermiamo quando, appunto, tale somma raggiunge il 50 per cento.

Ci servono due somme separate per maschi e femmine. Quando una delle due raggiunge il 50 per cento, smettiamo di visualizzare informazioni per quel sesso; quando entrambe raggiungono il 50 per cento, smettiamo di leggere dati.

Lo pseudocodice seguente descrive il processo di elaborazione che abbiamo progettato.

```
totaleMaschile = 0  
totaleFemminile = 0  
Finché totaleMaschile < 50 oppure totaleFemminile < 50  
    Leggi una posizione e visualizzala.  
    Leggi il nome maschile, il suo conteggio e la sua percentuale.  
    Se totaleMaschile < 50  
        Visualizza il nomeMaschile.  
        Aggiungi la percentualeMaschile al totaleMaschile.  
    Ripeti per il sesso femminile.
```

#### Fase 2 Individuate i file da leggere e quelli da scrivere.

C'è un solo file da leggere: `babynames.txt`. Non ci è stato chiesto di scrivere i risultati in un file, quindi visualizzeremo le informazioni usando `System.out`.

**Fase 3** Scegliete un meccanismo per acquisire i nomi dei file.

Non abbiamo bisogno di chiedere all'utente il nome del file.

**Fase 4** Decidete se acquisire i dati per righe, per parole o per singoli caratteri.

I dati in esame non contengono nomi con spazi, come "Mary Jane", quindi ogni riga di dati contiene esattamente sette entità. Dati di questo tipo possono essere acquisiti senza problemi elaborando parole e numeri.

**Fase 5** Quando i dati sono organizzati per righe, estraete i dati richiesti.

Possiamo ignorare questa fase, perché non leggeremo i dati acquisendo una riga alla volta.

Se, però, nella fase precedente avessimo deciso di acquisire i dati una riga alla volta, dovremmo scomporre la riga acquisita in sette stringhe, convertendone cinque in numeri: una procedura alquanto noiosa, che dovrebbe farvi cambiare idea.

**Fase 6** Usate classi e metodi per realizzare i compiti maggiormente ripetuti.

Nello pseudocodice abbiamo scritto *Ripeti per il sesso femminile*, quindi è facilmente individuabile un compito ripetuto, che suggerisce la definizione di un metodo ausiliario, costituito da tre azioni:

Leggi il nome, il suo conteggio e la sua percentuale.

Visualizza il nome se totale < 50.

Aggiungi la percentuale al totale.

Per risolvere questo problema usiamo una classe ausiliaria, *RecordReader*, e ne costruiamo due oggetti, uno per elaborare i nomi maschili e uno per i nomi femminili. Ogni oggetto *RecordReader* gestisce un totale distinto e lo aggiorna aggiungendovi la nuova percentuale, visualizzando i nomi fino al raggiungimento del limite previsto. Il nostro ciclo di elaborazione principale diventa quindi:

```
RecordReader boys = new RecordReader(LIMIT); // maschi
RecordReader girls = new RecordReader(LIMIT); // femmine
while (boys.hasMore() || girls.hasMore())
{
    int rank = in.nextInt();
    System.out.print(rank + " ");
    boys.process(in);
    girls.process(in);
    System.out.println();
}
```

Ecco il codice del metodo *process*:

```
/**
 * Legge una riga e visualizza il nome se il totale è inferiore al limite.
 * @param in il flusso di ingresso
 */
public void process(Scanner in)
{
```

```
String name = in.next();
int count = in.nextInt();
double percent = in.nextDouble();

if (total < limit) { System.out.print(name + " ");
total = total + percent;
}
```

Nel seguito trovate il programma completo.

Date un'occhiata a ciò che viene visualizzato dal programma: soltanto 69 nomi maschili e 153 nomi femminili danno conto di metà di tutte le nascite, davvero una buona notizia per i produttori di accessori personalizzati. L'Esercizio E11.12 vi chiederà di studiare l'evoluzione di questa distribuzione negli anni.

### File BabyNames.java

```
1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.util.Scanner;
4
5 public class BabyNames
6 {
7     public static final double LIMIT = 50;
8
9     public static void main(String[] args) throws FileNotFoundException
10    {
11        try (Scanner in = new Scanner(new File("babynames.txt")))
12        {
13            RecordReader boys = new RecordReader(LIMIT);
14            RecordReader girls = new RecordReader(LIMIT);
15
16            while (boys.hasMore() || girls.hasMore())
17            {
18                int rank = in.nextInt();
19                System.out.print(rank + " ");
20                boys.process(in);
21                girls.process(in);
22                System.out.println();
23            }
24        }
25    }
26 }
```

### File RecordReader.java

```
1 import java.util.Scanner;
2
3 /**
4  * Questa classe elabora dati sui nomi dei bambini.
5  */
6 public class RecordReader
7 {
8     private double total;
9     private double limit;
10 }
```

```

11  /**
12   * Costruisce un RecordReader con totale uguale a zero.
13 */
14 public RecordReader(double aLimit)
15 {
16     total = 0;
17     limit = aLimit;
18 }
19
20 /**
21  Legge una riga e visualizza il nome
22  se il totale è inferiore al limite.
23  @param in il flusso di ingresso
24 */
25 public void process(Scanner in)
26 {
27     String name = in.next();
28     int count = in.nextInt();
29     double percent = in.nextDouble();
30
31     if (total < limit) { System.out.print(name + " ");}
32     total = total + percent;
33 }
34
35 /**
36  Verifica se ci sono altri dati da elaborare.
37  @return true se il limite non è stato raggiunto
38 */
39 public boolean hasMore()
40 {
41     return total < limit;
42 }
43 }

```



## Computer e società 11.1

### Algoritmi di cifratura

Gli esercizi presentati alla fine di questo capitolo riportano alcuni algoritmi per cifrare testi, ma non usateli per mandare messaggi segreti alla persona che amate: qualunque crittografo esperto è in grado in breve tempo di *violare* quegli schemi, cioè di ricostruire il testo originale senza conoscere la chiave segreta.

Nel 1978, Ron Rivest, Adi Shamir e Leonard Adleman presentarono un metodo di cifratura decisamente più potente, chiamato *RSA* (dalle

iniziali dei cognomi dei suoi inventori). Lo schema completo è troppo complicato per poterlo descrivere qui, ma non è poi così difficile da seguire, consultandone i dettagli a questo indirizzo: <http://theory.lcs.mit.edu/~rivest/rsapaper.pdf>.

RSA è un metodo di cifratura veramente interessante. Come si può vedere nella figura, usa due chiavi: una pubblica (*public key*) e una privata (*private key*). La chiave pubblica può essere distribuita a chiunque, ad esempio sul proprio biglietto da visita

o nella firma dei messaggi di posta elettronica. A questo punto, chiunque voglia spedirvi un messaggio che possa essere decifrato soltanto da voi, lo può cifrare con la vostra chiave pubblica. Anche se tutti conoscono la vostra chiave pubblica e anche se qualcuno riuscisse a intercettare il messaggio cifrato diretto a voi, non sarebbe in grado di decifrarlo e di leggerlo. Nel 1994, centinaia di ricercatori, collaborando attraverso le reti Internet, furono in grado di violare un messaggio RSA cifrato con una

chiave a 129 cifre, ma i messaggi odierni, che usano chiavi a 230 cifre o più, sono ritenuti sicuri.

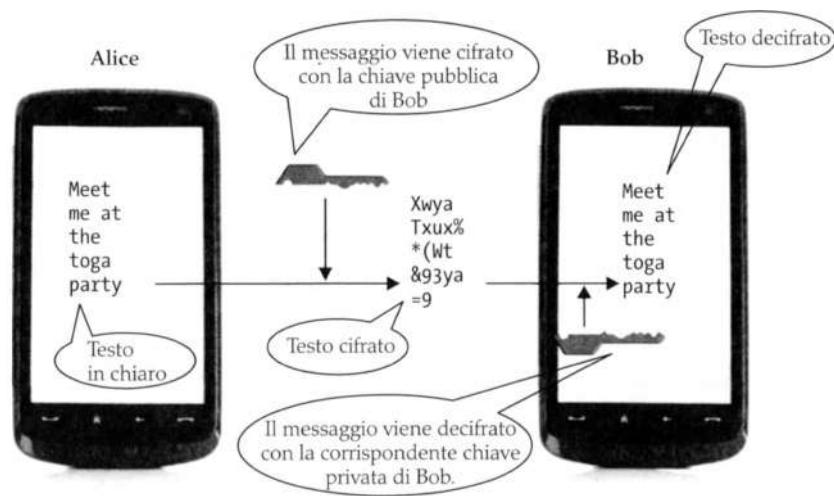
Gli inventori di questo algoritmo ottennero su di esso un *brevetto*, che è un contratto stipulato tra la società e un inventore: per un periodo di 20 anni, l'inventore ha il diritto esclusivo di commercializzare l'invenzione, può incassare i diritti (*royalties*) pagati da altri che vogliono utilizzare l'invenzione e può addirittura impedirne completamente l'utilizzo a eventuali concorrenti. Come contropartita, l'inventore è tenuto a rendere pubblica l'invenzione, in modo che altri la possano studiare, perdendo ogni diritto su di essa al termine del periodo di monopolio. Questo meccanismo si basa sull'ipotesi che, in mancanza di una legge sui brevetti, gli inventori non sarebbero incentivati a sviluppare nuove idee oppure cercherebbero di tenere nascoste le tecniche innovative per impedire ad altri di copiare i propri dispositivi.

In merito al brevetto dell'algoritmo RSA è sorta qualche disputa. In particolare, ci si chiese: in mancanza di una protezione del brevetto, i suoi inventori avrebbero comunque pubblicato il metodo, consegnandone così i benefici alla società senza che questa dovesse in qualche modo pagare il costo di un monopolio ventennale? In questo caso, la risposta è "probabilmente sì", perché gli inventori erano ricercatori accademici, che vivono del proprio stipendio piuttosto che delle vendite di prodotti e, solitamente, sono ricompensati per le proprie

scoperte da un aumento della reputazione e da avanzamenti di carriera. Ma ci sarebbe stata altrettanta attività nella messa a punto di miglioramenti di quell'algoritmo, con ulteriori brevetti, come in effetti c'è stata? Questo non si può sapere, ovviamente. Inoltre, è ragionevole brevettare un algoritmo oppure si tratta di un'evidenza matematica, che non è patrimonio di nessuno? Per lungo tempo l'ufficio brevetti ha seguito questa strada e, infatti, gli inventori di RSA e di molti altri algoritmi hanno descritto le loro invenzioni in termini di un immaginario dispositivo elettronico, piuttosto che di un algoritmo, aggirando così quella restrizione. Oggi, invece, l'ufficio brevetti ammette i brevetti software.

C'è anche un altro aspetto rilevante nella storia di RSA. Un programmatore, Phil Zimmermann, ha sviluppato un programma, chiamato PGP (che sta per *Pretty Good Privacy*, cioè "riservatezza piuttosto efficace")

), che è basato su RSA: chiunque può usare il programma per inviare messaggi cifrati e la loro decifrazione, in mancanza della chiave opportuna, non è praticabile nemmeno con i computer più potenti. Il programma si può ottenere gratuitamente e fa parte del progetto GNU (<http://www.gnupg.org>). L'esistenza di metodi di cifratura molto robusti preoccupa da sempre il governo degli Stati Uniti: i criminali e gli agenti stranieri sono in grado di scambiarsi comunicazioni che la polizia e i servizi segreti non possono decifrare. Il governo ha tentato di incriminare Zimmermann per aver infranto una legge che proibisce l'esportazione non autorizzata di armi, sostenendo che avrebbe dovuto sapere che il suo programma sarebbe stato diffuso tramite Internet. Inoltre è stato proposto di rendere illegale, per i privati cittadini, l'uso di questi metodi di cifratura o, quantomeno, di mantenere segrete le proprie chiavi anche di fronte alla legge.



## 11.4 Gestire eccezioni

Esistono due aspetti relativi agli errori che avvengono durante l'esecuzione di un programma: *individuazione* e *gestione*. Ad esempio, il costruttore di Scanner può *individuare* il tentativo di acquisizione di dati mediante la lettura di un file inesistente, ma non è in grado di *gestire* tale errore: si potrebbe ragionevolmente porre fine all'esecuzione del programma oppure chiedere all'utente di fornire un altro nome di file, e la classe Scanner non è in grado di scegliere tra queste alternative, per cui deve segnalare l'errore a una zona diversa del programma.

In Java, la *gestione delle eccezioni* (*exception handling*) è un meccanismo versatile che consente di trasferire il controllo dell'esecuzione del programma dal punto in cui viene segnalato l'errore a un gestore (*handler*) che sia in grado di gestirlo. La parte restante di questo capitolo discute in dettaglio tale meccanismo.

### 11.4.1 Lanciare eccezioni

Per segnalare una condizione eccezionale si usa l'enunciato `throw` lanciando un oggetto di tipo "eccezione".

```
if (amount > balance)
{
    // cosa facciamo ora?
}
```

Per prima cosa cerchiamo una classe di eccezioni che sia adeguata. La libreria Java mette a disposizione molte classi per segnalare vari tipi di condizioni eccezionali: la Figura 2 mostra quelle più utili (le classi sono organizzate secondo una gerarchia di ereditarietà ad albero, con le classi più specifiche nella zona più bassa).

Cercate un'eccezione che potrebbe descrivere la situazione in esame. Che ne dite di `AritmeticException`? Arrivare ad avere un saldo negativo è un errore aritmetico? Non è così, perché Java è assolutamente in grado di elaborare numeri negativi. È la quantità di denaro da prelevare a non essere valida? Certo, ha un valore troppo elevato, quindi lanciamo un oggetto di tipo `IllegalArgumentException`.

```
if (amount > balance)
{
    throw new IllegalArgumentException("Amount exceeds balance");
```

Quando si lancia un'eccezione, l'elaborazione continua in un gestore dell'eccezione.

Quando **lanciate un'eccezione** (*throw an exception*), l'esecuzione non procede con l'enunciato successivo ma passa a un **gestore dell'eccezione** (*exception handler*), di cui parleremo nel paragrafo successivo.

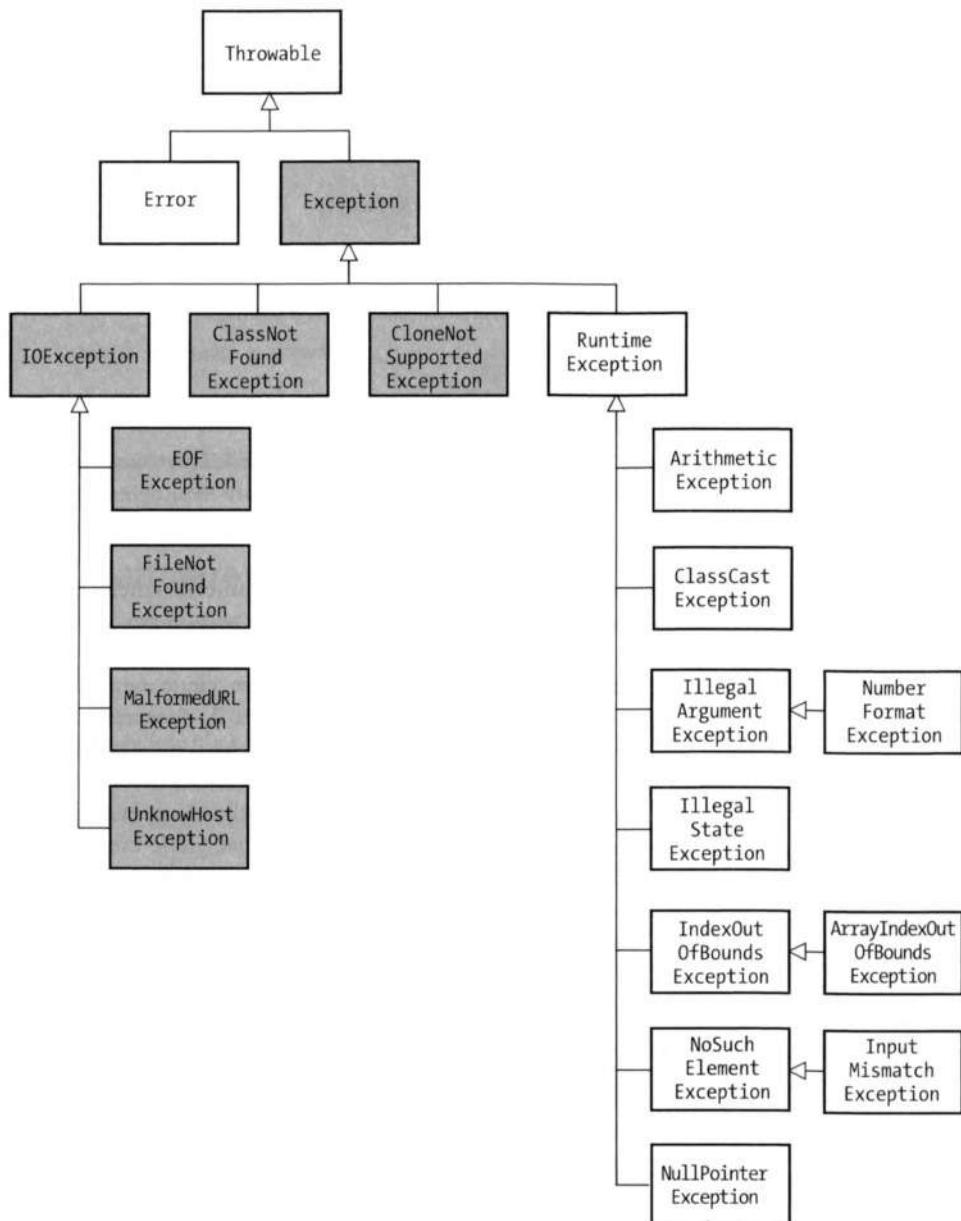
### 11.4.2 Catturare eccezioni

Inserite all'interno di un blocco `try` gli enunciati che possono lanciare un'eccezione, scrivendo poi il gestore in una clausola `catch`.

Tutte le eccezioni dovrebbero essere gestite in qualche punto del vostro programma. Se un'eccezione non ha un gestore e viene lanciata, viene visualizzato un messaggio d'errore e il programma termina, quindi un'eccezione non gestita può certamente confondere l'utente del programma.

**Figura 2**

Una parte della gerarchia delle classi di eccezioni.



Per gestire eccezioni si usa l'enunciato `try/catch`, che va scritto in una zona del programma che sappia come gestire una particolare eccezione. Il blocco `try` contiene enunciati che possono provocare il lancio di un'eccezione del tipo che si vuole gestire ed è seguito da clausole `catch`, ognuna contenente il gestore di uno specifico tipo di eccezione. Ecco un esempio:

```

try
{
    String filename = . . .;
  
```

```

Scanner in = new Scanner(new File(filename));
String input = in.next();
int value = Integer.parseInt(input);
. . .
}
catch (IOException exception)
{
    exception.printStackTrace();
}
catch (NumberFormatException exception)
{
    System.out.println(exception.getMessage());
}

```

In questo blocco try possono essere lanciate eccezioni di tre tipi diversi.

- Il costruttore di Scanner può lanciare `FileNotFoundException`.
- Il metodo `next` di Scanner può lanciare `NoSuchElementException`.
- Il metodo `Integer.parseInt` può lanciare `NumberFormatException`.

Se una di queste eccezioni viene effettivamente lanciata, i rimanenti enunciati del blocco try non vengono eseguiti. Ecco ciò che accade per i diversi tipi di eccezioni:

- Se viene lanciata `FileNotFoundException`, viene eseguita la clausola `catch` corrispondente a `IOException` (guardando la Figura 2, noterete che `FileNotFoundException` è una sottoclasse di `IOException`). Se volete che il messaggio visualizzato all'utente in seguito al verificarsi di una `FileNotFoundException` sia diverso, dovete inserire una clausola `catch` specifica per tale eccezione *prima* della clausola `catch` che gestisce `IOException`.
- Se viene lanciata `NumberFormatException`, viene eseguita la seconda clausola `catch`.
- Un'eccezione di tipo `NoSuchElementException` *non viene catturata* da nessuna delle clausole `catch`, per cui l'eccezione rimane "lanciata" (cioè attiva) finché non viene catturata da un altro blocco try.

Ogni clausola `catch` contiene un gestore di eccezione. Quando viene eseguito il blocco di codice della clausola `catch` (`IOException exception`), significa che uno dei metodi invocati all'interno del blocco try ha lanciato un oggetto di tipo `IOException` (o di una sua sottoclasse).

All'interno del gestore potete visualizzare un elenco della catena di invocazioni di metodi che ha portato all'eccezione, invocando

```
exception.printStackTrace()
```

Nel secondo gestore di eccezione abbiamo invocato `exception.getMessage()` per recuperare il messaggio associato all'eccezione: quando il metodo `parseInt` lancia `NumberFormatException`, tale messaggio contiene la stringa da cui non è stato possibile estrarre un numero intero. Quando lanciate un'eccezione potete anche fornire un vostro messaggio. Ad esempio, in seguito all'esecuzione di questo enunciato

```
throw new IllegalArgumentException("Amount exceeds balance");
```

**Sintassi di Java****11.1 Lanciare un'eccezione****Sintassi**

```
throw oggettoEccezione;
```

**Esempio**

Si costruisce un nuovo oggetto di tipo eccezione, poi lo si lancia.

```
if (amount > balance)
{
    throw new IllegalArgumentException("Amount exceeds balance");
}
balance = balance - amount;
```

La maggior parte degli oggetti di tipo eccezione possono essere costruiti fornendo un messaggio d'errore.

Questo enunciato viene eseguito soltanto se l'eccezione non viene lanciata.

**Sintassi di Java****11.2 Catturare eccezioni****Sintassi**

```
try
{
    enunciato
    enunciato
    ...
}
catch (ClasseEccezione oggettoEccezione)
{
    enunciato
    enunciato
    ...
}
```

**Esempio**

Quando viene lanciata IOException, l'esecuzione prosegue da questo punto.

Qui si possono inserire ulteriori clausole catch. Le eccezioni più specifiche vanno scritte prima di quelle più generiche.

```
try
{
    Scanner in = new Scanner(new File("input.txt"));
    String input = in.next();
    process(input);
}
catch (IOException exception)
{
    System.out.println("Could not open input file");
}
catch (Exception except)
{
    System.out.println(except.getMessage());
}
```

Questo costruttore può lanciare FileNotFoundException.

Questa è l'eccezione che è stata lanciata.

FileNotFoundException è un caso speciale di IOException.

all'eccezione lanciata viene associato il messaggio fornito come argomento nel costruttore.

Nelle clausole `catch` usate in questo esempio ci siamo limitati a informare l'utente della causa del problema, mentre spesso un modo migliore per gestire tali eccezioni consiste nel dare all'utente un'altra possibilità di inserire correttamente i dati, come vedrete nel Paragrafo 11.5.

### 11.4.3 Eccezioni a controllo obbligatorio

In Java le eccezioni ricadono entro tre categorie.

- Errori interni segnalati da sottoclassi di `Error`. Un esempio è `OutOfMemoryError`, un'eccezione che viene lanciata quando la memoria disponibile nel calcolatore è stata esaurita. Si tratta di errori fatali che accadono di rado e in questo libro non ne parleremo.
- Sottoclassi di `RuntimeException`, come `IndexOutOfBoundsException` o `IllegalArgumentException`: segnalano un errore nel codice e sono dette **a controllo non obbligatorio** (*unchecked exception*).
- Tutte le altre eccezioni sono **a controllo obbligatorio** (*checked exception*) e segnalano che qualcosa è andato storto per un motivo esterno al programma, al di fuori del controllo del programmatore. Nella Figura 2 le eccezioni a controllo obbligatorio sono rappresentate da riquadri di colore grigio.

Le eccezioni a controllo obbligatorio sono dovute a circostanze esterne che il programmatore non può evitare e il compilatore verifica che il programma le gestisca.

Perché ci sono due tipi di eccezioni? Un'eccezione a controllo obbligatorio descrive un problema che prima o poi può accadere, indipendentemente da quanto il programmatore sia attento. Per esempio, una `IOException` può essere provocata da fenomeni che non ricadono sotto il controllo del programmatore, come un errore hardware del disco o una connessione di rete interrotta. Il compilatore prende molto sul serio le eccezioni a controllo obbligatorio e si accerta che vengano effettivamente catturate: un programma non verrà compilato se non specifica come gestire le eccezioni a controllo obbligatorio che si possono verificare durante la sua esecuzione.

Le eccezioni a controllo non obbligatorio, al contrario, accadono per un errore del programmatore. Il compilatore non verifica se eccezioni di questa categoria, come `IndexOutOfBoundsException`, vengono gestite: in fin dei conti, invece di scrivere un gestore per una tale eccezione, fareste meglio a controllare i valori degli indici che usate.

La definizione di un gestore di un'eccezione a controllo obbligatorio all'interno dello stesso metodo che può lanciarla soddisfa il compilatore, come in questo esempio:

```
try
{
    File inFile = new File(filename);
    Scanner in = new Scanner(inFile); // può lanciare FileNotFoundException
    . .
}
catch (FileNotFoundException exception) // l'eccezione viene catturata qui
{
    . .
}
```

Aggiungete la clausola `throws` a un metodo che può lanciare un'eccezione a controllo obbligatorio.

Spesso, però, ci si trova in una situazione in cui un metodo *non è in grado* di gestire l'eccezione. In tal caso dovete comunicare al compilatore che siete consapevoli della possibilità che l'eccezione venga lanciata e che volete che l'esecuzione del metodo termini quando questo accade. Per farlo, aggiungete al metodo una clausola `throws`:

## Sintassi di Java

### 11.3 La clausola throws

#### Sintassi

```
modalitàDiAccesso tipoRestituito nomeMetodo(tipoParametro nomeParametro, ...)
throws ClasseEccezione1, ClasseEccezione2, ...
```

#### Esempio

Dovete elencare tutte le eccezioni a controllo obbligatorio che possono essere lanciate da questo metodo.

```
public void read(String filename)
    throws FileNotFoundException, NoSuchElementException
```

Potete elencare anche eccezioni a controllo non obbligatorio.

```
public void readData(String filename) throws FileNotFoundException
{
    File inFile = new File(filename);
    Scanner in = new Scanner(inFile);
    ...
}
```

La clausola `throws` segnala a chi invoca il metodo che, a sua volta, potrà trovarsi di fronte a un'eccezione di tipo `FileNotFoundException`. A questo punto, tale metodo invocante avrà di fronte le medesime alternative: gestire l'eccezione o dichiarare che può essere lanciata dal metodo.

Non gestire un'eccezione quando si sa che può accadere sembra un comportamento in qualche modo irresponsabile, ma, in realtà, è vero il contrario: Java mette a disposizione dei programmatore un meccanismo di gestione delle eccezioni per fare in modo che ciascuna di esse venga inviata a un gestore *adeguato*. Alcuni metodi rilevano un errore, altri metodi lo gestiscono e altri ancora lo lasciano semplicemente passare. La clausola `throws` ha semplicemente lo scopo di garantire che nessuna eccezione si perda per strada.

#### 11.4.4 Chiudere ricorse

Quando si utilizza una risorsa che necessita di un'azione di chiusura, come un oggetto `PrintWriter`, bisogna fare attenzione alla presenza di eccezioni. Analizziamo questa sequenza di enunciati:

```
PrintWriter out = new PrintWriter(filename);
writeData(out);
out.close(); // può darsi che questo enunciato non venga eseguito
```

Supponiamo, ora, che uno dei metodi eseguiti prima dell'ultimo enunciato lanci un'eccezione: in questo caso l'invocazione di `close` non verrà eseguita! È un problema, perché i dati scritti nel flusso `out` potrebbero non arrivare nel file.

La soluzione consiste nell'utilizzo dell'enunciato **try con indicazione di risorse**. Dichiariamo la variabile di tipo `PrintWriter` nell'intestazione di un enunciato `try`, in questo modo:

L'enunciato `try con indicazione di risorse` garantisce che la risorsa specificata venga chiusa sia quando il blocco termina normalmente sia quando viene lanciata un'eccezione.

**Sintassi di Java****11.4 L'enunciato try con indicazione di risorse****Sintassi**

```
try (Tipo1 variabile1 = espressione1; Tipo2 variabile2 = espressione2; . . .)
{
    . . .
}
```

**Esempio**

Questo codice può lanciare eccezioni.

```
try (PrintWriter out = new PrintWriter(filename))
{
    writeData(out);
}
```

A questo punto viene invocato `out.close()`, anche se è stata lanciata un'eccezione.

Implementa l'interfaccia AutoCloseable.

```
try (PrintWriter out = new PrintWriter(filename))
{
    writeData(out);
} // il metodo out.close() viene sempre invocato
```

Quando il blocco `try` termina la propria esecuzione viene invocato il metodo `close` con la variabile dichiarata nell'intestazione. Se non si è verificata alcuna eccezione questo avviene al termine dell'esecuzione del metodo `writeData`; se, invece, è stata lanciata un'eccezione, il metodo `close` viene invocato prima che questa venga trasferita al suo gestore.

In un enunciato `try` con indicazione di risorse si possono dichiarare più variabili, come in questo esempio:

```
try (Scanner in = new Scanner(inFile); PrintWriter out = new PrintWriter(outFile))
{
    while (in.hasNextLine())
    {
        String input = in.nextLine();
        String result = process(input);
        out.println(result);
    }
} // qui vengono invocati in.close() e out.close()
```

Ogni volta che usate un oggetto `Scanner` o `PrintWriter`, inseriteli in un blocco `try` con indicazione di risorse, per essere certi che tali risorse vengano chiuse in modo appropriato.

Più in generale, in tale struttura sintattica possono essere dichiarate variabili di qualsiasi classe che implementi l'interfaccia `AutoCloseable`.

**11.4.5 Progettare eccezioni**

A volte nessuno dei tipi di eccezioni disponibili descrive abbastanza bene la particolare condizione di errore che vi interessa: in tal caso, potete progettare una vostra classe di

eccezione. Considerate un conto bancario: quando si tenta di prelevare una somma superiore al saldo, volete segnalare un'eccezione di tipo `InsufficientFundsException`.

```
if (amount > balance)
{
    throw new InsufficientFundsException(
        "withdrawal of " + amount + " exceeds balance of " + balance);
}
```

Per descrivere una condizione d'errore,  
progettate una sottoclasse  
di una classe di eccezione esistente.

Ora dovete definire la classe `InsufficientFundsException`. Deve essere un'eccezione a controllo obbligatorio oppure no? È causata da qualche evento esterno o da un errore del programmatore? Decidiamo che il programmatore avrebbe dovuto evitare la condizione d'errore: in fin dei conti, non dovrebbe essere difficile verificare che la condizione `amount <= account.getBalance()` sia vera prima di invocare il metodo `withdraw`. Quindi, l'eccezione dovrebbe essere a controllo non obbligatorio ed estendere la classe `RuntimeException` o una delle sue sottoclassi.

Bisogna decidere con cura quale classe estendere, tra quelle già esistenti nella gerarchia: qui, ad esempio, potremmo considerare `InsufficientFundsException` un caso speciale di `IllegalArgumentException`, consentendo ad altri programmatori di catturare quest'ultima eccezione, se non sono interessati all'esatta natura del problema.

Solitamente in una classe che definisce eccezioni si forniscono due costruttori: uno senza argomenti e uno che accetta una stringa come messaggio che descrive la motivazione dell'eccezione. Ecco la dichiarazione della classe:

```
public class InsufficientFundsException extends IllegalArgumentException
{
    public InsufficientFundsException() {}

    public InsufficientFundsException(String message)
    {
        super(message);
    }
}
```

Quando l'eccezione viene catturata, la stringa del messaggio può essere recuperata usando il metodo `getMessage` della classe `Throwable`.



## Auto-valutazione

16. Se il valore di `balance` è 100 e quello di `amount` è 200, che valore ha `balance` dopo l'esecuzione di questo frammento di codice?  

```
if (amount > balance)
{
    throw new IllegalArgumentException("Amount exceeds balance");
}
balance = balance - amount;
```
17. Quando si effettua un versamento in un conto bancario non è possibile che il conto "vada in rosso", cioè che il saldo diventi negativo, a meno che la somma versata non sia negativa. Scrivete un enunciato che, in tal caso, lancia un'eccezione appropriata.
18. Dato il metodo seguente:  

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

```

try
{
    Scanner in = new Scanner(new File("input.txt"));
    int value = in.nextInt();
    System.out.println(value);
}
catch (IOException exception)
{
    System.out.println("Error opening file.");
}
}

```

supponete che il file `input.txt` esista e sia privo di contenuto: seguite passo dopo passo il flusso di esecuzione.

19. Perché `ArrayIndexOutOfBoundsException` è un'eccezione a controllo non obbligatorio?
  20. C'è differenza tra catturare un'eccezione a controllo obbligatorio e catturare un'eccezione a controllo non obbligatorio?
  21. Cosa c'è di sbagliato nel codice seguente e come lo si può correggere?
- ```

public static void writeAll(String[] lines, String filename)
{
    PrintWriter out = new PrintWriter(filename);
    for (String line : lines)
    {
        out.println(line.toUpperCase());
    }
    out.close();
}

```
22. A cosa serve l'invocazione `super(message)` nel secondo costruttore di `InsufficientFundsException`?
  23. Immaginate che il vostro programma acquisisca dati di conti bancari leggendoli da un file. Diversamente da quanto previsto, un valore non è di tipo `double` e decidete di lanciare un'eccezione di tipo `BadDataException`. Quale classe di eccezioni dovreste estendere?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R11.8, R11.9 e R11.10, al termine del capitolo.



## Suggerimenti per la programmazione 11.1

### Lanciare presto, catturare tardi

Quando un metodo individua un problema che non è in grado di risolvere, solitamente è meglio lanciare un'eccezione piuttosto che cercare di mettere in atto una soluzione imprecisa o incompleta. Immaginate, ad esempio, che un metodo preveda di leggere un numero da un file e che il file, invece, non contenga un numero: usare semplicemente il valore zero sarebbe una pessima idea, perché si nasconderebbe il problema reale e, probabilmente, si provocherebbe un diverso problema in un altro punto del programma.

Per contro, un metodo dovrebbe catturare un'eccezione soltanto se è effettivamente in grado di risolvere il problema che è sorto, altrimenti il rimedio migliore consiste nel lasciare che l'eccezione si propaghi al metodo invocante, consentendone la cattura da parte di un gestore competente.

Lanciate un'eccezione non appena si riscontra un problema, ma catturatela soltanto quando il problema può essere gestito.

Questi principi possono essere riassunti in un motto: "lanciare presto, catturare tardi".



## Suggerimenti per la programmazione 11.2

### Non mettete a tacere le eccezioni

Quando invocate un metodo che lancia un'eccezione a controllo obbligatorio e non ne avete definito il gestore, il compilatore protesta. Nell'ansia di portare a termine il vostro lavoro, è comprensibile che zittiate il compilatore *mettendo a tacere* ("squelching") l'eccezione:

```
try
{
    Scanner in = new Scanner(new File(filename));
    // il compilatore protestava per FileNotFoundException
    ...
}
catch (FileNotFoundException e) {} // ecco fatto!
```

Il gestore di eccezione vuoto fa credere al compilatore che l'eccezione sia stata gestita, ma, a lungo termine, questa è chiaramente una cattiva idea. Le eccezioni sono state progettate per trasferire un problema a un gestore competente: l'inserimento di un gestore incompetente nasconde semplicemente una condizione d'errore che potrebbe essere seria.



## Suggerimenti per la programmazione 11.3

### Lanciate eccezioni veramente specifiche

Quando lanciate un'eccezione, dovreste sempre scegliere una classe di eccezioni che descriva nel modo più preciso possibile la situazione d'errore che si è verificata. Ad esempio, quando un conto bancario non ha fondi sufficienti per consentire un prelievo, sarebbe una pessima idea lanciare semplicemente un oggetto `RuntimeException`, perché ciò renderebbe molto più complessa la cattura dell'eccezione. Se, infatti, catturaste tutte le eccezioni di tipo `RuntimeException`, la vostra clausola `catch` verrebbe attivata anche da eccezioni come `NullPointerException`, `ArrayIndexOutOfBoundsException` e così via. Dovreste in tal caso ispezionare attentamente l'oggetto che rappresenta l'eccezione, tentando di capire se questa sia stata provocata da fondi insufficienti oppure no.

Se la libreria standard non contiene un'eccezione che descrive la vostra particolare situazione di errore, definite semplicemente una nuova classe di eccezioni.



## Argomenti avanzati 11.6

### Asserzioni

Un'**asserzione** (*assertion*) è una condizione che il programmatore ritiene debba essere vera in qualsiasi momento in una specifica posizione del programma. Una verifica di asserzione (*assertion check*) controlla che un'asserzione sia effettivamente vera. Ecco un esempio tipico:

```
public void deposit(double amount)
{
```

```

    assert amount >= 0;
    balance = balance + amount;
}

```

In questo metodo il programmatore si aspetta che la quantità di denaro versata non possa mai essere negativa. Quando tale asserzione è verificata non succede nulla e il programma opera nel modo consueto. Se, per qualche ragione, questo non è vero e la verifica di asserzioni durante l'esecuzione è stata abilitata, allora l'enunciato `assert` lancia un'eccezione di tipo `AssertionError`, facendo terminare il programma.

Se, però, la verifica di asserzioni è disabilitata, l'asserzione non viene controllata e il programma funziona alla massima velocità, cosa che avviene per impostazione predefinita.

Per eseguire un programma con la verifica delle asserzioni abilitata, si usa questo comando:

```
java -enableassertions ClassePrincipale
```

Invece dell'opzione `-enableassertions` si può anche usare l'abbreviazione `-ea`. La verifica delle asserzioni dovrebbe essere abilitata durante le fasi di sviluppo e collaudo del programma.



## Argomenti avanzati 11.7

### L'enunciato `try/finally`

Nel Paragrafo 11.4 avete visto come garantire che una risorsa venga chiusa anche quando si verifica un'eccezione. L'enunciato `try` con indicazione di risorse invoca il metodo `close` con tutte le variabili dichiarate nella propria intestazione; per chiudere risorse dovreste sempre usare quella tecnica.

A volte, però, può succedere che si abbia bisogno di eseguire alcune azioni di "pulizia" che non siano l'invocazione del metodo `close`. In tal caso, usate l'enunciato `try/finally`:

```

try
{
    ...
}
finally
{
    ... // questo codice viene eseguito in ogni caso
}

```

Se il corpo dell'enunciato `try` viene eseguito senza che si verifichi un'eccezione, verrà poi eseguito il blocco `finally`. Se, invece, viene lanciata un'eccezione, il blocco `finally` viene eseguito prima che questa venga propagata al proprio gestore.

L'utilizzo dell'enunciato `try/finally` è comunque abbastanza raro, perché la maggior parte delle classi della libreria di Java che necessitano di azioni di chiusura implementano l'interfaccia `AutoCloseable`.

## 11.5 Applicazione: gestione di errori in ingresso

Questo paragrafo analizza un esempio completo di programma che utilizza la gestione di eccezioni. Il programma chiede all'utente il nome di un file, che deve contenere dati secondo queste specifiche: la prima riga del file contiene il numero totale di valori presenti nel seguito, mentre le righe successive contengono i dati veri e propri, uno per riga. Un tipico file di dati in ingresso potrebbe essere simile a questo:

```
3
1.45
-2.1
0.05
```

Durante la progettazione  
di un programma, chiedetevi  
che tipo di eccezioni si possono  
verificare.

Cosa può andare storto? Ci sono due rischi principali.

- Il file potrebbe non esistere.
- Il file potrebbe contenere dati in un formato errato.

Chi può individuare tali errori? Se il file non esiste, il costruttore di Scanner lancerà un'eccezione. I metodi che elaborano i valori in ingresso devono lanciare un'eccezione quando identificano un errore nel formato dei dati.

Quali eccezioni possono essere lanciate? Quando il file non esiste, il costruttore di Scanner lancia un'eccezione `FileNotFoundException`, che è perfetta per tale situazione. Quando il file di dati ha un formato non corretto, lanceremo un'eccezione a controllo obbligatorio progettata da noi, `BadDataException`. Usiamo un'eccezione a controllo obbligatorio perché la corruzione dei dati in un file sfugge al controllo del programmatore.

Per ciascuna eccezione dovete decidere  
quale zona del programma la può  
gestire in modo competente.

Chi può porre rimedio agli errori segnalati da tali eccezioni? Il metodo `main` del programma `DataAnalyzer` è l'unico a interagire con l'utente, per cui cattura qualsiasi eccezione, visualizza un messaggio d'errore appropriato e concede all'utente un'altra possibilità per fornire un file corretto.

### File DataAnalyzer.java

```
1 import java.io.FileNotFoundException;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 /**
6     Questo programma legge un file contenente numeri e ne analizza
7     il contenuto. Se il file non esiste o contiene stringhe che non siano
8     numeri, visualizza un messaggio d'errore.
9 */
10 public class DataAnalyzer
11 {
12     public static void main(String[] args)
13     {
14         Scanner in = new Scanner(System.in);
15         DataSetReader reader = new DataSetReader();
16
17         boolean done = false;
18         while (!done)
```

```

19    {
20        try
21        {
22            System.out.println("Please enter the file name: ");
23            String filename = in.next();
24
25            double[] data = reader.readFile(filename);
26            double sum = 0;
27            for (double d : data) { sum = sum + d; }
28            System.out.println("The sum is " + sum);
29            done = true;
30        }
31        catch (FileNotFoundException exception)
32        {
33            System.out.println("File not found.");
34        }
35        catch (BadDataException exception)
36        {
37            System.out.println("Bad data: " + exception.getMessage());
38        }
39        catch (IOException exception)
40        {
41            exception.printStackTrace();
42        }
43    }
44}
45}

```

Se il file non esiste o se contiene dati in formato errato, le clausole `catch` presenti nel metodo `main` generano una segnalazione d'errore comprensibile per l'utente.

Il seguente metodo `readFile` della classe `DataSetReader` costruisce un oggetto di tipo `Scanner` e invoca il metodo `readData`, disinteressandosi completamente delle eccezioni: se si verifica un problema durante la lettura del file, l'eccezione viene semplicemente trasferita all'invocante.

```

public double[] readFile(String filename) throws IOException
{
    File inFile = new File(filename);
    try (Scanner in = new Scanner(inFile))
    {
        readData(in);
        return data;
    }
}

```

Nella propria clausola `throws`, il metodo dichiara di poter lanciare `IOException`, che è la superclasse comune di `FileNotFoundException` (lanciata dal costruttore di `Scanner`) e di `BadDataException` (lanciata dal metodo `readData`).

Vediamo ora il metodo `readData` della classe `DataSetReader`: legge il numero di valori, costruisce un array e invoca `readValue` per ciascun valore da leggere.

```

private void readData(Scanner in) throws BadDataException
{
    if (!in.hasNextInt())
    {

```

```
        throw new BadDataException("Length expected");
    }
    int numberOfValues = in.nextInt();
    data = new double[numberOfValues];

    for (int i = 0; i < numberOfValues; i++)
    {
        readValue(in, i);
    }

    if (in.hasNext())
    {
        throw new BadDataException("End of file expected");
    }
}
```

Questo metodo controlla due potenziali errori: il file potrebbe non iniziare con un numero intero oppure potrebbe avere ulteriori righe dopo che tutti i valori sono stati letti.

Tuttavia, questo metodo non tenta in alcun modo di catturare eccezioni. Inoltre, se il metodo `readValue` lancia un'eccezione, cosa che avverrà se il file non contiene un numero sufficiente di valori, questa viene semplicemente trasferita a chi ha invocato il metodo.

Ecco il metodo `readValue`:

```
private void readValue(Scanner in, int i) throws BadDataException
{
    if (!in.hasNextDouble())
    {
        throw new BadDataException("Data value expected");
    }
    data[i] = in.nextDouble();
}
```

Per vedere all'opera la gestione delle eccezioni, diamo un'occhiata a uno specifico scenario d'errore.

1. `DataAnalyzer.main` invoca `DataSetReader.readFile`.
2. `readFile` invoca `readData`.
3. `readData` invoca `readValue`.
4. `readValue` non trova il valore atteso e lancia `BadDataException`.
5. `readValue` non ha gestori per tale eccezione e termina immediatamente la propria esecuzione.
6. `readData` non ha gestori per tale eccezione e termina immediatamente la propria esecuzione.
7. `readFile` non ha gestori per tale eccezione e termina immediatamente la propria esecuzione, dopo aver chiuso l'oggetto di tipo `Scanner`.
8. `DataAnalyzer.main` ha un gestore per l'eccezione `BadDataException` che visualizza un messaggio per l'utente, consentendo di nuovo la possibilità di inserire il nome di un file; notate che gli enunciati che calcolano la somma dei valori non sono stati eseguiti.

Questo esempio mostra la separazione tra l'individuazione dell'errore (nel metodo `DataSetReader.readValue`) e la sua gestione (nel metodo `DataAnalyzer.main`). In mezzo si

trovano i metodi `readData` e `readFile`, che semplicemente trasferiscono le eccezioni a chi li ha invocati.

### File DataSetReader.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 /**
6  * Legge un insieme di valori da un file, che deve avere questo formato:
7  * numeroDiValori
8  * valore1
9  * valore2
10 * ...
11 */
12 public class DataSetReader
13 {
14     private double[] data;
15
16     /**
17      * Legge un insieme di dati.
18      * @param filename il nome del file che contiene i dati
19      * @return i dati presenti nel file
20     */
21     public double[] readFile(String filename) throws IOException
22     {
23         File inFile = new File(filename);
24         try (Scanner in = new Scanner(inFile))
25         {
26             readData(in);
27             return data;
28         }
29     }
30
31     /**
32      * Legge tutti i valori.
33      * @param in lo Scanner con cui si acquisiscono i valori
34     */
35     private void readData(Scanner in) throws BadDataException
36     {
37         if (!in.hasNextInt())
38         {
39             throw new BadDataException("Length expected");
40         }
41         int numberOfValues = in.nextInt();
42         data = new double[numberOfValues];
43
44         for (int i = 0; i < numberOfValues; i++)
45         {
46             readValue(in, i);
47         }
48     }
```

```

49     if (in.hasNext())
50     {
51         throw new BadDataException("End of file expected");
52     }
53 }
54 /**
55 Legge un valore.
56 @param in lo Scanner con cui si acquisiscono i valori
57 @param i la posizione in cui memorizzare il valore
58 */
59 private void readValue(Scanner in, int i) throws BadDataException
60 {
61     if (!in.hasNextDouble())
62     {
63         throw new BadDataException("Data value expected");
64     }
65     data[i] = in.nextDouble();
66 }
67 }
68 }
```

### File BadDataException.java

```

1 import java.io.IOException;
2 /**
3  * Questa classe segnala un errore nei dati in ingresso.
4  */
5 public class BadDataException extends IOException
6 {
7     public BadDataException() {}
8     public BadDataException(String message)
9     {
10         super(message);
11     }
12 }
```



### Auto-valutazione

24. Perché il metodo `DataSetReader.readFile` non cattura alcuna eccezione?
25. Seguite passo dopo passo il flusso di esecuzione nel caso in cui l'utente specifichi un file che esiste ma è vuoto.
26. Come si modificherebbe l'implementazione se, quando non è possibile acquisire un numero in virgola mobile, il metodo `readValue` lanciasse un'eccezione `NoSuchElementException` invece di `BadDataException`?
27. Cosa succede all'oggetto `Scanner` se il metodo `readData` lancia un'eccezione?
28. Cosa succede all'oggetto `Scanner` se il metodo `readData` non lancia un'eccezione?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R11.16, R11.17 e E11.13, al termine del capitolo.



## Computer e società 11.2

### L'incidente del razzo Ariane

L'Agenzia Spaziale Europea (European Space Agency, ESA), la controparte europea della NASA, sviluppò un modello di missile denominato Ariane, usato più volte con successo per lanciare satelliti e per svolgere esperimenti scientifici nello spazio. Tuttavia, quando una nuova versione, Ariane 5, fu lanciata il 4 giugno 1996 dal sito di lancio dell'ESA a Kourou, nella Guyana Francese, il missile virò dalla sua rotta circa 40 secondi dopo la partenza. Volando a un angolo di più di 20 gradi, anziché verticalmente, si esercitò su di esso una tale forza aerodinamica che i razzi propulsori si staccarono dal missile, innescando il meccanismo automatico di auto-distrusione: il missile si fece esplodere.

La causa che innescò questo incidente fu un'eccezione non gestita! Il missile conteneva due dispositivi identici (chiamati sistemi di riferimento inerziali) che elaboravano dati di volo provenienti da dispositivi di misura e li trasformavano in informazioni riguardanti la posizione del missile, usate poi dal computer di bordo per controllare i razzi propul-

sori. Gli stessi sistemi di riferimento inerziali e lo stesso software per il calcolatore avevano funzionato bene per il predecessore, Ariane 4.

Tuttavia, a causa di modifiche al progetto del missile, uno dei sensori misurò una forza di accelerazione maggiore di quella che si riscontrava nell'Ariane 4. Tale valore, espresso come numero in virgola mobile, era memorizzato in un numero intero a 16 bit (come una variabile di tipo `short` in Java). Diversamente da Java, il linguaggio Ada usato per il software di quei dispositivi genera un'eccezione se un numero in virgola mobile troppo grande viene convertito in un intero, ma, sfortunatamente, i programmati avevano deciso che tale situazione non sarebbe mai accaduta e non avevano definito un gestore per l'eccezione.

Quando avvenne il trabocco numerico, venne lanciata l'eccezione e, poiché non vi era un gestore, il dispositivo si spense. Il computer di bordo rilevò il guasto e interrogò il sensore di riserva, che, però, si era spento per lo stesso identico motivo, una cosa che i progettisti del missile non avevano previsto: avevano immaginato

che tali dispositivi potessero guastarsi per motivi meccanici e la probabilità che i due dispositivi avessero lo stesso guasto meccanico era considerata assai remota. A quel punto il razzo era privo di informazioni affidabili sulla propria posizione e andò fuori rotta.

Sarebbe forse stato meglio che il software non fosse stato così diligente? Se avesse ignorato l'errore numerico di trabocco, il dispositivo non si sarebbe spento, avrebbe semplicemente elaborato dati errati. Ma in tal caso il sensore avrebbe segnalato errati valori di posizione, cosa che sarebbe stata altrettanto fatale. Al contrario, una implementazione corretta avrebbe dovuto catturare l'errore di trabocco, fornendo una strategia per ricalcolare i dati di volo. Ovviamente, in questo contesto "lasciar perdere" non era una scelta ragionevole.

Il vantaggio del meccanismo di gestione delle eccezioni sta nel fatto che rende questi problemi esplicativi per i programmati, una cosa a cui dovete pensare quando state maledicendo il compilatore Java perché si lamenta di eccezioni non catturate.

## Riepilogo degli obiettivi di apprendimento

### Leggere e scrivere testo memorizzato in file

- Per leggere file di testo usate la classe `Scanner`.
- Per scrivere file di testo usate la classe `PrintWriter` e i suoi metodi `print`/`println`/`printf`.
- Quando avete finito di elaborare un file, chiudetelo.

### Elaborare dati in file di testo

- Il metodo `next` legge una stringa delimitata da caratteri di spaziatura.
- La classe `Character` contiene metodi utili per classificare i caratteri.
- Il metodo `nextLine` legge un'intera riga.

- Se una stringa contiene le cifre di un numero, se ne può ottenere il valore invocando `Integer.parseInt` oppure `Double.parseDouble`.

### Elaborare gli argomenti presenti sulla riga dei comandi

- I programmi che vengono eseguiti dalla riga di comando ricevono i relativi argomenti nel metodo `main`.

### Trasferire il controllo dalla generazione di un errore alla sua gestione

- Per segnalare una condizione eccezionale si usa l'enunciato `throw`, lanciando un oggetto di tipo "eccezione".
- Quando si lancia un'eccezione, l'elaborazione continua in un gestore dell'eccezione.
- Inserite all'interno di un blocco `try` gli enunciati che possono lanciare un'eccezione, scrivendo poi il gestore in una clausola `catch`.
- Le eccezioni a controllo obbligatorio sono dovute a circostanze esterne che il programmatore non può evitare e il compilatore verifica che il programma le gestisca.
- Aggiungete la clausola `throws` a un metodo che può lanciare un'eccezione a controllo obbligatorio.
- L'enunciato `try` con indicazione di risorse garantisce che la risorsa specificata venga chiusa sia quando il blocco termina normalmente sia quando viene lanciata un'eccezione.
- Per descrivere una condizione d'errore, progettate una sottoclasse di una classe di eccezione esistente.
- Lanciate un'eccezione non appena si riscontra un problema, ma catturatela soltanto quando il problema può essere gestito.

### Usare la gestione delle eccezioni in un programma che acquisisce dati

- Durante la progettazione di un programma, chiedetevi che tipo di eccezioni si possono verificare.
- Per ciascuna eccezione dovete decidere quale zona del programma la può gestire in modo competente.

## Elementi di libreria presentati in questo capitolo

|                                                 |                                               |
|-------------------------------------------------|-----------------------------------------------|
| <code>java.io.File</code>                       | <code>java.lang.RuntimeException</code>       |
| <code>java.io.FileNotFoundException</code>      | <code>java.lang.String</code>                 |
| <code>java.io.IOException</code>                | <code>replaceAll</code>                       |
| <code>java.io.PrintWriter</code>                | <code>split</code>                            |
| <code>close</code>                              | <code>java.lang.Throwable</code>              |
| <code>java.lang.AutoCloseable</code>            | <code>getMessage</code>                       |
| <code>java.lang.Character</code>                | <code>printStackTrace</code>                  |
| <code>isDigit</code>                            | <code>java.net.URL</code>                     |
| <code>isLetter</code>                           | <code>openStream</code>                       |
| <code>isLowerCase</code>                        | <code>java.util.InputMismatchException</code> |
| <code>isUpperCase</code>                        | <code>java.util.NoSuchElementException</code> |
| <code>isWhiteSpace</code>                       | <code>java.util.Scanner</code>                |
| <code>java.lang.Double</code>                   | <code>close</code>                            |
| <code>parseDouble</code>                        | <code>hasNextLine</code>                      |
| <code>java.lang.Error</code>                    | <code>nextLine</code>                         |
| <code>java.lang.IllegalArgumentException</code> | <code>useDelimiter</code>                     |
| <code>java.lang.Integer</code>                  | <code>javax.swing.JFileChooser</code>         |
| <code>parseInt</code>                           | <code>getSelectedFile</code>                  |
| <code>java.lang.NullPointerException</code>     | <code>showOpenDialog</code>                   |
| <code>java.lang.NumberFormatException</code>    | <code>showSaveDialog</code>                   |

## Esercizi di riepilogo e approfondimento

- ★★ **R11.1.** Cosa succede se cercate di aprire in modalità di lettura un file inesistente? Che cosa succede se cercate di aprire in modalità di scrittura un file inesistente?
- ★★ **R11.2.** Cosa succede se cercate di aprire un file per scrivere, ma il file o il dispositivo sono protetti contro la scrittura (cioè sono risorse talvolta definite “a sola lettura”, *read-only*)? Fate un esperimento con un breve programma di prova.
- \* **R11.3.** Cosa succede se scrivete dati in un `PrintWriter` senza chiuderlo? Progettate un programma di prova e dimostrate come si possano perdere dati.
- \* **R11.4.** Come si apre un file il cui nome contiene una barra rovesciata, come `c:\temp\output.dat`?
- \* **R11.5.** Se il programma `Woozle` viene eseguito con il comando

```
java Woozle -Dname=piglet -I\eeeyore -v heff.txt a.txt lump.txt
```

quali sono i valori di `args[0]`, `args[1]` e così via?
- \* **R11.6.** Qual è la differenza tra lanciare e catturare un’eccezione?
- \* **R11.7.** Cos’è un’eccezione a controllo obbligatorio? Cos’è un’eccezione a controllo non obbligatorio? Fate un esempio di entrambe le categorie. Quali eccezioni siete obbligati a dichiarare con una clausola `throws`?
- ★★ **R11.8.** Perché non è obbligatorio dichiarare che un metodo può lanciare un’eccezione `IndexOutOfBoundsException`?
- ★★ **R11.9.** Quando un programma esegue un enunciato `throw`, qual è il successivo enunciato che viene eseguito?
- ★★ **R11.10.** Cosa succede se non esiste la clausola `catch` corrispondente a un’eccezione che viene lanciata?
- ★★ **R11.11.** Cosa può fare un programma con l’oggetto eccezione ricevuto da una clausola `catch`?
- ★★ **R11.12.** Il tipo dell’oggetto eccezione è sempre uguale al tipo dichiarato nella clausola `catch` che lo cattura? In caso di risposta negativa, perché?
- \* **R11.13.** A cosa serve l’enunciato `try` con indicazione di risorse? Fornite un esempio di utilizzo.
- ★★ **R11.14.** Cosa succede quando in un enunciato `try` con indicazione di risorse viene lanciata un’eccezione, quindi viene invocato il metodo `close` e questo, a sua volta, lancia un’eccezione diversa dalla prima? Quale eccezione viene catturata da una clausola `catch` circostante? Scrivete un programma d’esempio e provate.
- ★★ **R11.15.** Quali eccezioni possono essere lanciate dai metodi `next` e `nextInt` della classe `Scanner`? Sono eccezioni a controllo obbligatorio oppure no?
- ★★ **R11.16.** Supponete che il programma visto nel Paragrafo 11.5 legga un file contenente questi valori:

1  
2  
3  
4

Quale sarà il risultato? Come si può migliorare il programma per fare in modo che fornisca una più accurata segnalazione degli errori?

- \*\* **R11.17.** Il metodo `readFile` visto nel Paragrafo 11.5 può lanciare un'eccezione di tipo `NullPointerException`? Se sì, in quale situazione?
- \*\*\* **R11.18.** Il codice seguente cerca di chiudere il flusso in cui scrive senza usare un enunciato `try` con indicazione di risorse:

```
PrintWriter out = new PrintWriter(filename);
try
{
    . . . // scrittura dei dati attraverso out
    out.close();
}
catch (IOException exception)
{
    out.close();
}
```

Identificate lo svantaggio di questo approccio. Suggerimento: cosa succede se il costruttore di `PrintWriter` o il metodo `close` lancia un'eccezione?

## Esercizi di programmazione

---

- \* **E11.1.** Scrivete un programma porti a termine i seguenti compiti:
  - Apri un file di nome `hello.txt`.
  - Memorizza nel file il messaggio "Hello, World!".
  - Chiudi il file.
  - Apri di nuovo lo stesso file.
  - Leggi il messaggio memorizzandolo in una stringa e visualizzalo.
- \* **E11.2.** Scrivete un programma che legga un file, elimini le righe vuote e scriva le righe non vuote nel file originario, sovrascrivendolo.
- \*\* **E11.3.** Scrivete un programma che legga un file, elimini le righe vuote che si trovano all'inizio e alla fine e scriva le righe rimanenti nel file originario, sovrascrivendolo.
- \* **E11.4.** Scrivete un programma che legga un file, una riga per volta, scrivendola nel file di uscita preceduta dal *numero di riga*. Se il file letto è:

```
Mary had a little lamb
Whose fleece was white as snow.
And everywhere that Mary went,
The lamb was sure to go!
```

il programma deve scrivere il file:

```
/* 1 */ Mary had a little lamb
/* 2 */ Whose fleece was white as snow.
/* 3 */ And everywhere that Mary went,
/* 4 */ The lamb was sure to go!
```

I numeri di riga devono essere racchiusi tra i delimitatori /\* e \*/, in modo che il programma possa essere utilizzato per numerare le righe di un file sorgente Java.

Il programma deve chiedere all'utente i nomi dei file di ingresso e di uscita.

- ★ **E11.5.** Risolvete di nuovo l'esercizio precedente consentendo all'utente di specificare il nome dei file sulla riga dei comandi. Se l'utente non lo fa, il programma deve chiedere il nome del file durante l'esecuzione.
- ★ **E11.6.** Scrivete un programma che legga un file contenente due colonne di numeri in virgola mobile, per poi visualizzare il valore medio di ciascuna colonna. Chiedete all'utente di fornire il nome del file.
- ★★ **E11.7.** Scrivete un programma che chieda all'utente il nome di un file e, poi, ne visualizzi il numero di caratteri, di parole e di righe.
- ★★ **E11.8.** Scrivete un programma `Find` che effettui una ricerca in tutti i file specificati sulla riga dei comandi e visualizzi tutte le righe contenenti una parola specifica, fornita come primo argomento sulla riga dei comandi. Per esempio, se eseguite

```
java Find ring report.txt address.txt Homework.java
```

il programma potrebbe stampare

```
report.txt: has broken up an international ring of DVD bootleggers that
address.txt: Kris Kringle, North Pole
address.txt: Homer Simpson, Springfield
Homework.java: String filename;
```

- ★★ **E11.9.** Scrivete un programma che controlli l'ortografia di tutte le parole presenti in un file. Deve leggere ciascuna parola del file e controllare se è presente in un elenco di parole, disponibile nei sistemi Macintosh e Linux nel file `/usr/share/dict/words` (e facilmente reperibile anche in Internet). Il programma deve visualizzare tutte le parole del file che non sono presenti nell'elenco.
- ★★ **E11.10.** Scrivete un programma che sostituisca in un file ciascuna riga con la sua inversa. Per esempio, se eseguite

```
java Reverse HelloPrinter.java
```

il contenuto di `HelloPrinter.java` diventa

```
retnirPolleH ssalc cilbup
{
)sgra ]|[gnirts(niam diov citats cilbup
{
;"!dlrow ,olleH"(nltnirp.tuo.metsyS
}
}
```

Naturalmente eseguendo due volte `Reverse` sullo stesso file si ottiene di nuovo il file originale.

- ★★ **E11.11.** Scrivete un programma che legga un file, una riga per volta, e ne scriva le righe in un altro file in ordine inverso. Se, ad esempio, il file `input.txt` ha questo contenuto:

```
Mary had a little lamb
Whose fleece was white as snow
And everywhere that Mary went
The lamb was sure to go.
```

e il programma viene eseguito in questo modo:

```
java ReverseFile input.txt output.txt
```

al termine della sua esecuzione il file `output.txt` dovrà avere questo contenuto:

```
The lamb was sure to go.  
And everywhere that Mary went  
Whose fleece was white as snow  
Mary had a little lamb
```

- ★★ **E11.12.** Recuperate i dati relativi ai nomi usati nei decenni scorsi dal sito della Social Security Administration, inserendoli in file di nome `babynames80s.txt` e così via. Modificate il programma `BabyNames.java` visto nella sezione Esempi completi 11.1 in modo che chieda all'utente il nome di un file. I numeri all'interno dei file sono separati da virgole, quindi dovete modificare il programma in modo che gestisca questo formato. Siete in grado di individuare una tendenza, osservando le frequenze?
- ★★ **E11.13.** Scrivete un programma che chieda all'utente di inserire un insieme di valori in virgola mobile. Quando viene inserito un valore che non è un numero, date all'utente una seconda possibilità di introdurre un valore corretto e, dopo due tentativi, fate terminare il programma. Sommate tutti i valori che sono stati specificati in modo corretto e, quando l'utente ha terminato di inserire dati, visualizzate il totale. Usate la gestione delle eccezioni per identificare i valori di ingresso non validi.
- \* **E11.14.** Modificate la classe `BankAccount` in modo che lanci `IllegalArgumentException` quando viene costruito un conto con saldo negativo, quando viene versata una quantità di denaro negativa o quando viene prelevata una somma che non sia compresa tra 0 e il saldo del conto. Scrivete un programma di collaudo che provochi il lancio di tutte e tre le eccezioni, catturandole.
- ★★ **E11.15.** Ripetete l'esercizio precedente, ma lanciate eccezioni di tre tipi definiti da voi.
- ★★ **E11.16.** Modificate la classe `DataSetReader` in modo che non invochi `hasNextInt` né `hasNextDouble`. Semplicemente, lasciate che `nextInt` e `nextDouble` lancino un'eccezione di tipo `NoSuchElementException`, catturandola nel metodo `main`.

Sul sito web dedicato al libro si trova una raccolta di progetti di programmazione più complessi.



# 12

## Ricorsione



### Obiettivi del capitolo

---

- Imparare a “pensare ricorsivamente”
- Essere in grado di usare metodi ausiliari ricorsivi
- Capire la relazione esistente tra ricorsione e iterazione
- Capire come l’uso della ricorsione si ripercuote sull’efficienza di un algoritmo
- Analizzare problemi che sono molto più semplici da risolvere con la ricorsione che con l’iterazione
- Elaborare dati aventi una struttura ricorsiva usando la ricorsione mutua

La ricorsione è una potente tecnica per scomporre complessi problemi computazionali in problemi più semplici e spesso anche di minore dimensione. Il termine “ricorsione” o “ricorrenza” si riferisce al fatto che la medesima elaborazione “ricorre”, cioè accade ripetutamente, mentre il problema viene risolto. La ricorsione è spesso il modo più naturale di pensare a un problema e alcune elaborazioni sono molto difficili da portare a termine senza la ricorsione. Questo capitolo vi mostra esempi semplici e complessi di ricorsione e vi insegna a “pensare ricorsivamente”.

## 12.1 Numeri triangolari

Iniziamo questo capitolo con un esempio estremamente semplice che mette ben in evidenza la potenza del pensiero ricorsivo. In questo esempio considereremo forme triangolari come queste:

```
[]
[][]
[[][]]
```

Vorremmo calcolare l'area di un triangolo avente ampiezza  $n$ , nell'ipotesi che ciascun quadrato `[]` abbia area unitaria: questo valore viene a volte chiamato *numero triangolare  $n$ -esimo* e, osservando l'esempio, siamo in grado di affermare che il terzo numero triangolare è 6.

Può darsi che sappiate che esiste una formula molto semplice per calcolare questi numeri, ma per il momento dovete supporre di non conoscerla. L'obiettivo finale di questo paragrafo non è quello di calcolare numeri triangolari, ma di comprendere il concetto di *ricorsione* in una situazione semplice.

Ecco una traccia della classe che svilupperemo:

```
public class Triangle
{
    private int width;

    public Triangle(int aWidth)
    {
        width = aWidth;
    }

    public int getArea()
    {
        . . .
    }
}
```

Se l'ampiezza del triangolo è 1, allora il triangolo consiste di un unico quadrato e la sua area vale 1. Occupiamoci prima di questo caso.

```
public int getArea()
{
    if (width == 1) { return 1; }
    . . .
}
```

Per trattare il caso generale, consideriamo questa figura.

```
[]
[][]
[[][]]
[[][][]]
```

Supponiamo di conoscere l'area, `smallerArea`, del triangolo più piccolo, formato dalle prime tre righe: il terzo numero triangolare. Potremo quindi calcolare facilmente l'area

del triangolo più grande, formato da tutte le quattro righe, in questo modo:

```
smallerArea + width
```

Come possiamo calcolare l'area del triangolo più piccolo? Costruiamo un triangolo più piccolo e chiediamogliela!

```
Triangle smallerTriangle = new Triangle(width - 1);
int smallerArea = smallerTriangle.getArea();
```

A questo punto siamo in grado di completare il metodo `getArea`:

```
public int getArea()
{
    if (width == 1) { return 1; }
    Triangle smallerTriangle = new Triangle(width - 1);
    int smallerArea = smallerTriangle.getArea();
    return smallerArea + width;
}
```

Un'elaborazione ricorsiva risolve un problema usando la soluzione del problema stesso nel caso di dati di ingresso più semplici.

Ecco una schematizzazione di ciò che accade quando calcoliamo l'area del triangolo di ampiezza 4.

- Il metodo `getArea` crea un triangolo più piccolo di ampiezza 3.
- Invoca `getArea` su tale triangolo.
  - Quel metodo crea un triangolo più piccolo di ampiezza 2.
  - Invoca `getArea` su tale triangolo.
    - Quel metodo crea un triangolo più piccolo di ampiezza 1.
    - Invoca `getArea` su tale triangolo.
      - Tale metodo restituisce 1.
      - Il metodo restituisce `smallerArea + width = 1 + 2 = 3`.
      - Il metodo restituisce `smallerArea + width = 3 + 3 = 6`.
  - Il metodo restituisce `smallerArea + width = 6 + 4 = 10`.

Questa soluzione presenta un aspetto interessante: per risolvere il problema del calcolo dell'area di un triangolo di ampiezza assegnata, usiamo il fatto che possiamo risolvere lo stesso problema per un triangolo di ampiezza minore. Questa viene chiamata soluzione *ricorsiva*.

Lo schema delle invocazioni di un **metodo ricorsivo** sembra complicato: in effetti, la chiave per il successo nella progettazione di un metodo ricorsivo è *non pensare alla ricorsione*. Invece, osservate ancora una volta il metodo `getArea` e notate come sia tremendamente ragionevole. Se l'ampiezza è 1, ovviamente l'area è 1, e la parte successiva del metodo è altrettanto ragionevole: calcola l'area del triangolo più piccolo, *senza pensare a come questo possa essere fatto*, e aggiunge l'ampiezza, ottenendo chiaramente l'area del triangolo più grande.

Esistono due requisiti che sono basilari per il corretto funzionamento di una ricorsione:

- Ogni invocazione ricorsiva deve semplificare in qualche modo l'elaborazione.
- Devono esistere casi speciali che gestiscano in modo diretto le elaborazioni più semplici.

Perché una ricorsione termini, devono esistere casi speciali per i dati in ingresso più semplici.

Il metodo `getArea` invoca se stesso con valori di ampiezza sempre più piccoli: prima o poi l'ampiezza diventa uguale a 1, che è un caso speciale per il calcolo dell'area di un triangolo, che vale 1. Il metodo `getArea`, quindi, riesce sempre a concludere la propria elaborazione.

In realtà occorre fare molta attenzione: cosa succede se chiedete l'area di un triangolo di ampiezza  $-1$ ? Viene calcolata l'area di un triangolo di ampiezza  $-2$ , la quale operazione richiede il calcolo dell'area di un triangolo di ampiezza  $-3$ , e così via. Per evitare ciò, il metodo `getArea` dovrebbe restituire 0 se l'ampiezza non è maggiore di zero.

La ricorsione non è assolutamente necessaria per calcolare numeri triangolari. L'area di un triangolo è uguale a questa somma:

$$1 + 2 + 3 + \dots + \text{width}$$

e, ovviamente, possiamo calcolarla con un semplice ciclo:

```
double area = 0;
for (int i = 1; i <= width; i++)
{
    area = area + i;
}
```

Molte ricorsioni semplici possono essere calcolate con cicli, ma per molte ricorsioni complesse, come quella del nostro prossimo esempio, i cicli equivalenti possono essere complessi.

In realtà, in questo caso non avete nemmeno bisogno di un ciclo per calcolare la risposta. La somma dei primi  $n$  numeri interi si può calcolare con questa formula:

$$1 + 2 + \dots + n = n \times (n + 1)/2$$

Quindi, l'area vale

$$\text{width} * (\text{width} + 1) / 2$$

In sostanza, per risolvere questo problema non erano necessari né la ricorsione, né un ciclo. La soluzione ricorsiva è da intendersi come una fase di “riscaldamento” per il paragrafo successivo.

### File Triangle.java

```
1 /**
2  * Una forma triangolare composta di quadrati unitari impilati, come questa:
3  * []
4  * []
5  * []
6  * ...
7 */
8 public class Triangle
9 {
10     private int width;
11
12     /**
13      * Costruisce una forma triangolare.
14      * @param aWidth l'ampiezza (e l'altezza) del triangolo
15 }
```

```

15   */
16 public Triangle(int aWidth)
17 {
18     width = aWidth;
19 }
20
21 /**
22  Calcola l'area del triangolo.
23  @return l'area
24 */
25 public int getArea()
26 {
27   if (width <= 0) { return 0; }
28   else if (width == 1) { return 1; }
29   else
30   {
31     Triangle smallerTriangle = new Triangle(width - 1);
32     int smallerArea = smallerTriangle.getArea();
33     return smallerArea + width;
34   }
35 }
36 }

```

### File TriangleTester.java

```

1 public class TriangleTester
2 {
3   public static void main(String[] args)
4   {
5     Triangle t = new Triangle(10);
6     int area = t.getArea();
7     System.out.println("Area: " + area);
8     System.out.println("Expected: 55");
9   }
10 }

```

### Esecuzione del programma

Area: 55  
Expected: 55



### Auto-valutazione

- Perché nella versione finale del metodo `getArea` l'enunciato `else if (width == 1) { return 1; }` non è necessario?
- Come modifichereste il programma per fare in modo che calcoli ricorsivamente l'area di un quadrato?
- In alcune culture i numeri che contengono la cifra 8 sono ritenuti fortunati. Cosa c'è di sbagliato nel metodo seguente, che vorrebbe verificare se un numero è fortunato?

```

public static boolean isLucky(int number)
{
  int lastDigit = number % 10;
  if (lastDigit == 8) { return true; }
}

```

```

        else
    {
        return isLucky(number / 10); // verifica il numero senza l'ultima cifra
    }
}

```

4. Per calcolare una potenza di 2 si può raddoppiare il valore della potenza avente esponente immediatamente inferiore. Ad esempio, se volete calcolare  $2^{11}$  e conoscete il valore di  $2^{10} = 1024$ , potete scrivere che  $2^{11} = 2 \times 2^{10} = 2 \times 1024 = 2048$ . Scrivete il metodo ricorsivo `public static int pow2(int n)` basato su questa osservazione.
5. Analizzate il seguente metodo ricorsivo:

```

public static int mystery(int n)
{
    if (n <= 0) { return 0; }
    else
    {
        int smaller = n - 1;
        return mystery(smaller) + n * n;
    }
}

```

Quanto vale `mystery(4)`?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi E12.1, E12.2 e E12.12, al termine del capitolo.




---

## Errori comuni 12.1

### Ricorsione infinita

Un errore di programmazione molto comune è la ricorsione infinita: un metodo invoca se stesso ripetutamente, senza che si intraveda la fine di questo processo. Il computer ha bisogno di una certa quantità di memoria per gestire ciascuna invocazione, per cui, dopo un certo numero di invocazioni, si esaurisce la memoria disponibile per tale scopo e il programma termina bruscamente segnalando “stack overflow” (un errore di trabocco, *overflow*, nella pila, *stack*, che gestisce le invocazioni).

La ricorsione infinita accade perché i valori dei parametri non diventano più semplici oppure perché manca un caso speciale che ponga fine alle invocazioni. Ad esempio, supponiamo che il metodo `getArea` debba calcolare l’area del triangolo di ampiezza 0: se non fosse per la verifica del caso speciale, il metodo costruirebbe triangoli con ampiezze  $-1, -2, -3$ , e così via.




---

## Errori comuni 12.2

### Effettuare il tracciamento dell'esecuzione di metodi ricorsivi

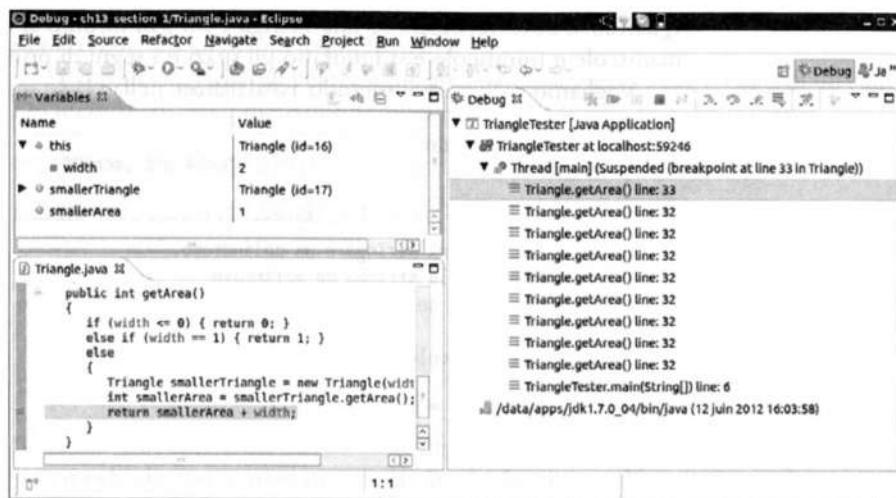
Identificare gli errori in un metodo ricorsivo può essere una vera sfida. Quando, durante una sessione di debugging, impostate un **punto di arresto** (*breakpoint*) in un metodo

ricorsivo, il programma si arresta non appena tale linea di codice viene raggiunta durante l'esecuzione di *una qualsiasi delle invocazioni del metodo ricorsivo*. Supponete di voler effettuare il debugging del metodo ricorsivo `getArea` della classe `Triangle`, eseguendo il programma `TriangleTester` con il debugger fino a quando si ferma, all'inizio del metodo `getArea`. Ispezionate la variabile di esemplare `width`: vale 10.

Rimuovete il punto di arresto e, ora, eseguite il programma fino al raggiungimento dell'enunciato `return smallerArea + width;` (come nella Figura 1). Se controllate nuovamente `width`, il suo valore è 2! Non ha senso: non c'era nessuna istruzione che modificasse il valore di `width`! È forse un errore del programma di debugging?

**Figura 1**

Debugging di un metodo ricorsivo.



No, non lo è: il programma si è arrestato alla prima invocazione ricorsiva di `getArea` che ha raggiunto l'enunciato `return`. Se vi sentite confusi, osservate la **pila delle invocazioni** (*call stack*, a destra nella Figura 1): vedrete che vi sono, in attesa (*pending*), nove invocazioni di `getArea`.

Si può usare il debugger anche con metodi ricorsivi: bisogna soltanto stare particolarmente attenti e dare un'occhiata alla pila delle invocazioni per capire in quale delle invocazioni ci si trova.



## Consigli pratici 12.1

### Pensare ricorsivamente

Per risolvere un problema ricorsivamente serve una diversa impostazione mentale rispetto alla soluzione realizzata mediante la programmazione di cicli. In effetti, la cosa risulta più facile se siete un po' pigri (o fate finta di esserlo) e vi piace che altri facciano la maggior parte del vostro lavoro. Se dovete risolvere un problema complesso, risolvete il problema nei casi più semplici e immaginate che "qualcun altro" faccia la maggior parte del lavoro difficile e pesante. Dovete, poi, solamente capire come trasformare le soluzioni dei casi semplici in una soluzione per il problema completo. Per illustrare la tecnica della ricorsione, consideriamo il seguente problema.

**Problema.** Vogliamo verificare se una frase è un *palindromo* (cioè una stringa che è uguale a se stessa quando se ne inverte l'ordine dei caratteri). Esempi classici di palindromi sono

- A man, a plan, a canal – Panama!
- Go hang a salami, I'm a lasagna hog

e, ovviamente, il palindromo più antico

- Madam, I'm Adam

Quando si deve verificare se una stringa è un palindromo si trascurano le differenze tra maiuscole e minuscole e si ignorano gli spazi e i segni di punteggiatura.

Vogliamo realizzare il metodo `isPalindrome` nella classe seguente:

```
public class Palindromes
{
    ...
    /**
     * Verifica se una stringa è un palindromo.
     * @param text la stringa da verificare
     * @return true se e solo se questa frase è un palindromo
     */
    public static boolean isPalindrome(String text)
    {
        ...
    }
}
```

### Fase 1 Considerate diversi modi per semplificare i dati

Focalizzate il vostro pensiero su un particolare dato (o insieme di dati) in ingresso per il problema che volete risolvere. Pensate a come poter semplificare i dati in modo che si possa porre lo stesso problema con dati più semplici.

Nel considerare dati più semplici, tipicamente si elimina soltanto una piccola parte dei dati originali: ad esempio, un carattere o due caratteri di una stringa, oppure una piccola porzione di una forma geometrica. Altre volte, invece, è più utile dividere i dati a metà e cercare di capire cosa significhi risolvere il problema separatamente per le due parti.

Nel problema della verifica di un palindromo, il dato in ingresso è la stringa che vogliamo verificare. Come possiamo semplificare tale dato? Ecco alcune possibilità

- Eliminare il primo carattere.
- Eliminare l'ultimo carattere.
- Eliminare il primo e l'ultimo carattere.
- Eliminare il carattere centrale.
- Dividere la stringa a metà.

Tutti questi dati di ingresso più semplici sono plausibili per la verifica di un palindromo.

## Fase 2 Combinare le soluzioni dei casi più semplici per risolvere il problema originario

Immaginate di aver ottenuto le soluzioni del vostro problema nei casi più semplici, così come li avete individuati nella Fase 1. Non preoccupatevi di *come* si ottengano queste soluzioni, confidate semplicemente nella loro effettiva disponibilità. Dite a voi stessi: questi sono casi più semplici, per cui qualcun altro risolverà questi problemi al posto mio.

Pensate ora a come poter trasformare la soluzione di questi casi più semplici in una soluzione per i dati a cui state pensando veramente. Può darsi che dobbiate aggiungere una piccola quantità, corrispondente alla piccola quantità che avete eliminato per arrivare al caso più semplice, oppure può darsi che abbiate diviso il problema originario a metà e che abbiate le soluzioni per le due parti, nel qual caso dovete comporre le due soluzioni per arrivare alla soluzione completa.

Considerate i metodi per semplificare i dati in ingresso nella verifica di un palindromo. Dividere la stringa a metà non sembra essere una buona idea. Se dividete

"Madam, I'm Adam"

a metà, ottenete due stringhe:

"Madam, I"

e

"I'm Adam"

Nessuna delle due è un palindromo: dividere i dati a metà e verificare se ciascuna parte sia un palindromo sembra portare in un vicolo cieco.

La semplificazione più promettente è l'eliminazione del primo *e* dell'ultimo carattere. Eliminando la *M* all'inizio e la *m* alla fine, si ottiene

"adam, I'm Ada"

Supponete di poter verificare se tale stringa più corta sia un palindromo: di conseguenza, *ovviamente*, la stringa originaria è un palindromo, dato che abbiamo tolto la stessa lettera all'inizio e alla fine. Molto promettente: quindi, possiamo dire che una parola è un palindromo se

- la prima e l'ultima lettera sono uguali (trascurando le differenze tra maiuscole e minuscole)

e

- la parola che si ottiene eliminando la prima e l'ultima lettera è un palindromo.

Di nuovo, non preoccupatevi di come funzioni la verifica per la stringa più corta: semplicemente, funziona.

C'è, però, un altro caso da considerare: cosa succede se la prima o l'ultima lettera della parola non è, in realtà, una lettera? Ad esempio, la stringa

"A man, a plan, a canal, Panama!"

termina con il carattere !, che non è uguale al carattere A iniziale. Sappiamo, però, che quando verifichiamo se una stringa è un palindromo dobbiamo ignorare tutti i caratteri che non sono lettere, quindi, quando l'ultimo carattere non è una lettera ma il primo carattere lo è, non ha senso eliminare sia il primo che l'ultimo carattere. Questo non è un problema: eliminate semplicemente l'ultimo carattere. Se la stringa più corta che rimane è un palindromo, allora rimane un palindromo anche quando le attaccate un carattere che non sia una lettera.

Lo stesso ragionamento si può applicare se è il primo carattere a non essere una lettera. Abbiamo, quindi, un insieme completo di casi.

- Se il primo e l'ultimo carattere sono lettere, verificate se sono uguali. In tal caso, eliminateli entrambi e verificate la stringa rimanente.
- Altrimenti, se l'ultimo carattere non è una lettera, eliminatelo e verificate la stringa rimanente.
- Altrimenti, il primo carattere non è una lettera: eliminatelo e verificate la stringa rimanente.

In tutti i tre casi potete usare la soluzione del problema più semplice per risolvere il vostro problema iniziale.

### Fase 3 Trovate le soluzioni per i casi più semplici

Un'elaborazione ricorsiva continua a semplificare i propri dati finché arriva a casi molto semplici. Per essere certi che la ricorsione termini, dovete gestire separatamente tali casi più semplici, identificando soluzioni speciali, cosa che generalmente è molto facile.

A volte, però, capita di addentrarsi in questioni filosofiche che riguardano la gestione di casi *degeneri*: stringhe vuote, forme geometriche di area nulla, e così via. In tali casi dovete prendere in considerazione un dato in ingresso un po' più complicato che venga ridotto a tale situazione banale e vedere quale valore dovreste attribuire al caso degenere per fare in modo che al caso un po' più complesso, calcolato secondo le regole identificate nella Fase 2, venga assegnato il valore corretto.

Diamo un'occhiata alle stringhe più semplici per la verifica di un palindromo:

- stringhe con due caratteri
- stringhe con un solo carattere
- stringa vuota

Non è necessario identificare una soluzione speciale per le stringhe di due caratteri: la Fase 2 si applica anche a loro, rimuovendo entrambi i caratteri o uno solo di essi, in base alle regole viste. Dobbiamo, invece, decidere cosa fare con le stringhe di lunghezza 0 e 1: in tali casi la Fase 2 non può essere applicata, semplicemente perché non ci sono due caratteri da eliminare.

La stringa vuota è un palindromo, in quanto è identica a se stessa quando la si legge al contrario. Se pensate che questo ragionamento sia troppo artificioso, considerate la stringa "m̄m". Secondo la regola identificata nella Fase 2, questa stringa (che è certamente un palindromo) è un palindromo se il primo e l'ultimo carattere sono uguali e se la parte

rimanente (cioè la stringa vuota) è un palindromo. Quindi, ha senso affermare che la stringa vuota è un palindromo.

Una stringa costituita da un'unica lettera, come "I", è un palindromo. Cosa possiamo dire in merito a una stringa contenente un unico carattere, che però non sia una lettera, come "?" Eliminando il carattere ! si ottiene la stringa vuota, che è un palindromo. In definitiva, tutte le stringhe di lunghezza 0 e 1 sono palindromi.

#### Fase 4 Implementate la soluzione combinando i casi semplici e il passo di semplificazione

Ora siete pronti per scrivere la soluzione. Gestite separatamente i casi dei dati speciali individuati nella Fase 3. Se i dati non rientrano in uno di questi casi più semplici, seguite il ragionamento delineato nella Fase 2.

Ecco il metodo `isPalindrome`.

```
public static boolean isPalindrome(String text)
{
    int length = text.length();

    // considera separatamente i casi delle stringhe più brevi
    if (length <= 1) { return true; }
    else
    {
        // prendi il primo e l'ultimo carattere, convertiti in minuscolo
        char first = Character.toLowerCase(text.charAt(0));
        char last = Character.toLowerCase(text.charAt(length - 1));

        if (Character.isLetter(first) && Character.isLetter(last))
        {
            // entrambi sono lettere
            if (first == last)
            {
                // elimina il primo e l'ultimo carattere
                String shorter = text.substring(1, length - 1);
                return isPalindrome(shorter);
            }
            else
            {
                return false;
            }
        }
        else if (!Character.isLetter(last))
        {
            // elimina l'ultimo carattere
            String shorter = text.substring(0, length - 1);
            return isPalindrome(shorter);
        }
        else
        {
            // elimina il primo carattere
            String shorter = text.substring(1);
            return isPalindrome(shorter);
        }
    }
}
```



## Esempi completi 12.1

### Ricerca di file

**Problema.** Dovete visualizzare i nomi di tutti i file appartenenti a un albero di cartelle e aventi una determinata estensione (che è la parte terminale del nome). Per risolvere questo problema, vi servono due metodi della classe `File`. Un oggetto `File` può rappresentare una cartella (*directory*) o un semplice file, e il metodo

```
boolean isDirectory()
```

vi consente di dissipare il dubbio. Il metodo

```
File[] listFiles()
```

restituisce un array contenente tutti gli oggetti `File` contenuti in una cartella: possono essere semplici file o altre cartelle.

#### Fase 1 Considerate diversi modi per semplificare i dati

Il problema opera su due dati: un oggetto `File`, che rappresenta un albero di cartelle, e un'estensione per nomi di file. È evidente che manipolando l'estensione non riusciamo a semplificare il problema, mentre c'è un modo ovvio per sfoltire l'albero di cartelle:

- Considerare ciascun file presente nella cartella che si trova alla radice dell'albero.
- Esaminare singolarmente gli alberi costituiti da ciascuna sotto-cartella.

Otteniamo, così, una strategia interessante: cerchiamo nella cartella radice i file aventi il nome che termina con l'estensione richiesta, poi li cerchiamo ricorsivamente in ogni sotto-cartella figlia della radice.

```
Per ogni oggetto File nella radice
Se l'oggetto File è una cartella
    Cerca ricorsivamente in tale cartella.
Altrimenti se il nome termina con l'estensione richiesta
    Visualizza il nome.
```

#### Fase 2 Combinate le soluzioni dei casi più semplici per risolvere il problema originario

Ci viene semplicemente chiesto di visualizzare i nomi dei file che troviamo, per cui non ci sono risultati da combinare.

Se ci fosse stato chiesto di costruire un vettore contenente tutti i file trovati, avremmo inserito nel vettore, inizialmente vuoto, tutti i file trovati nella cartella radice, aggiungendo poi i risultati relativi a ciascuna sotto-cartella.

#### Fase 3 Trovate le soluzioni per i casi più semplici

Il caso più semplice è costituito da un file che non sia una cartella: controlliamo semplicemente se il suo nome termina con l'estensione richiesta e, in caso affermativo, lo visualizziamo.

- Fase 4** Implementate la soluzione combinando i casi semplici e il passo di semplificazione  
Progettiamo la classe `FileFinder`, dotata di un metodo per la ricerca dei file desiderati.

```
public class FileFinder
{
    private File[] children;

    /**
     * Costruisce un oggetto che cerca file in un albero di cartelle.
     * @param startingDirectory la radice dell'albero di cartelle
     */
    public FileFinder(File startingDirectory)
    {
        children = startingDirectory.listFiles();
    }

    /**
     * Visualizza tutti i file il cui nome termina con l'estensione prevista.
     * @param extension un'estensione per file (come ".java")
     */
    public void find(String extension)
    {
        . . .
    }
}
```

Nel nostro caso la fase di semplificazione del problema consiste semplicemente nell'esame di tutti i file e sotto-cartelle:

```
for (File child : children)
{
    if (child.isDirectory())
        Cerca ricorsivamente in child.
    else
        Se il nome di child termina con extension
            Visualizza il nome.
}
```

Ecco il metodo `find` completo:

```
/**
 * Visualizza tutti i file il cui nome termina con l'estensione prevista.
 * @param extension un'estensione per file (come ".java")
 */
public void find(String extension)
{
    for (File child : children)
    {
        String fileName = child.toString();
        if (child.isDirectory())
        {
            FileFinder finder = new FileFinder(child);
            finder.find(extension);
        }
        else if (fileName.endsWith(extension))
```

```
        {
            System.out.println(fileName);
        }
    }
}
```

Il file `FileFinderDemo.java` nella cartella `worked_example_1` del Capitolo 12 del pacchetto dei file scaricabili per questo libro completa la soluzione.

Abbiamo usato un oggetto per ciascuna cartella, ma si potrebbe, in alternativa, usare un metodo ricorsivo statico:

```
/** Visualizza tutti i file il cui nome termina con l'estensione prevista.
 * @param aFile un file o una cartella
 * @param extension un'estensione per file (come ".java")
 */
public static void find(File aFile, String extension)
{
    if (aFile.isDirectory())
    {
        for (File child : aFile.listFiles())
        {
            find(child, extension);
        }
    }
    else
    {
        String fileName = aFile.toString();
        if (fileName.endsWith(extension))
        {
            System.out.println(fileName);
        }
    }
}
```

La strategia è sostanzialmente identica: esaminando un file, verifichiamo se il suo nome termina con l'estensione richiesta, nel qual caso lo visualizziamo; esaminando, invece, una cartella, eseguiamo l'esame per tutti i file e le sotto-cartelle presenti al suo interno.

In questa soluzione abbiamo deciso di accettare, come punto di partenza, sia un file sia una cartella e, per questo motivo, la modalità di invocazione è leggermente diversa. Nella prima soluzione le invocazioni ricorsive vengono effettuate soltanto con cartelle, mentre nella seconda soluzione si invoca il metodo ricorsivo su tutti gli elementi presenti nell'array restituito da `listFiles()`, anche se, nel caso di invocazione con un file semplice, la ricorsione termina subito.

## File FileFinder2.java

```
1 import java.io.File;  
2  
3 public class FileFinder2  
4 {  
5     public static void main(String[] args)  
6     {
```

```

7   File startingDirectory = new File("/home/myname");
8   find(startingDirectory, ".java");
9 }
10 /**
11  * Visualizza tutti i file il cui nome termina con l'estensione prevista.
12  * @param aFile un file o una cartella
13  * @param extension un'estensione per file (come ".java")
14 */
15 public static void find(File aFile, String extension)
16 {
17     if (aFile.isDirectory())
18     {
19         for (File child : aFile.listFiles())
20         {
21             find(child, extension);
22         }
23     }
24     else
25     {
26         String fileName = aFile.toString();
27         if (fileName.endsWith(extension))
28         {
29             System.out.println(fileName);
30         }
31     }
32 }
33 }
34 }
```

## 12.2 Metodi ausiliari ricorsivi

A volte è più semplice trovare una soluzione ricorsiva dopo aver apportato una piccola modifica al problema originario.

A volte è più semplice trovare una soluzione ricorsiva dopo aver apportato una piccola modifica al problema originario, che può, poi, essere risolto invocando un metodo ausiliario ricorsivo.

Ecco un esempio tipico. Considerate la verifica di palindromo vista nei Consigli pratici 12.1: costruire nuove stringhe a ogni passo è poco efficiente. Provate a considerare la seguente modifica al problema: invece di verificare se l'intera frase è un palindromo, verifichiamo se una sottostringa è un palindromo.

```

/***
 * Verifica se una sottostringa della frase è un palindromo.
 * @param start l'indice del primo carattere della sottostringa
 * @param end l'indice dell'ultimo carattere della sottostringa
 * @return true se la sottostringa è un palindromo
 */
public static boolean isPalindrome(String text, int start, int end)
```

Questo metodo si dimostra essere di più facile realizzazione della verifica originaria. Nelle invocazioni ricorsive, modifichiamo semplicemente i parametri `start` e `end` in modo che non vengano prese in esame le coppie di lettere uguali e i caratteri che non sono lettere: non c'è più bisogno di costruire nuovi oggetti di tipo `String` per rappresentare le stringhe che diventano sempre più brevi.

```

public static boolean isPalindrome(String text, int start, int end)
{
    // considera separatamente i casi delle stringhe di lunghezza 0 e 1
    if (start >= end) { return true; }
    else
    {
        // prendi il primo e l'ultimo carattere, convertiti in minuscolo
        char first = Character.toLowerCase(text.charAt(start));
        char last = Character.toLowerCase(text.charAt(end));

        if (Character.isLetter(first) && Character.isLetter(last))
        {
            if (first == last)
            {
                // verifica la sottostringa che non contiene le due lettere uguali
                return isPalindrome(start + 1, end - 1);
            }
            else
            {
                return false;
            }
        }
        else if (!Character.isLetter(last))
        {
            // verifica la sottostringa che non contiene l'ultimo carattere
            return isPalindrome(start, end - 1);
        }
        else
        {
            // verifica la sottostringa che non contiene il primo carattere
            return isPalindrome(start + 1, end);
        }
    }
}

```

Dovreste comunque fornire un metodo per risolvere il problema complessivo, perché l'utente del vostro metodo non è tenuto a conoscere i trucchi che riguardano le posizioni nella sottostringa. Invocate semplicemente il metodo ausiliario con valori di posizioni che provochino la verifica dell'intera stringa:

```

public static boolean isPalindrome(String text)
{
    return isPalindrome(text, 0, text.length() - 1);
}

```

Notate che questo metodo *non* è ricorsivo: il metodo `isPalindrome(String)` invoca il metodo ausiliario, `isPalindrome(String, int, int)`. In questo esempio usiamo il **sovraffunzione** per dichiarare due metodi aventi lo stesso nome: il metodo `isPalindrome` con un parametro di tipo `String` è quello destinato al pubblico utilizzo, mentre il secondo metodo, con un parametro di tipo `String` e due parametri di tipo `int`, è il metodo ausiliario ricorsivo. Se volete, potete evitare di usare metodi sovraccarichi, scegliendo per il metodo ausiliario un nome diverso, come, ad esempio, `substringIsPalindrome`.

Usate la tecnica dei metodi ausiliari ricorsivi ogniqualvolta sia più semplice risolvere un problema ricorsivo leggermente diverso dal problema originale.



## Auto-valutazione

6. Dovevamo necessariamente dare lo stesso nome ai due metodi che abbiamo chiamato `isPalindrome`?
7. In quale momento il metodo ricorsivo `isPalindrome` smette di invocare se stesso?
8. Per calcolare la somma dei valori presenti in un array, aggiungete il primo valore alla somma dei valori rimanenti, eseguendo il calcolo ricorsivamente. Progettate un metodo ausiliario ricorsivo per risolvere questo problema.
9. Come si può scrivere un metodo ricorsivo `public static void sum(int[] a)` senza usare un metodo ausiliario? Perché è meno efficiente?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi E12.6, E12.9 e E12.13, al termine del capitolo.

## 12.3 L'efficienza della ricorsione

Come avete visto in questo capitolo, la ricorsione può essere uno strumento potente per realizzare algoritmi complessi, ma può anche portare ad algoritmi con prestazioni scadenti. In questo paragrafo ci porremo il problema di capire quando la ricorsione sia un bene e quando, invece, sia inefficiente.

Considerate la sequenza di Fibonacci, una sequenza di numeri definita da queste equazioni:

$$\begin{aligned}f_1 &= 1 \\f_2 &= 1 \\f_n &= f_{n-1} + f_{n-2}\end{aligned}$$

In sostanza, ciascun valore della sequenza è la somma dei due valori precedenti. I primi dieci valori della sequenza sono, quindi:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55

È facile estendere indefinitamente la sequenza: basta aggiungere la somma degli ultimi due valori presenti. Ad esempio, il valore successivo è  $34 + 55 = 89$ .

Vorremmo scrivere un metodo che calcola  $f_n$  per qualsiasi valore di  $n$ : traduciamo direttamente la definizione in un metodo ricorsivo.

### File RecursiveFib.java

```

1 import java.util.Scanner;
2
3 /**
4  * Questo programma calcola i numeri di Fibonacci usando un metodo ricorsivo.
5 */
6 public class RecursiveFib
7 {
8     public static void main(String[] args)

```

```

9
10    Scanner in = new Scanner(System.in);
11    System.out.print("Enter n: ");
12    int n = in.nextInt();
13
14    for (int i = 1; i <= n; i++)
15    {
16        long f = fib(i);
17        System.out.println("fib(" + i + ") = " + f);
18    }
19}
20
21 /**
22  * Calcola un numero di Fibonacci.
23  * @param n un numero intero
24  * @return l'n-esimo numero di Fibonacci
25 */
26 public static long fib(int n)
27 {
28     if (n <= 2) { return 1; }
29     else { return fib(n - 1) + fib(n - 2); }
30 }
31 }
```

### Esecuzione del programma

```

Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
...
fib(50) = 12586269025
```

Questo è sicuramente semplice e il metodo funziona in modo corretto, ma osservate attentamente la visualizzazione dei dati prodotti mentre eseguite il programma di prova. Le prime invocazioni del metodo `fib` sono abbastanza veloci, ma, per valori maggiori, il programma lascia trascorrere una quantità sorprendente di tempo tra due visualizzazioni.

Questo non ha senso: armati di carta e penna, nonché di una calcolatrice tascabile, potreste calcolare questi numeri piuttosto in fretta, per cui il computer non dovrebbe assolutamente impiegare tanto tempo.

Per identificare il problema, inseriamo nel metodo alcuni **messaggi di tracciatura** (*trace message*).

### File RecursiveFibTracer.java

```

1 import java.util.Scanner;
2
3 /**
4  * Questo programma visualizza messaggi che mostrano quanto spesso,
```

```
5     per calcolare i numeri di Fibonacci, il metodo ricorsivo invoca se stesso.
6 */
7 public class RecursiveFibTracer
8 {
9     public static void main(String[] args)
10    {
11        Scanner in = new Scanner(System.in);
12        System.out.print("Enter n: ");
13        int n = in.nextInt();
14
15        long f = fib(n);
16
17        System.out.println("fib(" + n + ") = " + f);
18    }
19
20 /**
21     Calcola un numero di Fibonacci.
22     @param n un numero intero
23     @return l'n-esimo numero di Fibonacci
24 */
25 public static long fib(int n)
26 {
27     System.out.println("Entering fib: n = " + n);
28     long f;
29     if (n <= 2) { f = 1; }
30     else { f = fib(n - 1) + fib(n - 2); }
31     System.out.println("Exiting fib: n = " + n
32                         + " return value = " + f);
33     return f;
34 }
35 }
```

## Esecuzione del programma

```
Enter n: 6
Entering fib: n = 6
Entering fib: n = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Exiting fib: n = 5 return value = 5
Entering fib: n = 4
```

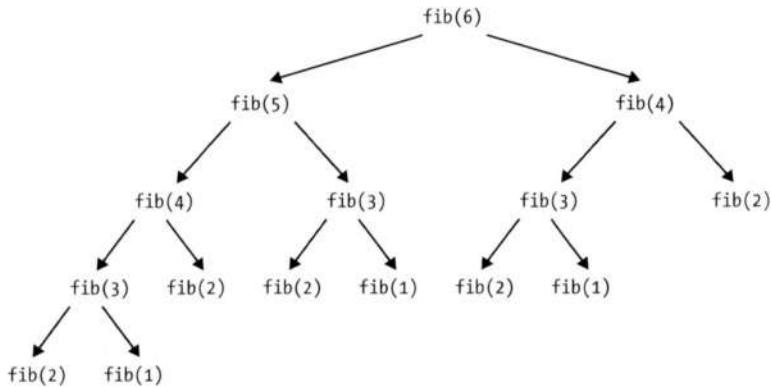
```

Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Exiting fib: n = 6 return value = 8
fib(6) = 8

```

**Figura 2**

Schema delle invocazioni del metodo ricorsivo fib



La Figura 2 mostra lo schema delle invocazioni usate per il calcolo di `fib(6)`. A questo punto risulta evidente cosa renda così lento questo metodo: gli stessi valori vengono calcolati più e più volte. Ad esempio, il calcolo di `fib(6)` richiede di calcolare due volte `fib(4)` e tre volte `fib(3)`. Ciò è molto diverso dai calcoli che faremmo con carta e penna: ci annoteremmo i valori man mano che li calcoliamo, sommando gli ultimi due per generare il successivo fino al raggiungimento del valore desiderato, e nessun valore della sequenza verrebbe calcolato due volte.

Imitando il procedimento “carta e penna”, otteniamo il programma seguente.

### File LoopFib.java

```

1 import java.util.Scanner;
2
3 /**
4  * Questo programma calcola i numeri di Fibonacci con un metodo iterativo.
5 */
6 public class LoopFib
7 {
8     public static void main(String[] args)
9     {
10         Scanner in = new Scanner(System.in);
11         System.out.print("Enter n: ");
12         int n = in.nextInt();
13
14         for (int i = 1; i <= n; i++)

```

```

15     {
16         long f = fib(i);
17         System.out.println("fib(" + i + ") = " + f);
18     }
19 }
20 /**
21     Calcola un numero di Fibonacci.
22     @param n un numero intero
23     @return l'n-esimo numero di Fibonacci
24 */
25 public static long fib(int n)
26 {
27     if (n <= 2) { return 1; }
28     else
29     {
30         long olderValue = 1;
31         long oldValue = 1;
32         long newValue = 1;
33         for (int i = 3; i <= n; i++)
34         {
35             newValue = oldValue + olderValue;
36             olderValue = oldValue;
37             oldValue = newValue;
38         }
39         return newValue;
40     }
41 }
42 }
43 }

```

## Esecuzione del programma

```

Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
...
fib(50) = 12586269025

```

Questo metodo viene eseguito *molto* più velocemente della versione ricorsiva.

In questo esempio relativo al metodo `fib` la soluzione ricorsiva è più facile da realizzare perché segue fedelmente la definizione matematica, ma viene eseguita molto più lentamente della soluzione iterativa, perché calcola più volte molti risultati intermedi.

È sempre possibile velocizzare una soluzione ricorsiva trasformandola in un ciclo? Spesso le soluzioni iterativa e ricorsiva hanno sostanzialmente le stesse prestazioni. Ad esempio, ecco una soluzione iterativa per la verifica di un palindromo.

```

public static boolean isPalindrome(String text)
{
    int start = 0;

```

A volte una soluzione ricorsiva viene eseguita molto più lentamente di una soluzione iterativa del medesimo problema, ma nella maggior parte dei casi la soluzione ricorsiva è soltanto poco più lenta.

```

int end = text.length() - 1;
while (start < end)
{
    char first = Character.toLowerCase(text.charAt(start));
    char last = Character.toLowerCase(text.charAt(end));

    if (Character.isLetter(first) && Character.isLetter(last))
    {
        // entrambi i caratteri sono lettere
        if (first == last)
        {
            start++;
            end--;
        }
        else { return false; }
    }
    if (!Character.isLetter(last)) { end--; }
    if (!Character.isLetter(first)) { start++; }
}
return true;
}

```

Questa soluzione usa due variabili con funzione di indice: `start` e `end`. Il primo indice parte dalla posizione iniziale della stringa e avanza quando una lettera trova una corrispondenza oppure quando viene ignorato un carattere che non è una lettera. Il secondo indice parte dalla posizione finale della stringa e procede a ritroso. Quando le due variabili indice si incontrano, il ciclo termina.

L'iterazione e la ricorsione vengono eseguite pressappoco alla stessa velocità. Se un palindromo ha  $n$  caratteri, l'iterazione esegue il ciclo un numero di volte compreso tra  $n/2$  e  $n$ , in relazione a quanti caratteri sono lettere, perché a ogni passo vengono modificate entrambe le variabili indice oppure soltanto una. Similmente, la soluzione ricorsiva invoca se stessa un numero di volte compreso tra  $n/2$  e  $n$ , perché a ogni passo vengono eliminati uno o due caratteri.

In tale situazione, la soluzione iterativa tende a essere un po' più veloce, perché ciascuna invocazione di un metodo ricorsivo richiede una certa quantità di tempo di elaborazione del processore. In linea di principio, per un compilatore efficiente è possibile eliminare l'esecuzione di invocazioni ricorsive nel caso in cui queste seguano uno schema semplice, ma la maggior parte dei compilatori non fa questo. Da questo punto di vista, una soluzione iterativa è quindi preferibile.

In molti casi una soluzione ricorsiva è più facile da capire e da realizzare correttamente, rispetto a una soluzione iterativa.

Ci sono, però, molti problemi che sono assai più facili da capire e risolvere ricorsivamente di quanto non lo siano iterativamente. A volte, come vedrete nell'esempio presentato nel prossimo paragrafo, non è per niente banale trovare una soluzione iterativa. Nelle soluzioni ricorsive c'è una certa eleganza ed economia di pensiero che le rende più attraenti. Come afferma lo scienziato dell'informazione L. Peter Deutsch (creatore dell'interprete Ghostscript per il linguaggio di descrizione grafico PostScript): "Iterare è umano, usare la ricorsione è divino".



## Auto-valutazione

10. È più veloce calcolare i numeri triangolari ricorsivamente, come visto nel Paragrafo 12.1, oppure usando un ciclo che calcoli  $1 + 2 + 3 + \dots + \text{width}$ ?
11. La funzione fattoriale può essere calcolata con un ciclo, seguendo la definizione  $n! = 1 \times 2 \times \dots \times n$ , oppure ricorsivamente, seguendo la definizione secondo cui  $0! = 1$  e  $n! = (n - 1)! \times n$ . In questo caso l'approccio ricorsivo è inefficiente?
12. Per calcolare la somma dei valori presenti in un array si può suddividere l'array in due parti di dimensioni il più possibile simili (si parla impropriamente di "divisione a metà"), calcolando ricorsivamente le somme relative alle due parti e sommando i risultati. Confrontate le prestazioni di questo algoritmo con quelle di un ciclo che somma i valori.

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R12.7, R12.9, E12.7 e E12.27, al termine del capitolo.

## 12.4 Permutazioni

Le permutazioni di una stringa si possono ottenere in modo più naturale tramite la ricorsione piuttosto che usando un ciclo.

In questo paragrafo vedremo un esempio di ricorsione più complessa, che sarebbe difficile realizzare con un semplice ciclo (come mostra l'Esercizio P12.4, è possibile evitare l'uso della ricorsione, ma la soluzione risultante è abbastanza complicata e non è più veloce).

Progetteremo una classe che elenchi tutte le permutazioni di una stringa (una permutazione è semplicemente una qualsiasi disposizione delle lettere della stringa). Ad esempio, la stringa "eat" ha sei permutazioni (compresa la stringa stessa).

```
"eat"
"eta"
"aet"
"ate"
"tea"
"tae"
```

A questo punto ci serve un'idea per generare le permutazioni ricorsivamente. Consideriamo la stringa "eat" e cerchiamo di semplificare il problema: dapprima genereremo tutte le permutazioni che iniziano con la lettera 'e', poi quelle che iniziano con 'a', infine quelle che iniziano con 't'. Come facciamo a generare le permutazioni che iniziano con 'e'? Ci servono le permutazioni della sottostringa "at". Ma questo non è altro che il problema originario (cioè la generazione di tutte le permutazioni di una stringa) con un dato di ingresso più semplice: la stringa più breve "at". Possiamo quindi usare la ricorsione, generando le permutazioni della sottostringa "at", che sono:

```
"at"
"ta"
```

A ogni permutazione di tale sottostringa si inserisce la lettera 'e' come prefisso, ottenendo le permutazioni di "eat" che iniziano con 'e':

```
"eat"
"eta"
```

Consideriamo ora le permutazioni di "eat" che iniziano con 'a'. Dobbiamo generare le permutazioni delle lettere rimanenti, "et", che sono:

```
"et"
"te"
```

Aggiungiamo la lettera 'a' all'inizio delle stringhe e otteniamo:

```
"aet"
"ate"
```

Allo stesso modo generiamo le permutazioni che iniziano con la lettera 't'.

Avuta l'idea, l'implementazione è quasi banale. Nel metodo `permutations`, scriviamo un ciclo che prenda in esame tutte le posizioni all'interno della parola che deve essere permutata; per ciascuna posizione, *i*, calcoliamo la parola più breve che si ottiene eliminando il carattere *i*-esimo:

```
String shorter = word.substring(0, i) + word.substring(i + 1);
```

Calcoliamo le permutazioni di tale parola più breve:

```
ArrayList<String> shorterPermutations = permutations(shorter);
```

Infine, aggiungiamo a tutte le permutazioni della parola più breve il carattere precedentemente escluso:

```
for (String s : shorterPermutations)
{
    result.add(word.charAt(i) + s);
}
```

Come sempre, dobbiamo prevedere un caso speciale per gestire le stringhe più semplici. La più semplice stringa possibile è la stringa vuota, che ha un'unica permutazione: se stessa.

Ecco la classe `Permutations` completa.

### File Permutations.java

```
1 import java.util.ArrayList;
2
3 /**
4  * Questo programma calcola le permutazioni di una stringa.
5 */
6 public class Permutations
7 {
8     public static void main(String[] args)
9     {
10         for (String s : permutations("eat"))
11         {
12             System.out.println(s);
13         }
14     }
15 }
```

```
14     }
15
16 /**
17     Fornisce tutte le permutazioni della parola data.
18     @param word la stringa di cui calcolare le permutazioni
19     @return un vettore contenente tutte le permutazioni
20 */
21 public static ArrayList<String> permutations(String word)
22 {
23     ArrayList<String> result = new ArrayList<String>();
24
25     // la stringa vuota ha un'unica permutazione: se stessa
26     if (word.length() == 0)
27     {
28         result.add(word);
29         return result;
30     }
31     else
32     {
33         // effettua un ciclo esaminando tutti i caratteri della stringa
34         for (int i = 0; i < word.length(); i++)
35         {
36             // elimina il carattere i-esimo
37             String shorter = word.substring(0, i) + word.substring(i + 1);
38
39             // genera tutte le permutazioni della parola più breve
40             ArrayList<String> shorterPermutations = permutations(shorter);
41
42             // aggiungi il carattere escluso all'inizio di ciascuna
43             // permutazione della parola più breve
44             for (String s : shorterPermutations)
45             {
46                 result.add(word.charAt(i) + s);
47             }
48         }
49         // restituisce tutte le permutazioni
50         return result;
51     }
52 }
53 }
```

### Esecuzione del programma:

```
eat
eta
aet
ate
tea
tae
```

Confrontate le classi Permutations e Triangle, che funzionano secondo lo stesso principio: quando elaborano dati complessi in ingresso, per prima cosa risolvono il problema relativo a dati più semplici, quindi combinano il risultato ottenuto con ulteriori elaborazioni, fornendo i risultati per i dati più complessi. Non c'è davvero alcuna particolare complessità in questa procedura, se pensate alla soluzione soltanto a questo livello: dietro le quinte,



## Computer e società 12.1

### I limiti del calcolo automatico

Vi siete mai chiesti come fa il vostro docente a essere sicuro che i vostri esercizi di programmazione siano corretti? Molto probabilmente guarda la vostra soluzione e forse la esegue con alcuni valori di ingresso di prova, ma solitamente ha una soluzione corretta con cui confrontare la vostra. Ciò suggerisce che ci potrebbe essere un metodo migliore: forse si potrebbe fornire il vostro programma e il suo programma corretto a *un programma che li confronti*, un programma per computer che analizzi entrambi i programmi e che determini se essi calcolano gli stessi risultati. Ovviamente, alla vostra soluzione non si chiede di essere identica a quella che si sa essere corretta: ciò che importa è che esse producano gli stessi risultati quando vengono forniti loro gli stessi dati in ingresso.

Come potrebbe funzionare tale programma comparatore? Bene, il compilatore Java sa come leggere un programma e dare un senso a classi, metodi ed enunciati, per cui sembra plausibile che qualcuno possa, con qualche sforzo, scrivere un programma che legge due programmi Java, analizza ciò che fanno e determina se sono in grado di risolvere lo stesso problema. Evidentemente, un tale programma sarebbe molto interessante per i docenti, perché renderebbe automatica la correzione delle prove di programmazione. Quindi, nonostante tale programma oggi non esista, potremmo essere tentati di provare a svilupparlo, per venderlo alle università di tutto il mondo.

Tuttavia, prima che iniziiate a raccogliere capitali per questa im-

presa, dovete sapere che gli informatici teorici hanno dimostrato che è impossibile sviluppare questo programma, *indipendentemente da quanto duramente ci proviate*.

Esistono diversi problemi irrisolvibili come questo. Il primo, chiamato *problema della terminazione (halting problem)*, fu scoperto dal ricercatore britannico Alan Turing nel 1936. Poiché la sua ricerca era precedente alla costruzione del primo calcolatore reale, Turing dovette ideare una macchina teorica, la **macchina di Turing**, per spiegare come i computer avrebbero potuto funzionare. La macchina di Turing è costituita da un lungo nastro magnetico, una testina di lettura e scrittura, e un programma con istruzioni numerate, del tipo: "Se il simbolo attualmente sotto la testina è *x*, sostituisco con *y*,

La macchina di Turing

Programma

| Numero di istruzione | Se il simbolo sul nastro è | Sostituisci con | Poi sposta la testina verso | E poi prosegui con l'istruzione numero |
|----------------------|----------------------------|-----------------|-----------------------------|----------------------------------------|
| 1                    | 0                          | 2               | destra                      | 2                                      |
|                      | 1                          | 1               | sinistra                    | 4                                      |
| 2                    | 0                          | 0               | destra                      | 2                                      |
|                      | 1                          | 1               | destra                      | 2                                      |
| 3                    | 2                          | 0               | sinistra                    | 3                                      |
|                      | 0                          | 0               | sinistra                    | 3                                      |
|                      | 1                          | 1               | sinistra                    | 3                                      |
| 4                    | 2                          | 2               | destra                      | 1                                      |
|                      | 1                          | 1               | destra                      | 5                                      |
|                      | 2                          | 0               | sinistra                    | 4                                      |

Unità di controllo

Testina lettura/scrittura

Nastro



possibile determinare se il programma, elaborando quell'ingresso, termina. Il problema della terminazione afferma, però, che è impossibile trovare un unico algoritmo decisionale che funzioni con qualsiasi programma e qualsiasi dato in ingresso. Notate che non potete semplicemente eseguire il programma  $P$  con il dato di ingresso  $I$  per rispondere alla domanda: se il programma rimane in esecuzione per 1000 giorni, non sapete se si trova in un ciclo infinito, forse basta soltanto aspettare un altro giorno per vederlo terminare.

Un tale "verificatore di terminazione", se potesse essere scritto, potrebbe essere utile anche per la correzione degli esercizi di programmazione. Un docente potrebbe usarlo con gli elaborati degli studenti per vedere se entrano in un ciclo infinito con un particolare dato di ingresso, evitando di controllarli più a fondo. Tuttavia, come dimostrò Turing, un tale programma non può essere scritto. La sua dimostrazione è ingegnosa e abbastanza semplice.

Supponiamo che il "verificatore di terminazione" (*halt checker*) esista e chiamiamolo  $H$ . A partire da  $H$ , sviluppiamo un altro programma, il programma "killer",  $K$ , che svolge la seguente elaborazione: il suo dato di ingresso è una stringa che contiene il codice sorgente di un programma  $R$ , quindi esso applica il verificatore di terminazione  $H$  al programma  $R$  con la stringa di ingresso  $R$ , cioè verifica se il programma  $R$  si arresta quando gli viene fornito il suo stesso codice sorgente come dato di ingresso. Potrebbe suonare strano che a un programma venga fornito in ingresso il programma stesso, ma ciò non è impossibile: ad esempio, il compilatore Java è scritto in Java e lo potete usare per compilare se stesso. Oppure,

ecco un altro esempio più semplice: potete usare un programma che conta le parole per contare le parole che si trovano nel suo stesso codice sorgente.

Se  $K$  ottiene da  $H$  la risposta che  $R$  termina quando viene applicato a se stesso, esso è programmato per entrare in un ciclo infinito; altrimenti  $K$  termina. In Java, il programma potrebbe assomigliare a questo:

```
public class Killer
{
    public static void main(...)
    {
        String r = legge i dati
        HaltChecker checker = new
            HaltChecker();
        if (checker.check(r, r))
        {
            while (true)
            { // ciclo infinito
            }
        }
        else
        {
            return;
        }
    }
}
```

Ora chiedetevi: qual è la risposta del verificatore di terminazione quando gli viene chiesto se  $K$  termina nel caso in cui gli venga fornito  $K$  come dato di ingresso? Forse scopre che  $K$  entra in un ciclo infinito con tale ingresso. Ma, attenzione: questo non può essere esatto, perché significherebbe che  $checker.check(r, r)$  restituisce false quando  $r$  è il codice sorgente di  $K$ . Come potete facilmente vedere, in tal caso il metodo `main` del programma killer termina, per cui  $K$  non è entrato in un ciclo infinito. Ciò mostra che  $K$  deve terminare quando analizza se stesso, per cui  $checker.check(r, r)$  dovrebbe restituire true. Ma, in tal caso, il metodo `main` del programma killer non termina: entra in un ciclo infinito. Ciò dimostra che

è impossibile, dal punto di vista logico, realizzare un programma che sia in grado di verificare se un programma qualsiasi termina con un particolare dato in ingresso.

È triste sapere che esistono *limiti* alla capacità di elaborazione dei computer: esistono problemi ai quali nessun programma per computer, per quanto ingegnoso, può dare una risposta.

Gli informatici teorici stanno lavorando a molte altre ricerche che riguardano la natura dell'elaborazione. Una domanda importante, ancora senza risposta, riguarda i problemi che richiedono un tempo di elaborazione troppo lungo per essere risolti: questi problemi potrebbero essere intrinsecamente difficili, nel qual caso non avrebbe senso cercare di identificare algoritmi migliori. Queste ricerche teoriche possono avere importanti applicazioni pratiche. Ad esempio, al giorno d'oggi nessuno sa se gli schemi di crittografia più comuni possano essere violati scoprendo nuovi algoritmi: sapere che non esistono algoritmi veloci per violare una particolare codifica potrebbe farci sentire più a nostro agio quando pensiamo alla sicurezza della crittografia.



però, l'elaborazione dei dati più semplici crea, a sua volta, dati da elaborare ancora più semplici, la cui elaborazione crea dati ancora più semplici, e così via, finché i dati che devono essere elaborati sono così semplici che si possono calcolare i risultati senza ulteriore aiuto. Pensare al procedimento è interessante, ma può anche confondere le idee. Ciò che importa è concentrarsi sul livello giusto: costruire la soluzione a partire da un problema un po' più semplice, ignorando il fatto che venga usata la ricorsione per ottenere tale risultato.



### Auto-valutazione

13. Quali sono le permutazioni della parola di quattro lettere beat?
14. La ricorsione che abbiamo progettato per generare le permutazioni si ferma quando deve elaborare una stringa vuota. Con quale semplice modifica la ricorsione si interromperebbe elaborando una stringa di lunghezza 0 o 1?
15. Perché non è facile sviluppare una soluzione iterativa per la generazione di permutazioni?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi E12.14 e E12.15, al termine del capitolo.

## 12.5 Ricorsione mutua

Una ricorsione mutua è caratterizzata da un insieme di metodi cooperanti che si invocano l'un l'altro ripetutamente.

Negli esempi precedenti un metodo invocava se stesso per risolvere un problema più semplice. A volte un insieme di metodi cooperanti si invocano l'un l'altro in modo ricorsivo: in questo paragrafo esamineremo una tipica situazione in cui si realizza tale **ricorsione mutua**, una tecnica significativamente più avanzata della ricorsione semplice di cui abbiamo parlato nei paragrafi precedenti.

Svilupperemo un programma che è in grado di calcolare i valori di espressioni aritmetiche, come:

$$\begin{aligned} & 3+4*5 \\ & (3+4)*5 \\ & 1-(2-(3-(4-5))) \end{aligned}$$

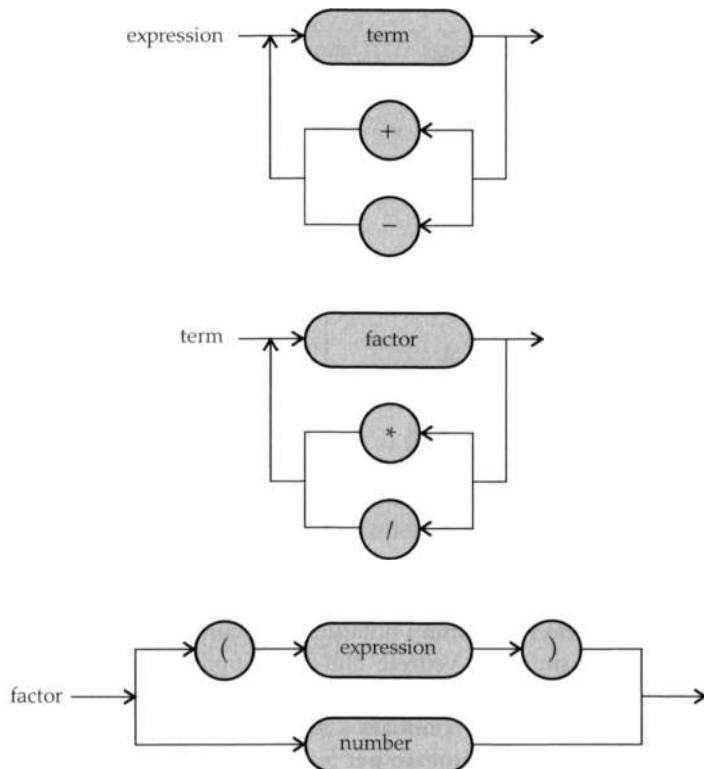
Il calcolo di un'espressione di questo tipo è complicato dal fatto che le operazioni \* e / hanno una precedenza più elevata delle operazioni + e -; inoltre, si possono usare parentesi per raggruppare sotto-espressioni.

La Figura 3 mostra un insieme di **diagrammi sintattici** che descrive la sintassi di queste espressioni. Per capire come funzionano diagrammi di questo tipo, esaminiamo l'espressione  $3+4*5$ .

- Entriamo nel diagramma sintattico corrispondente a una *espressione* (*expression*): la freccia punta direttamente a un *termine* (*term*), per cui non abbiamo alternative.
- Entriamo nel diagramma sintattico corrispondente a un *termine*: la freccia punta direttamente a un *fattore* (*factor*), lasciandoci nuovamente senza alternative.
- Entriamo nel diagramma che descrive un *fattore* e, questa volta, siamo di fronte a una scelta tra due alternative: seguire il ramo superiore o il ramo inferiore. Dato che il primo elemento in ingresso è il numero 3 e non una parentesi tonda aperta, dobbiamo seguire il ramo inferiore.

**Figura 3**

Diagrammi sintattici per la valutazione di un'espressione

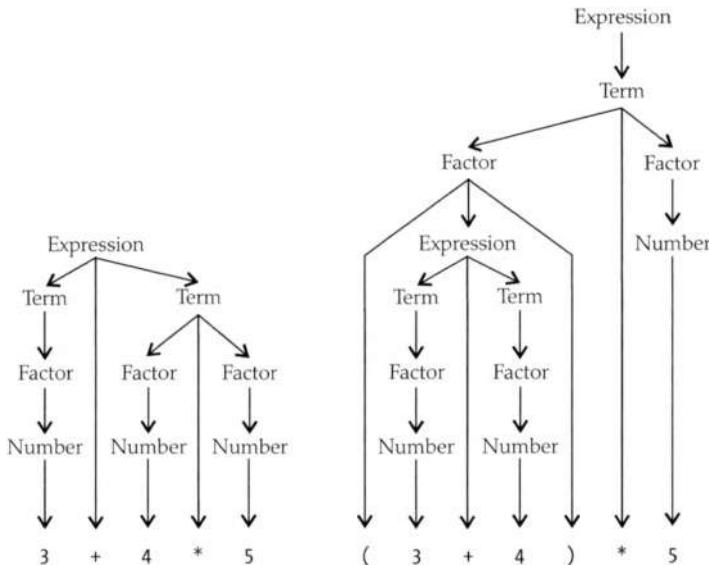


- Accettiamo il dato in ingresso perché corrisponde alla definizione di *numero* (*number*). A questo punto i dati in ingresso ancora da elaborare sono costituiti dalla stringa  $4*5$ .
- Seguiamo la freccia uscente da *numero* fino alla fine di *fattore*: proprio come avviene nell'invocazione di un metodo, torniamo verso l'alto, ritrovandoci alla fine dell'elemento *fattore* presente nel diagramma che descrive un *termine*.
- A questo punto dobbiamo nuovamente scegliere: tornare indietro facendo un ciclo all'interno del diagramma che descrive un *termine* oppure uscire da esso. Il dato successivo in ingresso è il segno  $+$ , che non corrisponde né al simbolo  $*$  né al simbolo  $/$  che sarebbero necessari per entrare nel ciclo, per cui usciamo e torniamo all'*espressione*.
- Di nuovo, possiamo entrare in un ciclo che ci riporta all'indietro oppure uscire dal diagramma, ma in questo caso il segno  $+$  corrisponde ad una delle due scelte che ci portano a compiere un ciclo: accettiamo tale dato in ingresso e torniamo all'elemento *termine*. I dati in ingresso ancora da elaborare sono costituiti dalla stringa  $4*5$ .

Procedendo in questo modo un'*espressione* viene scomposta in una sequenza di *termini*, separati dai segni  $+$  o  $-$ , e ciascun *termine* viene a sua volta scomposto in una sequenza di *fattori*, separati da segni  $*$  o  $/$ ; ciascun *fattore*, infine, è un *numero* o un'*espressione* racchiusa fra parentesi tonde. Questa scomposizione può essere rappresentata mediante un albero e la Figura 4 mostra come vengono derivate dal diagramma sintattico le espressioni  $3+4*5$  e  $(3+4)*5$ .

**Figura 4**

Alberi sintattici  
per due espressioni



In che modo i diagrammi sintattici ci aiutano a calcolare il valore dell'albero? Se guardate gli alberi sintattici, vedrete che essi rappresentano molto accuratamente l'ordine di esecuzione delle operazioni. Nel primo albero 4 e 5 devono essere moltiplicati, quindi il risultato deve essere sommato a 3. Nel secondo albero 3 e 4 devono essere sommati, per poi moltiplicare il risultato per 5.

Alla fine di questo paragrafo troverete l'implementazione della classe `Evaluator` che valuta questo tipo di espressioni. Un oggetto `Evaluator` usa la classe `ExpressionTokenizer`, che scomponete una stringa in elementi, detti **token**: numeri, operatori e parentesi (per semplicità, accettiamo come numeri soltanto numeri interi positivi e non consentiamo la presenza di spazi nei dati in ingresso).

Quando si invoca `nextToken`, l'elemento successivo in ingresso viene restituito sotto forma di stringa. Forniamo anche un altro metodo, `peekToken`, che consente di ispezionare l'elemento successivo senza estrarre. Per capire perché questo metodo sia necessario, considerate il diagramma sintattico di un termine: se l'elemento successivo è `*` o `/` continuerete a moltiplicare o dividere fattori, mentre se è un carattere diverso, come `+` o `-`, dovrete fermarvi senza estrarre veramente, in modo che venga analizzato successivamente.

Per calcolare il valore di un'espressione realizziamo tre metodi: `getExpressionValue`, `getTermValue` e `getFactorValue`. Per prima cosa il metodo `getExpressionValue` invoca `getTermValue` per ottenere il valore del primo termine dell'espressione, quindi verifica se l'elemento successivo è un carattere `+` o `-`: in tal caso invoca `getTermValue` di nuovo e somma o sottrae il valore che ottiene.

```

public int getExpressionValue()
{
    int value = getTermValue();
    boolean done = false;
    while (!done)
    {
        String next = tokenizer.peekToken();
        if ("+".equals(next) || "-".equals(next))

```

```

    {
        tokenizer.nextToken(); // ignora "+" o "-"
        int value2 = getTermValue();
        if ("+".equals(next)) { value = value + value2; }
        else { value = value - value2; }
    }
    else
    {
        done = true;
    }
}
return value;
}

```

Il metodo `getTermValue` invoca `getFactorValue` allo stesso modo, moltiplicando o dividendo i valori dei fattori.

Infine, il metodo `getFactorValue` verifica se l'elemento successivo è un numero o se inizia con una parentesi tonda aperta. Nel primo caso il valore restituito è semplicemente il valore del numero, mentre nel secondo caso il metodo `getFactorValue` invoca ricorsivamente il metodo `getExpressionValue`. In tal modo, i tre metodi risultano essere mutuamente ricorsivi.

```

public int getFactorValue()
{
    int value;
    String next = tokenizer.peekToken();
    if ("(".equals(next))
    {
        tokenizer.nextToken(); // ignora "("
        value = getExpressionValue();
        tokenizer.nextToken(); // ignora ")"
    }
    else
    {
        value = Integer.parseInt(tokenizer.nextToken());
    }
    return value;
}

```

Per evidenziare con chiarezza la ricorsione mutua, seguiamo passo dopo passo la valutazione dell'espressione  $(3+4)*5$ :

- `getExpressionValue` invoca `getTermValue`
  - `getTermValue` invoca `getFactorValue`
    - `getFactorValue` legge  $($  dalla stringa d'ingresso
    - `getFactorValue` invoca `getExpressionValue`
      - `getExpressionValue` restituisce il valore 7, dopo aver letto  $3+4$ ; ecco l'invocazione ricorsiva
      - `getFactorValue` legge  $)$  dalla stringa d'ingresso
      - `getFactorValue` restituisce 7
    - `getTermValue` legge  $*$  e 5 dalla stringa d'ingresso e restituisce 35
  - `getExpressionValue` restituisce 35

Come sempre accade con le soluzioni ricorsive, dobbiamo assicurarci che la ricorsione abbia termine. In questa situazione ciò si vede facilmente, perché, quando il metodo `getExpressionValue` invoca se stesso, la seconda invocazione elabora una sotto-espressione più corta dell'espressione originale. Dato che ogni invocazione ricorsiva estrae dai dati di ingresso alcuni elementi, la ricorsione deve necessariamente terminare.

### File Evaluator.java

```

1  /**
2   * Una classe che calcola il valore di una espressione aritmetica.
3  */
4 public class Evaluator
5 {
6     private ExpressionTokenizer tokenizer;
7
8     /**
9      * Costruisce un valutatore.
10     * @param anExpression una stringa che contiene
11     *                      l'espressione da valutare
12    */
13    public Evaluator(String anExpression)
14    {
15        tokenizer = new ExpressionTokenizer(anExpression);
16    }
17
18    /**
19     * Valuta l'espressione.
20     * @return il valore dell'espressione
21    */
22    public int getExpressionValue()
23    {
24        int value = getTermValue();
25        boolean done = false;
26        while (!done)
27        {
28            String next = tokenizer.peekToken();
29            if ("+".equals(next) || "-".equals(next))
30            {
31                tokenizer.nextToken(); // ignora "+" o "-"
32                int value2 = getTermValue();
33                if ("+".equals(next)) { value = value + value2; }
34                else { value = value - value2; }
35            }
36            else
37            {
38                done = true;
39            }
40        }
41        return value;
42    }
43
44    /**
45     * Valuta il successivo termine nell'espressione.
46     * @return il valore del termine
47    */

```

```
48     public int getTermValue()
49     {
50         int value = getFactorValue();
51         boolean done = false;
52         while (!done)
53         {
54             String next = tokenizer.peekToken();
55             if ("*".equals(next) || "/".equals(next))
56             {
57                 tokenizer.nextToken();
58                 int value2 = getFactorValue();
59                 if ("*".equals(next)) { value = value * value2; }
60                 else { value = value / value2; }
61             }
62             else
63             {
64                 done = true;
65             }
66         }
67         return value;
68     }
69
70 /**
71  * Valuta il successivo fattore nell'espressione.
72  * @return il valore del fattore
73 */
74 public int getFactorValue()
75 {
76     int value;
77     String next = tokenizer.peekToken();
78     if ("(".equals(next))
79     {
80         tokenizer.nextToken(); // ignora "("
81         value = getExpressionValue();
82         tokenizer.nextToken(); // ignora ")"
83     }
84     else
85     {
86         value = Integer.parseInt(tokenizer.nextToken());
87     }
88     return value;
89 }
90 }
```

### File ExpressionTokenizer.java

```
1 /**
2  * Questa classe scomponete un'espressione in elementi
3  * (detti token): numeri, parentesi tonde e operatori.
4 */
5 public class ExpressionTokenizer
6 {
7     private String input;
8     private int start; // l'inizio del token attuale
9     private int end; // la posizione successiva all'ultima del token attuale
10 }
```

```

11  /**
12   * Costruisce uno scompositore (tokenizer).
13   * @param anInput la stringa da scomporre
14  */
15  public ExpressionTokenizer(String anInput)
16  {
17      input = anInput;
18      start = 0;
19      end = 0;
20      nextToken(); // cerca il primo token
21  }
22
23 /**
24   * Restituisce il token successivo senza estrarlo.
25   * @return il token successivo (null se non ce ne sono più)
26  */
27  public String peekToken()
28  {
29      if (start >= input.length()) { return null; }
30      else { return input.substring(start, end); }
31  }
32
33 /**
34   * Restituisce il token successivo e si sposta al token seguente.
35   * @return il token successivo (null se non ce ne sono più)
36  */
37  public String nextToken()
38  {
39      String r = peekToken();
40      start = end;
41      if (start >= input.length()) { return r; }
42      if (Character.isDigit(input.charAt(start)))
43      {
44          end = start + 1;
45          while (end < input.length()
46                  && Character.isDigit(input.charAt(end)))
47          {
48              end++;
49          }
50      }
51      else
52      {
53          end = start + 1;
54      }
55      return r;
56  }
57 }

```

### File ExpressionCalculator.java

```

1 import java.util.Scanner;
2
3 /**
4  * Questo programma calcola il valore di un'espressione
5  * costituita da numeri, operatori aritmetici e parentesi tonde.
6 */

```

```

7 public class ExpressionCalculator
8 {
9     public static void main(String[] args)
10    {
11        Scanner in = new Scanner(System.in);
12        System.out.print("Enter an expression: ");
13        String input = in.nextLine();
14        Evaluator e = new Evaluator(input);
15        int value = e.getExpressionValue();
16        System.out.println(input + " = " + value);
17    }
18 }

```

### Esecuzione del programma

Enter an expression:  $3+4*5$   
 $3+4*5=23$



### Auto-valutazione

16. Qual è la differenza tra termine e fattore? Perché ci servono entrambi questi concetti?
17. Perché il valutatore di espressioni usa la ricorsione mutua?
18. Cosa succede se cercate di valutare l'espressione  $3+4*5$ , che non è valida? In particolare, quale metodo lancia un'eccezione?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R12.13 e E12.21, al termine del capitolo.

## 12.6 Backtracking

La tecnica di backtracking esamina soluzioni parziali, abbandonando quelle che non porteranno a nulla  
 è tornando sui propri passi per prendere in esame altri candidati alla soluzione finale.

Il **backtracking** (letteralmente, “tornare sui propri passi”) è una tecnica di soluzione dei problemi che costruisce soluzioni parziali via via più vicine alla soluzione finale. Se una soluzione parziale non può essere resa completa, la si abbandona e si ritorna a esaminare altre soluzioni candidate.

Il backtracking può essere usato per risolvere cruciverba, per uscire da labirinti o, più in generale, per trovare soluzioni di problemi vincolati da regole. Per risolvere un problema mediante backtracking servono due cose:

1. una procedura che esamina una soluzione parziale e determini se:
  - accettarla come effettiva soluzione del problema, oppure;
  - abbandonarla (perché viola alcune regole oppure perché è evidente che non può portare a una soluzione valida), oppure;
  - continuare a estenderla verso una soluzione completa;
2. una procedura che estenda una soluzione parziale, generando una o più soluzioni che siano più vicine all'obiettivo.

Quindi, la tecnica di backtracking si può implementare mediante il seguente algoritmo ricorsivo:

```

Risovi(soluzioneParziale):
  Esamina(soluzioneParziale)
    Se accettata
      Aggiungi soluzioneParziale alla lista delle soluzioni.
    Altrimenti se non abbandonata
      Per ogni p in Estendi(soluzioneParziale)
        Risovi(p).
  
```

Ovviamente le procedure necessarie per esaminare ed estendere una soluzione parziale dipendono dalla natura del problema.

Come esempio progetteremo un programma che trovi tutte le soluzioni del problema delle otto regine: posizionare otto regine su una scacchiera in modo che nessuna di esse possa attaccarne un'altra secondo le regole degli scacchi. In altre parole, non ci devono essere due regine sulla stessa riga, colonna o diagonale. La Figura 5 mostra una possibile soluzione.

In questo problema è facile esaminare una soluzione parziale: se due regine si attaccano reciprocamente, la soluzione va abbandonata; altrimenti, se ha otto regine, va accettata; altrimenti, va estesa.

L'estensione di una soluzione parziale è altrettanto semplice: basta aggiungere un'ulteriore regina in una casella vuota.

Per ottenere una migliore efficienza, renderemo un po' più sistematica la procedura di estensione: posizioneremo la prima regina nella riga 1, la seconda regina nella riga 2 e così via.

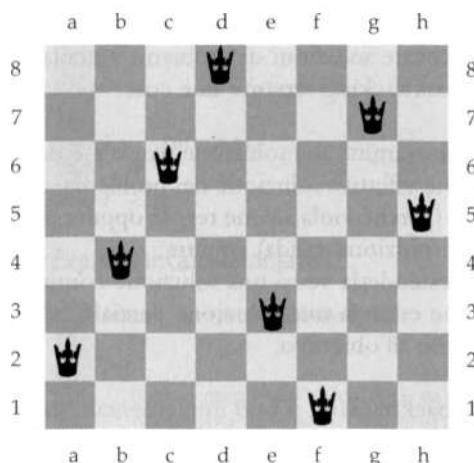
Progettiamo la classe `PartialSolution` che rappresenta una soluzione parziale, con alcune regine posizioante, e che ha metodi per esaminare e per estendere la soluzione.

```

public class PartialSolution
{
  private Queen[] queens;

  public int examine() { . . . }
  public PartialSolution[] extend() { . . . }
}
  
```

**Figura 5**  
Una soluzione  
del problema  
delle otto regine



Il metodo `examine` ha il semplice compito di verificare se, in una soluzione parziale, due regine si attaccano reciprocamente:

```
public int examine()
{
    for (int i = 0; i < queens.length; i++)
    {
        for (int j = i + 1; j < queens.length; j++)
        {
            if (queens[i].attacks(queens[j])) { return ABANDON; }
        }
    }
    if (queens.length == NQUEENS) { return ACCEPT; }
    else { return CONTINUE; }
}
```

Il metodo `extend` riceve una soluzione parziale e ne fa otto copie, aggiungendo a ciascuna una nuova regina in una diversa colonna:

```
public PartialSolution[] extend()
{
    // genera una nuova soluzione per ciascuna colonna
    PartialSolution[] result = new PartialSolution[NQUEENS];
    for (int i = 0; i < result.length; i++)
    {
        int size = queens.length;

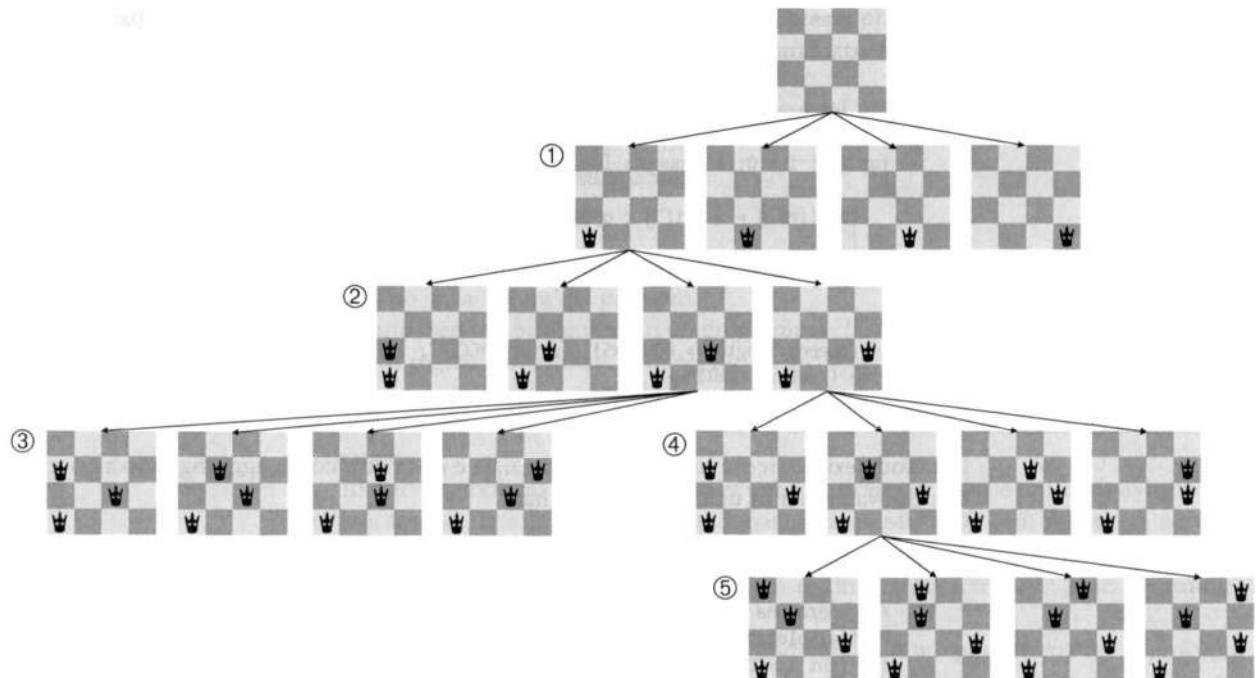
        // la nuova soluzione ha una riga in più di questa
        result[i] = new PartialSolution(size + 1);

        // copia questa soluzione nella nuova
        for (int j = 0; j < size; j++)
        {
            result[i].queens[j] = queens[j];
        }

        // aggiunge la nuova regina nella colonna di indice i
        result[i].queens[size] = new Queen(size, i);
    }
    return result;
}
```

La classe `Queen` si trova al termine di questo paragrafo. L'unico problema che rimane da risolvere è determinare se due regine si attaccano reciprocamente in diagonale, e si può fare facilmente in questo modo: si calcola la pendenza della retta che le congiunge e si verifica se è uguale a  $\pm 1$ . Tale condizione può essere semplificata in questo modo:

$$\begin{aligned} (\text{riga}_2 - \text{riga}_1) / (\text{colonna}_2 - \text{colonna}_1) &= \pm 1 \\ (\text{riga}_2 - \text{riga}_1) &= \pm (\text{colonna}_2 - \text{colonna}_1) \\ |\text{riga}_2 - \text{riga}_1| &= |\text{colonna}_2 - \text{colonna}_1| \end{aligned}$$



**Figura 6** Backtracking nel problema della quattro regine

Osservate con attenzione il metodo `solve` nel programma `EightQueens` presentato nel seguito: è una semplice traduzione dello pseudocodice che abbiamo presentato per la tecnica di backtracking generica. Notate come in questo metodo non ci sia nessun cenno specifico al problema delle otto regine: funziona con qualsiasi tipo di soluzione parziale e per qualsiasi problema, purché vengano definiti opportunamente i metodi `examine` e `extend` (si veda, in proposito, l'Esercizio E12.22).

La Figura 6 mostra il metodo `solve` in azione per risolvere il problema delle quattro regine, cioè il problema discusso finora ma, per semplicità, limitato al caso di una scacchiera  $4 \times 4$ . Partendo da una scacchiera vuota (in alto), vengono generate quattro soluzioni parziali, con una regina nella riga 1 ①. Quando la regina si trova nella colonna 1 esistono quattro soluzioni parziali che hanno un'ulteriore regina nella riga 2 ②, due delle quali vengono abbandonate immediatamente. Ognuna delle altre due genera quattro nuove soluzioni parziali con tre regine sulla scacchiera (③ e ④), ma vengono tutte abbandonate, tranne una. Quella che non è stata abbandonata viene estesa generando quattro soluzioni, ciascuna delle quali ha quattro regine sulla scacchiera, ma vengono tutte abbandonate ⑤. A questo punto l'algoritmo effettua un'azione di backtracking, cioè “torna sui propri passi”, decidendo che non è possibile trovare una soluzione che abbia una regina nella posizione `a1` e procedendo con l'estensione della soluzione parziale che ha un'unica regina nella posizione `b1` (i successivi passi dell'algoritmo non sono mostrati in figura).

Eseguendo il programma si ottiene un elenco di 92 soluzioni, compresa quella già vista nella Figura 5. L'Esercizio E12.23 vi chiede di eliminare dall'elenco quelle soluzioni che sono semplicemente una rotazione o una riflessione di un'altra soluzione.

**File PartialSolution.java**

```
1 import java.util.Arrays;
2
3 /**
4     Una soluzione parziale del rompicapo delle otto regine.
5 */
6 public class PartialSolution
7 {
8     private Queen[] queens;
9     private static final int NQUEENS = 8;
10
11    public static final int ACCEPT = 1;
12    public static final int ABANDON = 2;
13    public static final int CONTINUE = 3;
14
15    /**
16     Costruisce una soluzione parziale di dimensione assegnata.
17     @param size la dimensione assegnata
18    */
19    public PartialSolution(int size)
20    {
21        queens = new Queen[size];
22    }
23
24    /**
25     Esamina una soluzione parziale.
26     @return un valore tra ACCEPT, ABANDON e CONTINUE
27    */
28    public int examine()
29    {
30        for (int i = 0; i < queens.length; i++)
31        {
32            for (int j = i + 1; j < queens.length; j++)
33            {
34                if (queens[i].attacks(queens[j])) { return ABANDON; }
35            }
36        }
37        if (queens.length == NQUEENS) { return ACCEPT; }
38        else { return CONTINUE; }
39    }
40
41    /**
42     Restituisce tutte le estensioni di questa soluzione parziale.
43     @return un array con le soluzioni parziali che estendono questa.
44    */
45    public PartialSolution[] extend()
46    {
47        // genera una nuova soluzione per ciascuna colonna
48        PartialSolution[] result = new PartialSolution[NQUEENS];
49        for (int i = 0; i < result.length; i++)
50        {
51            int size = queens.length;
52
53            // la nuova soluzione ha una riga in più di questa
54            result[i] = new PartialSolution(size + 1);
55        }
56    }
57}
```

```

55     // copia questa soluzione nella nuova
56     for (int j = 0; j < size; j++)
57     {
58         result[i].queens[j] = queens[j];
59     }
60
61     // aggiunge la nuova regina nella colonna di indice i
62     result[i].queens[size] = new Queen(size, i);
63 }
64
65     return result;
66 }
67
68     public String toString() { return Arrays.toString(queens); }
69 }

```

### File Queen.java

```

1 /**
2  * Una regina nel rompicapo delle otto regine.
3 */
4 public class Queen
5 {
6     private int row;
7     private int column;
8
9     /**
10      Costruisce una regina nella posizione assegnata.
11      @param r la riga
12      @param c la colonna
13  */
14     public Queen(int r, int c)
15     {
16         row = r;
17         column = c;
18     }
19
20     /**
21      Verifica se questa regina ne attacca un'altra.
22      @param other l'altra regina
23      @return true se questa regina e l'altra si trovano sulla stessa
24          riga, colonna o diagonale.
25  */
26     public boolean attacks(Queen other)
27     {
28         return row == other.row
29             || column == other.column
30             || Math.abs(row - other.row) == Math.abs(column - other.column);
31     }
32
33     public String toString()
34     {
35         return "" + "abcdefgh".charAt(column) + (row + 1) ;
36     }
37 }

```

### File EightQueen.java

```

1  /**
2   * Risolve il rompicapo delle otto regine usando il backtracking.
3  */
4  public class EightQueens
5  {
6      public static void main(String[] args)
7      {
8          solve(new PartialSolution(0));
9      }
10
11 /**
12  Visualizza tutte le soluzioni del problema che si possono
13  ottenere estendendo una soluzione parziale data.
14  @param sol la soluzione parziale
15 */
16 public static void solve(PartialSolution sol)
17 {
18     int exam = sol.examine();
19     if (exam == PartialSolution.ACCEPT)
20     {
21         System.out.println(sol);
22     }
23     else if (exam == PartialSolution.CONTINUE)
24     {
25         for (PartialSolution p : sol.extend())
26         {
27             solve(p);
28         }
29     }
30 }
31 }
```

### Esecuzione del programma

```

['a1', 'e2', 'h3', 'f4', 'c5', 'g6', 'b7', 'd8']
['a1', 'f2', 'h3', 'c4', 'g5', 'd6', 'b7', 'e8']
['a1', 'g2', 'd3', 'f4', 'h5', 'b6', 'e7', 'c8']
.
.
.
['f1', 'a2', 'e3', 'b4', 'h5', 'c6', 'g7', 'd8']
.
.
.
['h1', 'c2', 'a3', 'f4', 'b5', 'e6', 'g7', 'd8']
['h1', 'd2', 'a3', 'c4', 'f5', 'b6', 'g7', 'e8']
```

(92 soluzioni)



### Auto-valutazione

19. Perché nel metodo `examine` il primo valore di `j` è `i + 1`?
20. Continuate a seguire, passo dopo passo, la soluzione del problema delle quattro regine iniziata nella Figura 6. Quante soluzioni esistono con la prima regina nella colonna a?
21. Quante soluzioni ha complessivamente il problema delle quattro regine?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi E12.22, E12.25 e E12.26, al termine del capitolo.



## Esempi completi 12.2

### Le Torri di Hanoi

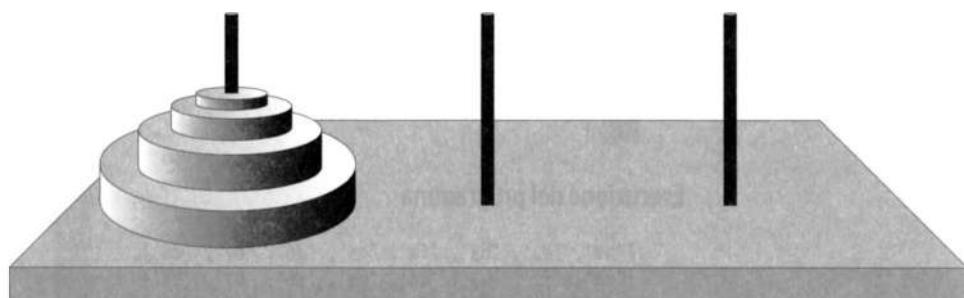
**Descrizione del problema.** Il rompicapo delle “Torri di Hanoi” si gioca con una tavoletta su cui sono infissi tre pioli verticali (*peg*), ciascuno capace di accogliere una pila di dischi circolari, bucati centralmente per infilarsi nel piolo. I dischi hanno diametri tutti diversi e all’inizio del gioco tutti i dischi sono infilati nel primo piolo in ordine di dimensione decrescente verso l’alto (cioè il più grande è in basso e il più piccolo è in cima, disposti ordinatamente, come si può vedere nella Figura 7).

L’obiettivo è quello di spostare tutti i dischi sul terzo piolo. Si può spostare un solo disco per volta, da un piolo a un altro, senza poterlo posizionare sopra un disco più piccolo.

La leggenda dice che in un tempio, presumibilmente nella città di Hanoi, esiste una struttura di questo tipo con sessantaquattro dischi d’oro che vengono continuamente spostati dai monaci seguendo le regole qui indicate: quando avranno spostato tutti i dischi sul terzo piolo, il mondo finirà.

Aiutiamo i monaci scrivendo un programma che visualizzi le mosse da eseguire, nell’ordine che consenta di risolvere il rompicapo.

**Figura 7**  
Le Torri di Hanoi



Consideriamo il problema relativo allo spostamento di  $d$  dischi dal piolo  $p_1$  al piolo  $p_2$ , dove  $p_1$  e  $p_2$  sono i pioli numerati con 1, 2 o 3, con  $p_1 \neq p_2$ . Dato che  $1 + 2 + 3 = 6$ , possiamo calcolare l’indice del terzo piolo, dati  $p_1$  e  $p_2$ , come  $p_3 = 6 - p_1 - p_2$ .

Possiamo spostare i dischi in questo modo:

- spostiamo da  $p_1$  a  $p_3$  i  $d - 1$  dischi che si trovano più in alto;
- spostiamo da  $p_1$  a  $p_2$  il disco che si trovava in fondo alla pila di  $d$  dischi;
- spostiamo da  $p_3$  a  $p_2$  i  $d - 1$  dischi che erano stati parcheggiati temporaneamente.

La prima e la terza fase devono essere gestite ricorsivamente, ma riguardano lo spostamento di un numero di dischi inferiore (di un’unità), quindi, prima o poi, la ricorsione terminerà.

Non è difficile tradurre in Java questo algoritmo. Nella seconda fase visualizziamo, a beneficio dei monaci, la mossa da compiere, in un formato come questo:

Move disk from peg 1 to 3

### File TowersOfHanoiInstructions.java

```
1  /**
2   * Visualizza le istruzioni per risolvere il rompicapo delle Torri di Hanoi.
3  */
4 public class TowersOfHanoiInstructions
5 {
6     public static void main(String[] args)
7     {
8         move(5, 1, 3);
9     }
10
11 /**
12  * Visualizza istruzioni per spostare una pila di dischi da un piolo a un altro.
13  * @param disks il numero di dischi da spostare
14  * @param from il piolo da cui spostare i dischi
15  * @param to il piolo verso cui spostare i dischi
16 */
17 public static void move(int disks, int from, int to)
18 {
19     if (disks > 0)
20     {
21         int other = 6 - from - to;
22         move(disks - 1, from, other);
23         System.out.println("Move disk from peg " + from + " to " + to);
24         move(disks - 1, other, to);
25     }
26 }
27 }
```

### Esecuzione del programma

```
Move disk from peg 1 to 3
Move disk from peg 1 to 2
Move disk from peg 3 to 2
Move disk from peg 1 to 3
Move disk from peg 2 to 1
Move disk from peg 2 to 3
Move disk from peg 1 to 3
Move disk from peg 1 to 2
Move disk from peg 3 to 2
Move disk from peg 3 to 1
Move disk from peg 2 to 1
Move disk from peg 3 to 2
Move disk from peg 1 to 3
Move disk from peg 1 to 2
Move disk from peg 3 to 2
Move disk from peg 1 to 3
Move disk from peg 2 to 1
Move disk from peg 2 to 3
Move disk from peg 1 to 3
Move disk from peg 2 to 1
Move disk from peg 3 to 2
```

```

Move disk from peg 3 to 1
Move disk from peg 2 to 1
Move disk from peg 2 to 3
Move disk from peg 1 to 3
Move disk from peg 1 to 2
Move disk from peg 3 to 2
Move disk from peg 1 to 3
Move disk from peg 2 to 1
Move disk from peg 2 to 3
Move disk from peg 1 to 3

```

Queste istruzioni possono essere sufficienti per i monaci, ma non è semplice capire quale sia la strategia da seguire per risolvere il rompicapo. Cerchiamo di migliorare il programma in modo che esegua effettivamente gli spostamenti e mostri il contenuto delle torri dopo ogni mossa.

Usiamo la classe `Tower` per gestire i dischi presenti in una torre, cioè in un piolo. Ogni disco è rappresentato da un numero intero che indica la sua dimensione, da 1 a  $n$ , che è il numero complessivo di dischi presenti nel rompicapo.

Progettiamo metodi per eliminare il disco che si trova in cima alla torre, per aggiungervi un disco in cima e per mostrare il contenuto della torre come elenco di dimensioni dei suoi dischi, ad esempio [5, 4, 1].

### File Tower.java

```

1 import java.util.ArrayList;
2
3 /**
4     Una torre contenente dischi nel rompicapo delle Torri di Hanoi.
5 */
6 public class Tower
7 {
8     private ArrayList<Integer> disks;
9
10    /**
11        Costruisce una torre con dischi di dimensione decrescente.
12        @param ndisks il numero di dischi
13    */
14    public Tower(int ndisks)
15    {
16        disks = new ArrayList<Integer>();
17        for (int d = ndisks; d >= 1; d--) { disks.add(d); }
18    }
19
20    /**
21        Elimina da questa torre il disco che si trova in cima.
22        @return la dimensione del disco eliminato
23    */
24    public int remove()
25    {
26        return disks.remove(disks.size() - 1);
27    }
28
29    /**

```

```

30     Aggiunge un disco in cima a questa torre.
31     @param size la dimensione del disco da aggiungere
32 */
33     public void add(int size)
34     {
35         if (disks.size() > 0 && disks.get(disks.size() - 1) < size)
36         {
37             throw new IllegalStateException("Disk is too large");
38         }
39         disks.add(size);
40     }
41
42     public String toString() { return disks.toString(); }
43 }
```

Un oggetto `TowersOfHanoi` che rappresenta un rompicapo ha tre torri:

```

public class TowersOfHanoi
{
    private Tower[] towers;

    public TowersOfHanoi(int ndisks)
    {
        towers = new Tower[3];
        towers[0] = new Tower(ndisks);
        towers[1] = new Tower(0);
        towers[2] = new Tower(0);
    }
    . .
}
```

Il suo metodo `move` esegue per prima cosa uno spostamento, poi visualizza il contenuto delle torri:

```

public void move(int disks, int from, int to)
{
    if (disks > 0)
    {
        int other = 3 - from - to;
        move(disks - 1, from, other);
        towers[to].add(towers[from].remove());
        System.out.println(Arrays.toString(towers));
        move(disks - 1, other, to);
    }
}
```

In questo programma abbiamo identificato i pioli con i numeri 0, 1 e 2, per cui l'indice del piolo `other` è dato da `3 - from - to`.

Ecco il metodo `main`:

```

public static void main(String[] args)
{
    final int NDISKS = 5;
    TowersOfHanoi towers = new TowersOfHanoi(NDISKS);
```

```

    towers.move(NDISKS, 0, 2);
}

```

Eseguendo il programma, si ottiene la visualizzazione di queste informazioni:

```

[[5, 4, 3, 2], [], [1]]
[[5, 4, 3], [2], [1]]
[[5, 4, 3], [2, 1], []]
[[5, 4], [2, 1], [3]]
[[5, 4], [2], [3]]
[[5, 4, 1], [], [3, 2]]
[[5, 4], [], [3, 2, 1]]
[[5], [4], [3, 2, 1]]
[[5], [4, 1], [3, 2]]
[[5, 2], [4, 1], [3]]
[[5, 2, 1], [4], [3]]
[[5, 2, 1], [4, 3], []]
[[5, 2], [4, 3], [1]]
[[5], [4, 3, 2], [1]]
[[5], [4, 3, 2, 1], []]
[], [4, 3, 2, 1], [5]]
[[1], [4, 3, 2], [5]]
[[1], [4, 3], [5, 2]]
[[], [4, 3], [5, 2, 1]]
[[3], [4], [5, 2, 1]]
[[3], [4, 1], [5, 2]]
[[3, 2], [4, 1], [5]]
[[3, 2, 1], [4], [5]]
[[3, 2, 1], [], [5, 4]]
[[3, 2], [], [5, 4, 1]]
[[3], [2], [5, 4, 1]]
[[3], [2, 1], [5, 4]]
[[], [2, 1], [5, 4, 3]]
[[1], [2], [5, 4, 3]]
[[1], [], [5, 4, 3, 2]]
[], [], [5, 4, 3, 2, 1]]

```

Decisamente meglio: ora si può vedere come si spostano i dischi e si può verificare facilmente che tutte le mosse effettuate siano valide, controllando che il contenuto di ogni lista sia sempre decrescente.

Come si può notare, per risolvere il rompicapo con 5 dischi servono  $31 = 2^5 - 1$  mosse, quindi per risolvere il rompicapo con 64 dischi, quello dei monaci, occorrono  $2^{64} - 1 = 18446744073709551615$  mosse. Se i monaci riescono a spostare un disco al secondo, per terminare il lavoro servono circa 585 miliardi di anni. Dato che la Terra si è formata circa 4.5 miliardi di anni fa, non dobbiamo preoccuparci molto del fatto che il mondo finisce quando i monaci completeranno il rompicapo.

### File TowersOfHanoiDemo.java

```

1 import java.util.ArrayList;
2 import java.util.Arrays;
3
4 /**

```

```
5     Visualizza una soluzione del rompicapo delle Torri di Hanoi.
6 */
7 public class TowersOfHanoiDemo
8 {
9     public static void main(String[] args)
10    {
11        final int NDISKS = 5;
12        TowersOfHanoi towers = new TowersOfHanoi(NDISKS);
13        towers.move(NDISKS, 0, 2);
14    }
15 }
```

### File TowersOfHanoi.java

```
1 import java.util.Arrays;
2
3 /**
4     Un rompicapo delle Torri di Hanoi con tre torri.
5 */
6 public class TowersOfHanoi
7 {
8     private Tower[] towers;
9
10    /**
11        Costruisce un rompicapo nel quale la prima torre ha il numero di dischi dato.
12        @param ndisks il numero di dischi
13    */
14    public TowersOfHanoi(int ndisks)
15    {
16        towers = new Tower[3];
17        towers[0] = new Tower(ndisks);
18        towers[1] = new Tower(0);
19        towers[2] = new Tower(0);
20    }
21
22    /**
23        Sposta una pila di dischi da un piolo a un altro.
24        @param disks il numero di dischi da spostare
25        @param from il piolo da cui spostare i dischi
26        @param to il piolo verso cui spostare i dischi
27    */
28    public void move(int disks, int from, int to)
29    {
30        if (disks > 0)
31        {
32            int other = 3 - from - to;
33            move(disks - 1, from, other);
34            towers[to].add(towers[from].remove());
35            System.out.println(Arrays.toString(towers));
36            move(disks - 1, other, to);
37        }
38    }
39 }
```

## Riepilogo degli obiettivi di apprendimento

### Comprendere il flusso d'esecuzione di un'elaborazione ricorsiva

- Un'elaborazione ricorsiva risolve un problema usando la soluzione del problema stesso nel caso di dati di ingresso più semplici.
- Perché una ricorsione termini, devono esistere casi speciali per i dati in ingresso più semplici.

### Individuare i metodi ausiliari ricorsivi utili per risolvere un problema

- A volte è più semplice trovare una soluzione ricorsiva dopo aver apportato una piccola modifica al problema originario.

### Confrontare l'efficienza di algoritmi ricorsivi e non ricorsivi

- A volte una soluzione ricorsiva viene eseguita molto più lentamente di una soluzione iterativa del medesimo problema, ma nella maggior parte dei casi la soluzione ricorsiva è soltanto poco più lenta.
- In molti casi una soluzione ricorsiva è più facile da capire e da realizzare correttamente, rispetto a una soluzione iterativa.

### Un esempio di ricorsione complessa che non si può risolvere con un semplice ciclo

- Le permutazioni di una stringa si possono ottenere in modo più naturale tramite la ricorsione piuttosto che usando un ciclo.

### La ricorsione mutua: l'esempio di un valutatore di espressione

- Una ricorsione mutua è caratterizzata da un insieme di metodi cooperanti che si invocano l'un l'altro ripetutamente.

### Usare il backtracking per risolvere problemi che richiedono l'esplorazione di più percorsi

- La tecnica di backtracking esamina soluzioni parziali, abbandonando quelle che non porteranno a nulla e tornando sui propri passi per prendere in esame altri candidati alla soluzione finale.

## Esercizi di riepilogo e approfondimento

- ★ **R12.1.** Definite i seguenti termini:
  - ricorsione
  - iterazione
  - ricorsione infinita
  - metodo ausiliario ricorsivo
  
- ★★ **R12.2.** Delineate, senza implementarla, una soluzione ricorsiva per trovare il valore minimo in un array.
  
- ★★★ **R12.3.** Delineate, senza implementarla, una soluzione ricorsiva per trovare il  $k$ -esimo elemento più piccolo in un array. *Suggerimento:* cercate gli elementi che sono minori dell'elemento iniziale; se questi sono  $m$ , come si procede se  $k \leq m$ ? E se  $k > m$ ?
  
- ★★ **R12.4.** Delineate, senza implementarla, una soluzione ricorsiva per ordinare un array di numeri. *Suggerimento:* per prima cosa trovate il valore minimo nell'array.
  
- ★ **R12.5.** Delineate, senza implementarla, una soluzione ricorsiva per ordinare un array di numeri. *Suggerimento:* per prima cosa ordinate il sotto-array privo dell'elemento iniziale.

- ★ **R12.6.** Scrivete una definizione ricorsiva di  $x^n$ , con  $n \geq 0$ , che sia simile alla definizione ricorsiva dei numeri di Fibonacci. Suggerimento: come si calcola  $x^n$  a partire da  $x^{n-1}$ ? Come si fa terminare la ricorsione?
- ★★ **R12.7.** Nel caso in cui  $n$  sia pari, migliorate l'esercizio precedente calcolando  $x^n$  come  $(x^{n/2})^2$ . Perché questa soluzione è decisamente più veloce? Suggerimento: calcolate  $x^{1023}$  e  $x^{1024}$  in entrambi i modi.
- ★ **R12.8.** Scrivete una definizione ricorsiva di  $n! = 1 \times 2 \times \dots \times n$  che sia simile alla definizione ricorsiva dei numeri di Fibonacci.
- ★★ **R12.9.** Scoprite quante volte la versione ricorsiva di fib invoca se stessa. Usate una variabile statica, fibCount, e incrementatela dopo ogni invocazione di fib. Qual è la relazione tra fib(n) e fibCount?
- ★★★ **R12.10.** Nel rompicapo delle "Torri di Hanoi" visto nella sezione Esempi completi 12.2 indichiamo con mosse( $n$ ) il numero di mosse necessarie per spostare  $n$  dischi. Trovate una formula che esprima mosse( $n$ ) in funzione di mosse( $n - 1$ ), poi dimostrate che  $\text{mosse}(n) = 2^n - 1$ .
- ★★ **R12.11.** Delineate, senza implementarla, una soluzione ricorsiva per generare tutti i sottoinsiemi dell'insieme  $\{1, 2, \dots, n\}$ .
- ★★★ **R12.12.** L'Esercizio P12.5 mostra una strategia iterativa per generare tutte le permutazioni della sequenza  $(0, 1, \dots, n - 1)$ . Spiegate per quale motivo l'algoritmo produce il risultato corretto.
- ★★ **R12.13.** Seguite passo dopo passo l'esecuzione del programma di valutazione delle espressioni, visto nel Paragrafo 12.5, quando analizza  $3 - 4 + 5, 3 - (4 + 5), (3 - 4) * 5$  e  $3 * 4 + 5 * 6$ .

## Esercizi di programmazione

---

- ★ **E12.1.** In una classe Rectangle dotata delle variabili di esemplare width e height (rispettivamente, larghezza e altezza del rettangolo), definite il metodo ricorsivo getArea: costruite un rettangolo la cui larghezza sia inferiore di un'unità rispetto all'originale e invocatene il metodo getArea.
- ★★ **E12.2.** In una classe Square dotata della variabile di esemplare width (dimensione del lato del quadrato), definite il metodo ricorsivo getArea: costruite un quadrato il cui lato sia inferiore di un'unità rispetto all'originale e invocatene il metodo getArea.
- ★ **E12.3.** Scrivete un metodo ricorsivo che scomponga in fattori primi un numero intero  $n$ . Per prima cosa trovate un fattore,  $f$ , poi scomponete ricorsivamente in fattori  $n/f$ .
- ★ **E12.4.** Scrivete un metodo ricorsivo che costruisca una stringa contenente le cifre binarie di un numero intero  $n$ . Se  $n$  è pari, allora l'ultima cifra è 0; se  $n$  è dispari, l'ultima cifra è 1. Calcolate ricorsivamente le altre cifre.
- ★ **E12.5.** Scrivete un metodo ricorsivo String reverse(String text) che inverta il contenuto di una stringa. Ad esempio, reverse("Hello!") restituisce la stringa "olleH". Realizzate una soluzione ricorsiva eliminando il primo carattere, invertendo la stringa rimanente e combinando le due parti.
- ★★ **E12.6.** Risolvete di nuovo l'esercizio precedente usando, però, un metodo ausiliario ricorsivo che inverte una sottostringa.
- ★ **E12.7.** Realizzate iterativamente il metodo reverse dell'Esercizio E12.5.
- ★★ **E12.8.** Usate la ricorsione per realizzare il metodo

```
public static boolean find(String text, String str)
```

che verifica se la stringa `text` contiene la stringa `str`. Ad esempio, `find("Mississippi", "sip")` restituisce `true`. Suggerimento: se il testo inizia con la stringa che state cercando, avete finito; altrimenti, considerate la stringa che si ottiene eliminando il primo carattere.

- ★★ **E12.9.** Usate la ricorsione per realizzare il metodo

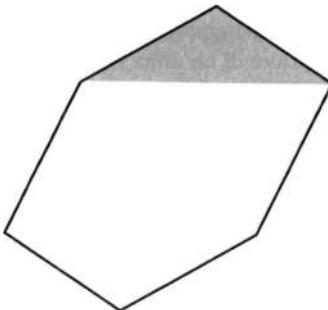
```
public static int indexOf(String text, String str)
```

che restituisce la posizione iniziale della prima sottostringa di `text` che sia uguale alla stringa `str`. Restituite `-1` se `str` non è una sottostringa di `text`. Ad esempio, `indexOf("Mississippi", "sip")` restituisce `6`.

Suggerimento: questo è un po' più difficile del problema precedente, perché dovete tenere traccia di quanto sia lontana dall'inizio della frase la corrispondenza che state cercando; inserite tale valore come parametro di un metodo ausiliario.

- ★ **E12.10.** Usando la ricorsione, trovate l'elemento massimo in un array. Suggerimento: trovate l'elemento massimo nel sottoinsieme che contiene tutti gli elementi dell'array tranne l'ultimo; quindi, confrontate tale massimo con il valore dell'ultimo elemento.
- ★ **E12.11.** Usando la ricorsione, calcolate la somma di tutti i valori presenti in un array.
- ★★ **E12.12.** Usando la ricorsione, calcolate l'area di un poligono. Identificatene una porzione triangolare, come indicato nella figura, e sfruttate il fatto che un triangolo con vertici  $(x_1, y_1)$ ,  $(x_2, y_2)$  e  $(x_3, y_3)$  ha area:

$$\frac{|x_1y_2 + x_2y_3 + x_3y_1 - y_1x_2 - y_2x_3 - y_3x_1|}{2}$$



- ★★ **E12.13.** Il metodo seguente per il calcolo della radice quadrata era noto anche agli antichi greci. Dato un valore  $x > 0$  e un valore  $g$  candidato a essere la radice quadrata di  $x$ , il valore  $(g + x/g) / 2$  è un candidato migliore di  $g$ . Scrivete un metodo ausiliario ricorsivo, `squareRootGuess(double x, double g)`: se  $g^2$  è circa uguale a  $x$  restituisce  $g$ , altrimenti restituisce il valore restituito dal metodo `squareRootGuess` invocato con il candidato migliorato usando la formula descritta in precedenza. Scrivete, poi, il metodo `public static squareRoot(double x)` che usi tale metodo ausiliario.
- ★★★ **E12.14.** Realizzate una classe `SubstringGenerator` che generi tutte le sottostringhe di una stringa. Ad esempio, le sottostringhe della stringa "rum" sono queste sette stringhe:

"r", "ru", "rum", "u", "um", "m", ""

*Suggerimento:* dapprima trovate tutte le sottostringhe che iniziano con il primo carattere, che sono  $n$  se la stringa ha lunghezza  $n$ ; poi, trovate le sottostringhe della stringa che si ottiene eliminando il primo carattere.

- ★★★ **E12.15.** Realizzate una classe `SubsetGenerator` che generi tutti i sottoinsiemi dei caratteri di una stringa. Ad esempio, i sottoinsiemi di caratteri della stringa "rum" sono queste otto stringhe:

```
"rum", "ru", "rm", "r", "um", "u", "m", ""
```

Osservate che non è necessario che i sottoinsiemi siano sottostringhe: ad esempio, "rm" non è una sottostringa di "rum".

- ★★★ **E12.16.** Generate ricorsivamente tutti i modi in cui un vettore può essere suddiviso in una sequenza di sotto-vettori non vuoti. Ad esempio, dato il vettore [1, 7, 2, 9], dovete restituire il seguente vettore di vettori:

```
[[1], [7], [2], [9]], [[1, 7], [2], [9]], [[1], [7, 2], [9]], [[1, 7, 2], [9]],
[[1], [7], [2, 9]], [[1, 7], [2, 9]], [[1], [7, 2, 9]], [[1, 7, 2, 9]]
```

*Suggerimento:* per prima cosa generate tutti i sotto-vettori del vettore privato dell'ultimo elemento, che, poi, può essere aggiunto in fondo all'ultimo sotto-vettore oppure può costituire un sotto-vettore di lunghezza unitaria.

- ★★ **E12.17.** Dato un vettore `a` di numeri interi, trovate ricorsivamente tutti i vettori di elementi di `a` la cui somma è uguale al numero intero  $n$ .

- ★★ **E12.18.** Immaginate di voler salire una scala di  $n$  gradini (identificati da numeri naturali in sequenza) e di poter fare un gradino o due a ogni passo. Elencate ricorsivamente tutte le possibili sequenze di gradini. Ad esempio, se  $n$  vale 5, le sequenze possibili sono:

```
[1, 2, 3, 4, 5], [1, 3, 4, 5], [1, 2, 4, 5], [1, 2, 3, 5], [1, 4, 5]
```

- ★★★ **E12.19.** Risolvete nuovamente l'esercizio precedente nell'ipotesi che a ogni passo si possano salire fino a  $k$  gradini.

- ★★ **E12.20.** Dato un prezzo sotto forma di numero intero, elencate usando la ricorsione tutti i modi possibili per pagarla con banconote da \$100, \$20, \$5 e \$1, senza che siano presenti modi ripetuti.

- ★★ **E12.21.** Migliorate il valutatore di espressioni del Paragrafo 12.5 in modo che possa gestire l'operatore % e l'operatore di "elevamento a potenza" ^ . Ad esempio,  $2 ^ 3$  vale 8. Come in matematica, l'elevamento a potenza deve avere la precedenza sulla moltiplicazione:  $5 * 2 ^ 3$  vale 40.

- ★★ **E12.22.** L'algoritmo di backtracking funziona per qualsiasi problema le cui soluzioni parziali possano essere valutate ed estese. Definite un tipo interfaccia, `PartialSolution`, dotato dei metodi `examine` e `extend`; poi, realizzate un metodo, `solve`, che elabori oggetti di tipo `PartialSolution` e una classe, `EightQueensPartialSolution`, che implementi l'interfaccia.

- ★★★ **E12.23.** Migliorate il programma che risolve il rompicapo delle otto regine in modo che non visualizzi soluzioni che siano rotazioni o riflessioni di soluzioni visualizzate in precedenza. Il programma deve visualizzare dodici soluzioni.

- ★★★ **E12.24.** Migliorate il programma che risolve il rompicapo delle otto regine in modo che le soluzioni vengano scritte in un file HTML, usando tabelle con sfondo bianco e nero per le scacchiere e il carattere Unicode '\u2655' per le regine.

- ★★ **E12.25.** Rendete più generale il programma che risolve il rompicapo delle otto regine in modo che risolva il problema per  $n$  regine. Il programma deve chiedere all'utente il valore di  $n$  e, poi, visualizzare le soluzioni
- ★★ **E12.26.** Usando il backtracking, scrivete un programma che risolva rompicapi rappresentati da addizioni tra stringhe di lettere, dove ogni lettera deve essere sostituita da una cifra, come questi:

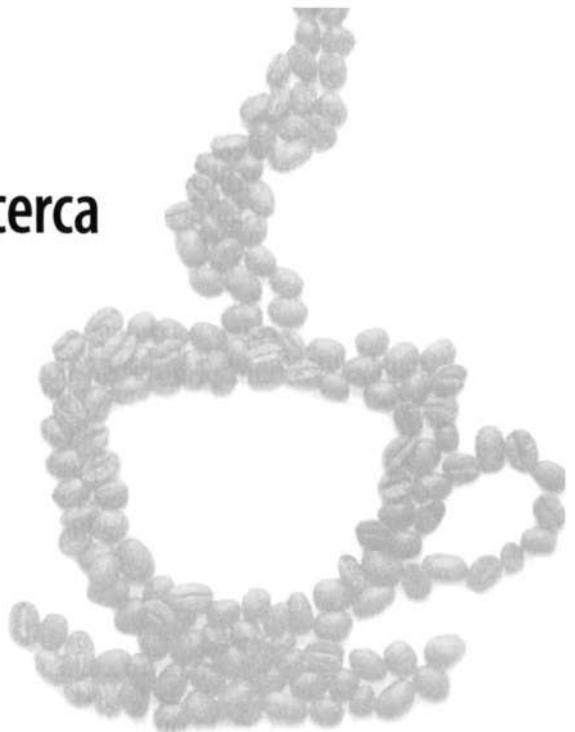
send + more = money  
base + ball = games  
kyoto + osaka = tokyo

- ★★ **E12.27.** Il calcolo ricorsivo dei numeri di Fibonacci può essere accelerato in modo significativo tenendo traccia dei valori che sono già stati calcolati: realizzate una nuova versione del metodo `fib` usando questa strategia. Ogni volta che restituite un nuovo valore, memorizzatelo anche in un array ausiliario; prima di iniziare il calcolo di un valore, consultate l'array per vedere se tale calcolo sia già stato eseguito. Confrontate il tempo di esecuzione della versione così migliorata con le realizzazioni originali, ricorsiva e iterativa.

Sul sito web dedicato al libro si trova una raccolta di progetti di programmazione più complessi.

# 13

## Ordinamento e ricerca



### Obiettivi del capitolo

- Studiare alcuni algoritmi di ordinamento e di ricerca
- Osservare come algoritmi che risolvono lo stesso problema possano avere prestazioni molto diverse
- Capire la notazione O-grande
- Imparare a stimare le prestazioni di algoritmi e a confrontarle
- Scrivere codice per misurare il tempo d'esecuzione di un programma

L'ordinamento è una delle operazioni più frequenti nell'elaborazione dei dati: ad esempio, spesso c'è la necessità di visualizzare un array di dipendenti in ordine alfabetico oppure in ordine di stipendio. In questo capitolo imparerete alcuni algoritmi di ordinamento e le tecniche che consentono di confrontare le loro prestazioni. Tali tecniche non sono utili soltanto per i metodi di ordinamento, ma anche per l'analisi di molti altri algoritmi.

Dopo aver ordinato un array di elementi vi si possono facilmente effettuare ricerche, per verificare la presenza di specifici elementi. Studieremo l'algoritmo di *ricerca binaria*, che esegue una ricerca rapida.

## 13.1 Ordinamento per selezione

In questo paragrafo illustreremo un algoritmo di ordinamento, il primo di quelli che vedremo. Un **algoritmo di ordinamento** (*sorting algorithm*) dispone gli elementi di una raccolta di dati in modo che, al termine, siano memorizzati in qualche ordine specifico. Per rimanere su esempi semplici, considereremo l'ordinamento di un array di numeri interi, prima di passare a ordinare stringhe o dati più complessi. Considerate il seguente array a:

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] |
| 11  | 9   | 17  | 5   | 12  |

L'algoritmo di ordinamento per selezione ordina un array cercando ripetutamente l'elemento minimo della zona terminale non ancora ordinata, spostandolo all'inizio della zona stessa.

Una prima, elementare fase consiste nella ricerca dell'elemento minimo, che in questo caso è 5 e si trova in  $a[3]$ . Dovremmo spostare 5 all'inizio dell'array, in  $a[0]$ , dove naturalmente c'è già un elemento, precisamente il numero 11. Di conseguenza non possiamo semplicemente spostare  $a[3]$  in  $a[0]$  senza spostare 11 da qualche altra parte. Non sappiamo ancora dove dovrà andare a finire il numero 11, ma sappiamo con certezza che non deve stare in  $a[0]$ . Ce lo togliamo semplicemente di torno scambiandolo con  $a[3]$ .

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] |
| 5   | 9   | 17  | 11  | 12  |

Ora il primo elemento si trova nel posto giusto. Nella figura la zona più scura indica la porzione di array che è già stata ordinata, mentre la parte rimanente è ancora da ordinare.

Successivamente cerchiamo l'elemento minimo tra quelli rimanenti,  $a[1] \dots a[4]$ . Tale valore minimo, 9, si trova già nella posizione corretta: in questo caso non dobbiamo fare nulla e possiamo semplicemente estendere di una posizione verso destra la porzione di array che risulta essere già ordinata.

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] |
| 5   | 9   | 17  | 11  | 12  |

Ripetiamo il procedimento. Il valore minimo della zona non ordinata è 11, che deve essere scambiato con il primo valore di tale zona, 17.

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] |
| 5   | 9   | 17  | 11  | 12  |

Ora la zona non ordinata è composta da due soli elementi, ma continuiamo ad applicare la stessa strategia vincente. Il valore minimo è 12 e lo scambiamo con il primo valore, 17.

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] |
| 5   | 9   | 17  | 11  | 12  |

Questo ci porta ad avere una zona non ancora elaborata di lunghezza 1, ma, naturalmente, una zona di lunghezza 1 è sempre ordinata. Abbiamo finito.

Proviamo a scrivere il codice per questo algoritmo, chiamato **ordinamento per selezione** (*selection sort*). Per questo programma, come per gli altri di questo capitolo, utilizzeremo un metodo ausiliario per generare un array di valori casuali: lo inseriamo nella classe `ArrayUtil` per non essere costretti a ripeterlo in ogni esempio. Per visualizzare il contenuto di un array usiamo invece la stringa restituita dal metodo statico `toString` della classe `java.util.Arrays` (visto nel Paragrafo 7.3.4). Aggiungiamo alla classe `ArrayUtil` anche un metodo, `swap`, che scambia tra loro due elementi di un array (nel Paragrafo 7.3.8 abbiamo presentato alcuni dettagli relativi allo scambio di elementi in un array).

Questo algoritmo ordinerà un array di numeri interi. Se la velocità non fosse un argomento di interesse o se non esistessero metodi di ordinamento migliori, potremmo interrompere qui la discussione sull'ordinamento. Tuttavia, come mostrerà il paragrafo successivo, questo algoritmo, pur essendo assolutamente corretto, ha prestazioni davvero deludenti quando viene eseguito su grandi insiemi di dati.

Nella sezione Argomenti avanzati 13.2 viene presentato l'algoritmo di ordinamento per inserimento (*insertion sort*), un altro algoritmo di ordinamento molto semplice.

### File SelectionSorter.java

```
1  /**
2   * Il metodo sort di questa classe ordina un array
3   * usando l'algoritmo di ordinamento per selezione.
4  */
5  public class SelectionSorter
6  {
7      /**
8       * Ordina un array usando l'ordinamento per selezione.
9       * @param a l'array da ordinare
10      */
11     public static void sort(int[] a)
12     {
13         for (int i = 0; i < a.length - 1; i++)
14         {
15             int minPos = minimumPosition(a, i);
16             ArrayUtil.swap(a, minPos, i);
17         }
18     }
19
20     /**
21      Trova l'elemento minimo in una zona terminale di un array.
22      @param a l'array da analizzare
23      @param from la prima posizione da analizzare in a
24      @return la posizione dell'elemento minimo nella porzione di array
25      a[from] . . . a[a.length - 1]
26    */
27    private static int minimumPosition(int[] a, int from)
28    {
29        int minPos = from;
30        for (int i = from + 1; i < a.length; i++)
31        {
32            if (a[i] < a[minPos]) { minPos = i; }
```

```

33     }
34     return minPos;
35   }
36 }
```

### File SelectionSortDemo.java

```

1 import java.util.Arrays;
2
3 /**
4  * Questo programma applica l'algoritmo di ordinamento
5  * per selezione a un array riempito con numeri casuali.
6 */
7 public class SelectionSortDemo
8 {
9   public static void main(String[] args)
10  {
11    int[] a = ArrayUtil.randomIntArray(20, 100);
12    System.out.println(Arrays.toString(a));
13
14    SelectionSorter.sort(a);
15
16    System.out.println(Arrays.toString(a));
17  }
18 }
```

### File ArrayUtil.java

```

1 import java.util.Random;
2
3 /**
4  * Questa classe contiene metodi utili per elaborare array.
5 */
6 public class ArrayUtil
7 {
8   private static Random generator = new Random();
9
10  /**
11   * Costruisce un array contenente numeri interi casuali.
12   * @param length la lunghezza dell'array
13   * @param n il numero di valori diversi possibili
14   * @return un array contenente length numeri
15   *         casuali compresi fra 0 e n - 1
16  */
17  public static int[] randomIntArray(int length, int n)
18  {
19    int[] a = new int[length];
20    for (int i = 0; i < a.length; i++)
21    {
22      a[i] = generator.nextInt(n);
23    }
24
25    return a;
26  }
27 }
```

```

28  /**
29   Scambia tra loro due elementi di un array.
30   @param a l'array
31   @param i la posizione del primo elemento da scambiare
32   @param j la posizione del secondo elemento da scambiare
33 */
34 public static void swap(int[] a, int i, int j)
35 {
36     int temp = a[i];
37     a[i] = a[j];
38     a[j] = temp;
39 }
40 }
```

### Esempio di esecuzione del programma

```
[65, 46, 14, 52, 38, 2, 96, 39, 14, 33, 13, 4, 24, 99, 89, 77, 73, 87, 36, 81]
[2, 4, 13, 14, 14, 24, 33, 36, 38, 39, 46, 52, 65, 73, 77, 81, 87, 89, 96, 99]
```



### Auto-valutazione

- Perché nel metodo `swap` serve la variabile `temp`? Cosa succederebbe seassegnassimo semplicemente `a[i]` a `a[j]` e `a[j]` a `a[i]`?
- Quali sono i passi compiuti dall'algoritmo di ordinamento per selezione nell'ordinare la sequenza 6 5 4 3 2 1?
- Come si può modificare l'algoritmo di ordinamento per selezione perché ordini gli elementi in ordine decrescente (cioè con l'elemento massimo all'inizio dell'array)?
- Immaginate di aver modificato l'algoritmo di ordinamento per selezione in modo che parta dalla fine dell'array, procedendo verso la posizione iniziale: ad ogni passo l'elemento che si trova nella posizione in esame viene scambiato con il valore minimo. Che risultati produce questo algoritmo modificato?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R13.2, R13.12, E13.1 e E13.2, al termine del capitolo.

## 13.2 Prestazioni dell'ordinamento per selezione

Per misurare le prestazioni temporali di un programma potreste semplicemente eseguirlo e usare un cronometro per misurare il tempo trascorso, ma la maggior parte dei nostri programmi viene eseguita molto rapidamente e non sarebbe facile misurare i tempi in modo accurato in questo modo. Inoltre, anche quando l'esecuzione di un programma richiede un tempo percepibile, una parte di quel tempo viene semplicemente usata per caricare il programma dal disco alla memoria e per visualizzare i risultati sullo schermo (cose per le quali non dovremmo penalizzarla).

Per misurare in modo più accurato il tempo di esecuzione di un algoritmo progetteremo la classe `StopWatch`, che funziona proprio come un vero cronometro: potete farlo partire, fermarlo e leggere il tempo trascorso. La classe usa il metodo `System.currentTimeMillis`,

che restituisce il numero di millisecondi che sono trascorsi dalla mezzanotte del giorno 1 gennaio 1970. Ovviamente non ci interessa il numero assoluto di secondi trascorsi da quell'istante particolare, però la *differenza* fra due conteggi di questo genere ci fornisce la durata di un intervallo temporale, misurata in millisecondi.

Ecco il codice della classe `StopWatch`:

### File `StopWatch.java`

```
1  /**
2   * Un cronometro misura il tempo che trascorre mentre è in azione. Potete
3   * avviare e arrestare ripetutamente il cronometro. Potete utilizzare un
4   * cronometro per misurare il tempo di esecuzione di un programma.
5  */
6  public class StopWatch
7  {
8      private long elapsedTime;
9      private long startTime;
10     private boolean isRunning;
11
12     /**
13      Costruisce un cronometro fermo, con il tempo
14      totale misurato che vale zero.
15     */
16     public StopWatch()
17     {
18         reset();
19     }
20
21     /**
22      Fa partire il cronometro, iniziando a misurare il tempo.
23     */
24     public void start()
25     {
26         if (isRunning) { return; }
27         isRunning = true;
28         startTime = System.currentTimeMillis();
29     }
30
31     /**
32      Ferma il cronometro. Il tempo trascorso dall'ultimo avvio del
33      cronometro viene sommato al tempo totale misurato.
34     */
35     public void stop()
36     {
37         if (!isRunning) { return; }
38         isRunning = false;
39         long endTime = System.currentTimeMillis();
40         elapsedTime = elapsedTime + endTime - startTime;
41     }
42
43     /**
44      Restituisce il tempo totale misurato.
45      @return il tempo totale misurato
46  */
```

```
47 public long getElapsedTime()
48 {
49     if (isRunning)
50     {
51         long endTime = System.currentTimeMillis();
52         return = elapsedTime + endTime - startTime;
53     }
54     else
55     {
56         return elapsedTime;
57     }
58 }
59
60 /**
61     Ferma il cronometro e azzerà il tempo totale.
62 */
63 public void reset()
64 {
65     elapsedTime = 0;
66     isRunning = false;
67 }
68 }
```

Ed ecco come utilizzeremo il cronometro per misurare le prestazioni dell'algoritmo di ordinamento.

### File SelectionSortTimer.java

```
1 import java.util.Scanner;
2
3 /**
4     Questo programma misura il tempo richiesto per
5     ordinare, mediante l'algoritmo di ordinamento per
6     selezione, un array di dimensione specificata dall'utente.
7 */
8 public class SelectionSortTimer
9 {
10    public static void main(String[] args)
11    {
12        Scanner in = new Scanner(System.in);
13        System.out.print("Enter array size: ");
14        int n = in.nextInt();
15
16        // costruisce un array casuale
17
18        int[] a = ArrayUtil.randomIntArray(n, 100);
19
20        // usa il cronometro per misurare il tempo
21
22        Stopwatch timer = new Stopwatch();
23
24        timer.start();
25        SelectionSorter.sort(a);
26        timer.stop();
```

```

27
28     System.out.println("Elapsed time: "
29         + timer.getElapsedTime() + " milliseconds");
30 }
31 }

```

### Esempio di esecuzione del programma

```

Enter array size: 50000
Elapsed time: 13321 milliseconds

```

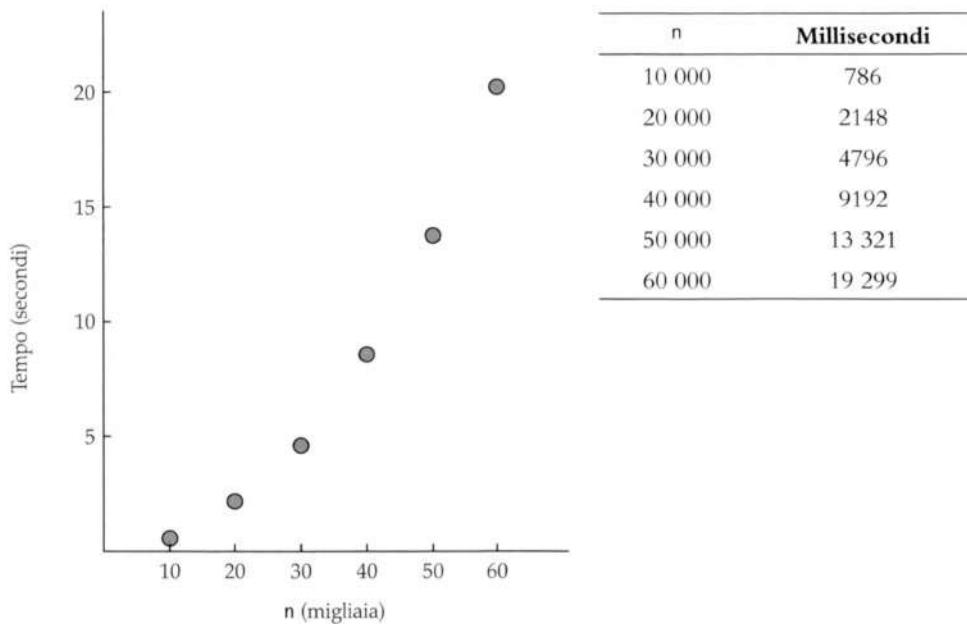
Per misurare il tempo di esecuzione di un metodo, chiedete l'ora al sistema subito prima e subito dopo l'invocazione del metodo stesso.

Avviando la rilevazione del tempo immediatamente prima dell'ordinamento e arrestandola subito dopo si misura il tempo richiesto per ordinare i dati, senza tener conto del tempo che occorre per le operazioni di input e di output.

La Figura 1 mostra i risultati di alcune esecuzioni del programma. Queste rilevazioni sono state ottenute con un processore Intel a 2 GHz, con sistema operativo Linux e Java 6. Su un altro computer i numeri effettivi potrebbero essere diversi, ma la relazione fra loro resterebbe la stessa.

La Figura 1 mostra anche un grafico delle rilevazioni: come potete vedere, raddoppiando la dimensione dell'insieme dei dati, il tempo che occorre per ordinarli è circa il quadruplo.

**Figura 1**  
Tempo impiegato dall'ordinamento per selezione.



### Auto-valutazione

5. Quanti secondi sarebbero necessari, approssimativamente, per ordinare un insieme di dati contenente 80000 valori?
6. Osservate il grafico della Figura 1: a quale curva matematica assomiglia?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi E13.3 e E13.9, al termine del capitolo.

## 13.3 Analisi delle prestazioni dell'ordinamento per selezione

Proviamo a contare le operazioni che il programma deve eseguire per ordinare un array usando l'algoritmo di ordinamento per selezione. Non sappiamo, in realtà, quante operazioni macchina vengono generate per ciascuna istruzione Java e neppure quali di queste istruzioni impiegano più tempo di altre, però possiamo fare una semplificazione: ci limiteremo a contare il numero di *visite* di elementi dell'array, sapendo che ciascuna visita richiede all'incirca la stessa quantità di lavoro dovuta anche ad altre operazioni correlate, quali l'incremento di indici e il confronto di valori.

Indichiamo con  $n$  la dimensione dell'array. Per prima cosa dobbiamo trovare il minimo fra  $n$  numeri: ciò richiede la visita degli  $n$  elementi dell'array. Poi scambiamo gli elementi, operazione che richiede due visite (potreste argomentare che esiste una certa probabilità che non si debbano scambiare i valori: è vero, e si potrebbe migliorare l'analisi per tenere conto di questa osservazione, ma, come vedremo fra un momento, se anche lo facessimo non altereremmo la conclusione complessiva). Nel passo successivo dobbiamo visitare soltanto  $n - 1$  elementi per trovare il minimo; nel passo ancora seguente vengono visitati soltanto  $n - 2$  elementi; l'ultimo passo visita soltanto due elementi, tra i quali deve nuovamente trovare il minimo. Ciascun passo, poi, richiede 2 visite per scambiare gli elementi. Quindi, il numero totale delle visite è:

$$\begin{aligned} n + 2 + (n - 1) + 2 + \cdots + 2 + 2 &= (n + (n - 1) + \cdots + 2) + (n - 1) \cdot 2 \\ &= (2 + \cdots + (n - 1) + n) + (n - 1) \cdot 2 \\ &= \frac{n(n + 1)}{2} - 1 + (n - 1) \cdot 2 \end{aligned}$$

perché

$$1 + 2 + \cdots + (n - 1) + n = \frac{n(n + 1)}{2}$$

Sviluppando le moltiplicazioni e raccogliendo  $n$  a fattore comune, troviamo che il numero delle visite è:

$$\frac{1}{2}n^2 + \frac{5}{2}n - 3$$

Abbiamo ottenuto un'equazione quadratica in  $n$ : questo spiega perché il grafico della Figura 1 assomiglia a una parabola.

Ora semplifichiamo ulteriormente l'analisi. Quando usiamo un valore di  $n$  elevato, come 1000 o 2000, il termine  $(1/2)n^2$  è uguale a 500000 o, rispettivamente, 2000000. I termini di grado inferiore,  $(5/2)n - 3$ , non contribuiscono più di tanto, valgono appena 2497 o, rispettivamente, 4997, una goccia nel mare rispetto alle centinaia di migliaia o

addirittura ai milioni di visite corrispondenti al termine quadratico. Semplicemente, ignoreremo questi termini di grado inferiore, così come il fattore costante  $1/2$ : non ci interessa il numero effettivo delle visite per un singolo valore di  $n$ , vogliamo soltanto confrontare i rapporti dei valori per diversi valori di  $n$ . Per esempio, possiamo dire che ordinare un array di 2000 numeri richiede un numero di visite pari a 4 volte quelle necessarie per ordinare un array di 1000 numeri:

$$\frac{\left(\frac{1}{2} \cdot 2000^2\right)}{\left(\frac{1}{2} \cdot 1000^2\right)} = 4$$

Il fattore  $1/2$  si elide in confronti di questo genere: diremo semplicemente che “il numero delle visite è dell’ordine di  $n^2$ ”. In questo modo possiamo agevolmente osservare che il numero di visite quadruplica quando le dimensioni dell’array raddoppiano, perché  $(2n)^2 = 4n^2$ .

Per indicare che il numero delle visite è dell’ordine di  $n^2$ , spesso nell’informatica teorica si usa la **notazione O-grande** (*big-Oh notation*). Il numero delle visite è  $O(n^2)$ : si tratta di una comoda abbreviazione, di cui daremo una definizione formale nella sezione Argomenti avanzati 13.1.

Per trasformare un’espressione esatta come

$$\frac{1}{2}n^2 + \frac{5}{2}n - 3$$

nella corrispondente notazione O-grande basta semplicemente individuare il termine che aumenta più rapidamente,  $n^2$ , e ignorare il suo coefficiente costante,  $1/2$ , indipendentemente dal fatto che sia grande o piccolo.

Abbiamo osservato prima che il numero effettivo delle istruzioni eseguite dal processore e l’effettiva quantità di tempo che il computer dedica a esse è all’incirca proporzionale al numero delle visite degli elementi dell’array. Forse per ciascuna visita servono una decina di istruzioni macchina (incrementi, confronti, letture e scritture nella memoria): il numero delle istruzioni macchina è quindi approssimativamente  $10 \cdot (1/2) \cdot n^2$ . Ancora una volta il coefficiente non ci interessa, per cui possiamo dire che il numero delle istruzioni macchina, e quindi il tempo necessario per l’ordinamento, è dell’ordine di  $n^2$ , ovvero  $O(n^2)$ .

Rimane il triste fatto che il raddoppio delle dimensioni dell’array quadruplica il tempo necessario per ordinarlo usando l’ordinamento per selezione. Quando la dimensione dell’array aumenta di un fattore 100, il tempo di ordinamento aumenta di un fattore 10000. Per ordinare un array con un milione di voci (per generare, ad esempio, un elenco telefonico), si impiega un tempo 10000 volte più lungo di quello che occorrerebbe per ordinare 10000 voci. Se 10000 voci si possono ordinare in circa tre quarti di secondo (come nel nostro esempio), allora l’ordinamento di un milione di voci richiede ben più di due ore. Questo è un problema: vedremo nel prossimo paragrafo come si possano migliorare in modo spettacolare le prestazioni del processo di ordinamento scegliendo un algoritmo più sofisticato.

Nell’informatica teorica si descrive il tasso di crescita di una funzione usando la notazione O-grande.

L’ordinamento per selezione è un algoritmo  $O(n^2)$ : il raddoppio della dimensione dell’insieme di dati quadruplica il tempo di elaborazione.



## Auto-valutazione

7. Se aumentate di dieci volte la dimensione dell'insieme di dati, come aumenta il tempo richiesto per ordinarlo con l'algoritmo di ordinamento per selezione?
8. Quanto deve valere  $n$  perché  $(1/2)n^2$  sia maggiore di  $(5/2)n - 3$ ?
9. Nel Paragrafo 7.3.6 sono stati presentati due algoritmi per l'eliminazione di un elemento da un array di lunghezza  $n$ . Quante visite a singoli elementi dell'array sono richieste, in media, da ciascun algoritmo?
10. Usate la notazione O-grande per descrivere il numero di visite nei problemi descritti nella domanda precedente.
11. Usando la notazione O-grande, qual è il tempo richiesto per verificare se un array è ordinato?
12. Considerate questo algoritmo di ordinamento di un array di lunghezza  $k$ . Trovate l'elemento massimo tra i primi  $k$  elementi e rimuovetelo dall'array usando il secondo algoritmo presentato nel Paragrafo 7.3.6, quindi diminuite  $k$  di un'unità e scrivete l'elemento rimosso nella posizione di indice  $k$  dell'array. Ripetete la procedura, terminando l'algoritmo quando  $k$  vale 1. Usando la notazione O-grande, qual è il tempo d'esecuzione dell'algoritmo?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R13.4, R13.6 e R13.8, al termine del capitolo.



## Argomenti avanzati 13.1

### O-grande, Omega ( $\Omega$ ) e Theta ( $\Theta$ )

In questo capitolo abbiamo usato in modo un po' approssimativo la notazione O-grande, per descrivere il comportamento della crescita di una funzione: vediamo come si può descrivere più formalmente la notazione O-grande. Sia data una funzione  $T(n)$ , che solitamente, nel nostro caso, rappresenta il tempo richiesto da un algoritmo per elaborare dati d'ingresso di dimensione  $n$ , ma potrebbe essere una funzione qualsiasi. Supponiamo, poi, che esista un'altra funzione,  $f(n)$ : solitamente viene scelta una funzione "semplice", ad esempio  $f(n) = n^k$  oppure  $f(n) = \log(n)$ , ma anche questa può essere una qualsiasi funzione. Scriviamo

$$T(n) = O(f(n))$$

se la crescita di  $T(n)$  è superiormente limitata dalla crescita di  $f(n)$ . Detto in modo più formale, deve essere vero che per ogni valore di  $n$  maggiore di una qualche soglia sia  $T(n) / f(n) \leq C$ , con  $C$  valore costante.

Se  $T(n)$  è un polinomio di grado  $k$  in  $n$ , allora si può dimostrare che  $T(n) = O(n^k)$ . Nel seguito del capitolo vedremo funzioni che sono  $O(\log(n))$  o  $O(n \log(n))$ , mentre alcuni algoritmi richiedono un tempo d'esecuzione molto maggiore. Ad esempio, un modo per ordinare una sequenza consiste nel calcolare tutte le permutazioni dei suoi elementi, fin quando non se ne trova una ordinata: tale algoritmo richiede un tempo  $O(n!)$ , non propriamente esaltante.

La Tabella 1 mostra alcune espressioni O-grande usate di frequente, in ordine crescente.

**Tabella 1**

Alcune espressioni O-grande di utilizzo frequente

| Espressione O-grande | Nome         |
|----------------------|--------------|
| $O(1)$               | Costante     |
| $O(\log(n))$         | Logaritmica  |
| $O(n)$               | Lineare      |
| $O(n \log(n))$       | Log-lineare  |
| $O(n^2)$             | Quadratica   |
| $O(n^3)$             | Cubica       |
| $O(2^n)$             | Esponenziale |
| $O(n!)$              | Fattoriale   |

Dicendo che  $T(n) = O(f(n))$  si afferma che  $T$  non cresce più velocemente di  $f$ , ma è possibile che  $T$  cresca molto più lentamente, per cui è tecnicamente corretto dire che  $T(n) = n^2 + 5n - 3$  è  $O(n^2)$  o anche  $O(n^{10})$ .

Gli informatici teorici hanno inventato ulteriori notazioni che descrivono in modo più accurato il modo in cui crescono le funzioni. L'espressione

$$T(n) = \Omega(f(n))$$

significa che  $T$  cresce almeno tanto velocemente quanto cresce  $f$  oppure, in modo più formale, che per tutti i valori di  $n$  maggiori di una certa soglia si ha  $T(n)/f(n) \geq C$ , con  $C$  costante (il simbolo  $\Omega$  è la lettera omega maiuscola dell'alfabeto greco). Ad esempio,  $T(n) = n^2 + 5n - 3$  è  $\Omega(n^2)$  o anche  $\Omega(n)$ .

L'espressione

$$T(n) = \Theta(f(n))$$

significa che  $T$  e  $f$  crescono con la stessa velocità, cioè è vero sia che  $T(n) = O(f(n))$  sia che  $T(n) = \Omega(f(n))$  (il simbolo  $\Theta$  è la lettera theta maiuscola dell'alfabeto greco).

La notazione  $\Theta$  fornisce la più precisa descrizione dell'andamento di crescita di una funzione. Ad esempio,  $T(n) = n^2 + 5n - 3$  è  $\Theta(n^2)$  ma non è  $\Theta(n)$  né  $\Theta(n^3)$ .

Queste notazioni sono molto importanti per effettuare un'analisi degli algoritmi con una certa precisione, ma è pratica comune parlare semplicemente di O-grande, pur fornendo per tale notazione la stima più precisa possibile.



## Argomenti avanzati 13.2

### Ordinamento per inserimento

L'ordinamento per inserimento (*insertion sort*) è un altro semplice algoritmo di ordinamento, nel quale si suppone che la parte iniziale

$a[0] \ a[1] \ \dots \ a[k]$

di un array sia già ordinata (quando l'algoritmo inizia il suo lavoro,  $k$  vale 0). Espandiamo questa parte iniziale ordinata inserendovi nella giusta posizione il successivo elemento dell'array,  $a[k + 1]$ . Giunti al termine dell'array, il processo di ordinamento è completato.

Ad esempio, supponiamo di iniziare con questo array:

|    |   |    |   |   |
|----|---|----|---|---|
| 11 | 9 | 16 | 5 | 7 |
|----|---|----|---|---|

La parte iniziale, di lunghezza 1, è ovviamente già ordinata. Aggiungiamo ora a tale porzione ordinata l'elemento  $a[1]$ , il cui valore è 9. Tale elemento deve essere inserito prima dell'elemento di valore 11, per cui il risultato è:

|   |    |    |   |   |
|---|----|----|---|---|
| 9 | 11 | 16 | 5 | 7 |
|---|----|----|---|---|

Procediamo aggiungendo l'elemento  $a[2]$ , il cui valore è 16: non c'è bisogno di spostarlo.

|   |    |    |   |   |
|---|----|----|---|---|
| 9 | 11 | 16 | 5 | 7 |
|---|----|----|---|---|

Ripetiamo il procedimento inserendo l'elemento  $a[3]$ , di valore 5, nella prima posizione della porzione iniziale.

|   |   |    |    |   |
|---|---|----|----|---|
| 5 | 9 | 11 | 16 | 7 |
|---|---|----|----|---|

Infine, l'elemento  $a[4]$ , di valore 7, viene inserito nella posizione corretta e l'ordinamento è completo.

La classe seguente realizza l'algoritmo di ordinamento per inserimento.

```
public class InsertionSorter
{
    /**
     * Ordina un array con l'algoritmo di ordinamento per inserimento.
     * @param a l'array da ordinare
     */
    public static void sort(int[] a)
    {
        for (int i = 1; i < a.length; i++)
        {
            int next = a[i];
            // sposta in avanti tutti gli elementi maggiori
            int j = i;
            while (j > 0 && a[j - 1] > next)
            {
                a[j] = a[j - 1];
                j--;
            }
            // inserisci l'elemento
            a[j] = next;
        }
    }
}
```

Quanto è efficiente questo algoritmo? Indichiamo con  $n$  la dimensione dell'array, per il cui ordinamento eseguiamo  $n - 1$  iterazioni. Durante la  $k$ -esima iterazione abbiamo una porzione già ordinata di  $k$  elementi, nella quale dobbiamo inserire un nuovo elemento; ciascun inserimento richiede di visitare gli elementi della porzione iniziale ordinata finché

non è stata trovata la posizione in cui inserire il nuovo elemento, dopodiché dobbiamo spostare verso posizioni di indice maggiore i rimanenti elementi della parte già ordinata. Di conseguenza, vengono visitati  $k + 1$  elementi dell'array, per cui il numero totale di visite è:

$$2 + 3 + \dots + n = \frac{n \cdot (n + 1)}{2} - 1$$

L'ordinamento per inserimento  
è un algoritmo  $O(n^2)$ .

Possiamo, quindi, concludere che l'ordinamento per inserimento è un algoritmo  $O(n^2)$ , con un'efficienza dello stesso ordine di quella dell'ordinamento per selezione.

L'ordinamento per inserimento ha, però, un'interessante caratteristica: le sue prestazioni sono  $O(n)$  se l'array è già ordinato (si veda l'Esercizio R13.19). Tale proprietà è utile in molti casi pratici, perché capita spesso di dover ordinare insiemi di dati già parzialmente ordinati.

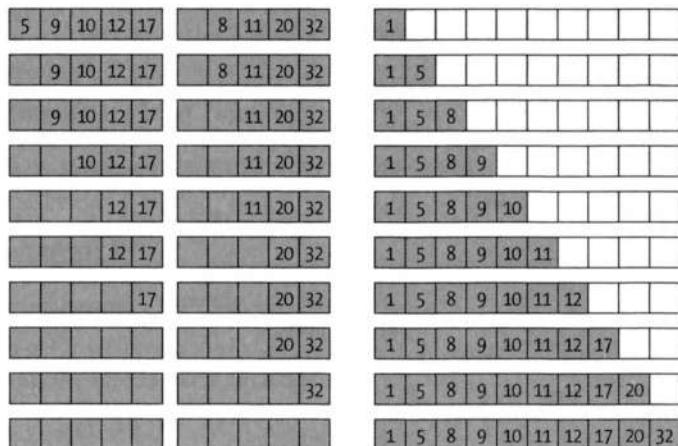
## 13.4 Ordinamento per fusione (MergeSort)

In questo paragrafo studierete l'algoritmo di ordinamento per fusione, che è molto più efficiente dell'ordinamento per selezione, anche se l'idea su cui si basa è molto semplice.

Supponiamo di avere un array di 10 numeri interi. Proviamo per una volta a essere ottimisti e speriamo che la prima metà dell'array sia già perfettamente ordinata e che lo sia anche la seconda metà, come in questo caso:

|   |   |    |    |    |  |   |   |    |    |    |
|---|---|----|----|----|--|---|---|----|----|----|
| 5 | 9 | 10 | 12 | 17 |  | 1 | 8 | 11 | 20 | 32 |
|---|---|----|----|----|--|---|---|----|----|----|

A questo punto è facile *fondere* i due array ordinati in un solo array ordinato, semplicemente prelevando un nuovo elemento dal primo o dal secondo sotto-array, scegliendo ogni volta l'elemento più piccolo:



In effetti, è probabile che abbiate eseguito proprio questo tipo di fusione quando vi siete trovati con un amico a dover ordinare una pila di fogli. Avete spartito la pila in due mucchi, ciascuno di voi ha ordinato la propria metà e, poi, avete fuso insieme i vostri risultati.

L'algoritmo di ordinamento per fusione ordina un array dividendolo a metà, ordinando ricorsivamente ciascuna delle due parti e fondendo, poi, le due metà ordinate.

Tutto questo sarà anche divertente, ma non sembra che possa essere una soluzione del problema per il computer, che si trova a dover ancora ordinare la prima e la seconda metà dell'array: non può certo chiedere a qualche amico di dargli una mano. Scopriamo, però, che, se il computer continua a suddividere l'array in array sempre più piccoli, ordinando ciascuna metà e fondendole poi insieme, i passi che deve eseguire sono assai meno numerosi di quelli richiesti dall'ordinamento per selezione.

Proviamo a scrivere una classe, MergeSorter, che implementi questa idea. Quando il suo metodo sort ordina un array, crea due array, ciascuno avente dimensione pari alla metà dell'array originario, e li ordina ricorsivamente. Quindi, fonde insieme i due array ordinati:

```
public static void sort(int[] a)
{
    if (a.length <= 1) { return; }
    int[] first = new int[a.length / 2];
    int[] second = new int[a.length - first.length];
    // copia in first la prima metà e in second la seconda
    . .
    sort(first);
    sort(second);
    merge(first, second, a);
}
```

Il metodo merge è noioso ma abbastanza semplice: lo troverete nel codice che segue.

### File MergeSorter.java

```
1 /**
2      Il metodo sort di questa classe ordina un array
3      usando l'algoritmo di ordinamento per fusione.
4 */
5 public class MergeSorter
6 {
7     /**
8      Ordina un array usando l'algoritmo di ordinamento per fusione.
9      @param a l'array da ordinare
10 */
11 public static void sort(int[] a)
12 {
13     if (a.length <= 1) { return; }
14     int[] first = new int[a.length / 2];
15     int[] second = new int[a.length - first.length];
16     // copia in first la prima metà e in second la seconda
17     for (int i = 0; i < first.length; i++)
18     {
19         first[i] = a[i];
20     }
21     for (int i = 0; i < second.length; i++)
22     {
23         second[i] = a[first.length + i];
24     }
25     sort(first);
26     sort(second);
```

```

27     merge(first, second, a);
28 }
29
30 /**
31  * Fonde in un array due array ordinati.
32  * @param first il primo array ordinato
33  * @param second il secondo array ordinato
34  * @param a l'array in cui viene memorizzato il risultato della fusione
35 */
36 private static void merge(int[] first, int[] second, int[] a)
37 {
38     int iFirst = 0; // il prossimo elemento da considerare nel primo array
39     int iSecond = 0; // il prossimo elemento da considerare nel secondo array
40     int j = 0; // la prossima posizione libera nell'array a
41
42     // finché né iFirst né iSecond oltrepassano la fine
43     // del relativo array, sposta in a l'elemento minore
44     while (iFirst < first.length && iSecond < second.length)
45     {
46         if (first[iFirst] < second[iSecond])
47         {
48             a[j] = first[iFirst];
49             iFirst++;
50         }
51         else
52         {
53             a[j] = second[iSecond];
54             iSecond++;
55         }
56         j++;
57     }
58
59     // notate che soltanto uno dei due cicli seguenti viene eseguito:
60     // copia in a tutti i valori rimasti nel primo array
61     while (iFirst < first.length)
62     {
63         a[j] = first[iFirst];
64         iFirst++; j++;
65     }
66     // copia in a tutti i valori rimasti nel secondo array
67     while (iSecond < second.length)
68     {
69         a[j] = second[iSecond];
70         iSecond++; j++;
71     }
72 }
73 }
```

### File MergeSortDemo.java

```

1 import java.util.Arrays;
2
3 /**
4  * Questo programma applica l'algoritmo di ordinamento
5  * per fusione a un array riempito con numeri casuali.

```

```

6  */
7 public class MergeSortDemo
8 {
9     public static void main(String[] args)
10    {
11        int[] a = ArrayUtil.randomIntArray(20, 100);
12        System.out.println(Arrays.toString(a));
13
14        MergeSorter.sort(a);
15
16        System.out.println(Arrays.toString(a));
17    }
18 }

```

### Esempio di esecuzione del programma

```
[8, 81, 48, 53, 46, 70, 98, 42, 27, 76, 33, 24, 2, 76, 62, 89, 90, 5, 13, 21]
[2, 5, 8, 13, 21, 24, 27, 33, 42, 46, 48, 53, 62, 70, 76, 81, 89, 90, 98]
```



### Auto-valutazione

13. Perché soltanto uno dei due cicli `while` presenti al termine del metodo `merge` fa qualcosa?
14. Eseguite manualmente l'algoritmo di ordinamento per fusione sull'array 8 7 6 5 4 3 2 1.
15. L'algoritmo di ordinamento per fusione elabora un array analizzando ricorsivamente le sue due metà. Descrivete un algoritmo ricorsivo analogo che calcoli la somma di tutti gli elementi presenti in un array.

### Per far pratica

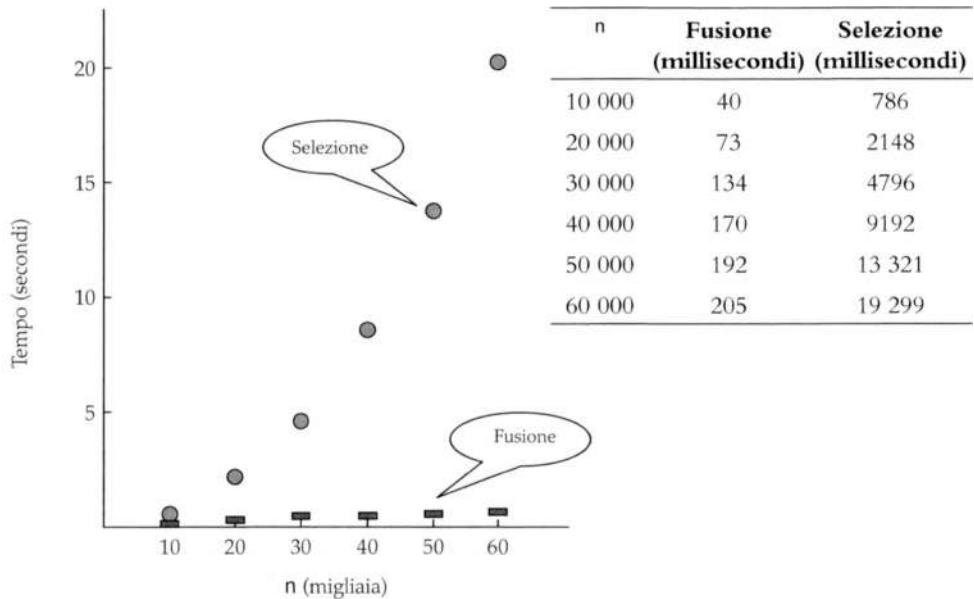
A questo punto si consiglia di svolgere gli esercizi R13.13, E13.4 e E13.14, al termine del capitolo.

## 13.5 Analisi dell'algoritmo di ordinamento per fusione

L'algoritmo di ordinamento per fusione sembra molto più complesso dell'algoritmo di ordinamento per selezione e ci si immagina che possa impiegare molto più tempo per eseguire tutte queste ripetute suddivisioni. Invece, i tempi che si rilevano con l'ordinamento per fusione sono decisamente migliori di quelli relativi all'ordinamento per selezione.

La Figura 2 mostra una tabella e un grafico che mettono a confronto i due insiemi di misurazioni delle prestazioni: come si può vedere, con l'ordinamento per fusione il miglioramento è strepitoso. Per capirne il motivo, proviamo a stimare il numero di visite agli elementi dell'array che sono necessarie per ordinare un array mediante l'algoritmo di ordinamento per fusione. Affrontiamo per prima cosa il processo di fusione che si effettua dopo che la prima e la seconda metà sono state ordinate.

**Figura 2**  
Tempo di esecuzione dell'ordinamento per fusione (in basso) e per selezione (in alto).



Ciascun passo del processo di fusione aggiunge all'array a un elemento, che può venire da first o da second: nella maggior parte dei casi bisogna confrontare gli elementi iniziali delle due metà per stabilire quale prendere. Conteggiamo questa operazione come 3 visite per ogni elemento (una per a e una ciascuna per first e second), ovvero  $3n$  visite in totale, essendo  $n$  la lunghezza dell'array a. Inoltre, all'inizio dobbiamo copiare tutti gli elementi dall'array a negli array first e second, rendendo necessarie altre  $2n$  visite, per un totale di  $5n$ .

Se chiamiamo  $T(n)$  il numero di visite necessarie per ordinare un array di  $n$  elementi mediante il processo di ordinamento per fusione, otteniamo:

$$T(n) = T(n/2) + T(n/2) + 5n$$

perché l'ordinamento di ciascuna metà richiede  $T(n/2)$  visite. In realtà, se  $n$  non è pari abbiamo un sotto-array di dimensione  $(n-1)/2$  e un altro di dimensione  $(n+1)/2$ : sebbene questo dettaglio si dimostrerà irrilevante ai fini del risultato del calcolo, supporremo per ora che  $n$  sia una potenza di 2, diciamo  $n = 2^m$ . In questo modo tutti i sotto-array si possono dividere in due parti uguali.

Sfortunatamente, la formula

$$T(n) = 2T(n/2) + 5n$$

non ci fornisce con chiarezza la relazione esistente fra  $n$  e  $T(n)$ . Per individuare la natura di tale relazione, valutiamo  $T(n/2)$  usando la stessa formula, ottenendo:

$$T(n/2) = 2T(n/4) + 5n/2$$

Quindi

Facciamolo di nuovo:

$$T(n/4) = 2T(n/8) + 5n/4$$

quindi

$$T(n) = 2 \times 2 \times 2T(n/8) + 5n + 5n + 5n$$

Generalizzando da 2, 4, 8 a potenze arbitrarie di 2:

$$T(n) = 2^k T(n/2^k) + 5nk$$

Ricordiamo che abbiamo assunto  $n = 2^m$ ; di conseguenza, per  $k = m$ ,

$$\begin{aligned} T(n) &= 2^m T(n/2^m) + 5nm \\ &= nT(1) + 5nm \\ &= n + 5n \log_2 n \end{aligned}$$

perché  $n = 2^m$  implica  $m = \log_2(n)$ .

Per capire come cresce la funzione eliminiamo il termine di grado inferiore,  $n$ , rimanendo con  $5n \log_2(n)$ . Eliminiamo anche il fattore costante, 5. Di solito si trascura anche la base del logaritmo, perché tutti i logaritmi sono tra loro correlati tramite un fattore costante. Per esempio:

$$\log_2(x) = \log_{10}(x) / \log_{10}(2) \approx 3.32193 \times \log_{10}(x)$$

Di conseguenza, possiamo dire che l'ordinamento per fusione è un algoritmo  $O(n \log(n))$ .

L'ordinamento per fusione  
è un algoritmo  $O(n \log(n))$ .  
La funzione  $n \log(n)$  cresce molto più  
lentamente di  $n^2$ .

L'algoritmo di ordinamento per fusione, con prestazioni  $O(n \log(n))$ , è migliore dell'algoritmo di ordinamento per selezione, avente prestazioni  $O(n^2)$ ? Ci potete scommettere. Ricordate che, con l'algoritmo  $O(n^2)$ , l'ordinamento di un milione di valori richiede  $100^2 = 10000$  volte il tempo che è necessario per ordinare 10000 valori. Con l'algoritmo  $O(n \log(n))$ , il rapporto è:

$$\frac{1000\,000 \log 1000\,000}{10\,000 \log 10\,000} = 100 \left( \frac{6}{4} \right) = 150$$

Supponiamo per un momento che, per ordinare un array di 10000 numeri, l'ordinamento per fusione impieghi lo stesso tempo che impiega l'ordinamento per selezione, cioè tre quarti di secondo sulla nostra macchina di prova (in realtà, è molto più veloce). Allora per ordinare un milione di numeri interi impiegherebbe circa  $0.75 \times 150$  secondi, ovvero meno di 2 minuti: confrontate questo tempo con quello richiesto dall'ordinamento per selezione, che ci metterebbe più di 2 ore per eseguire lo stesso compito. Come potete vedere, anche se vi servono alcune ore per imparare un algoritmo migliore, si tratta di tempo ben speso.

In questo capitolo abbiamo appena cominciato a scalfire la superficie di questo interessante argomento. Esistono molti algoritmi di ordinamento, alcuni dei quali hanno prestazioni persino migliori di quelle dell'ordinamento per fusione, e la cui analisi può essere una bella sfida. Se state seguendo un corso di studi in informatica, rivedrete questi importanti argomenti in un corso successivo.



## Auto-valutazione

16. Sulla base dei dati temporali presentati nella tabella all'inizio di questo paragrafo per l'algoritmo di ordinamento per fusione, quanto tempo occorre per ordinare un array di 100000 valori?
17. Se raddoppiate la dimensione di un array, come aumenta il tempo richiesto per ordinare il nuovo array usando l'algoritmo di ordinamento per fusione?

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi R13.7, R13.16 e R13.18, al termine del capitolo.



## Argomenti avanzati 13.3

### L'algoritmo Quicksort

Quicksort è un algoritmo di frequente utilizzo, che ha il vantaggio, rispetto all'ordinamento per fusione, di non aver bisogno di array temporanei per ordinare e fondere i risultati parziali.

L'algoritmo quicksort, come l'ordinamento per fusione, si basa sulla strategia di “dividere per vincere” (*divide and conquer*). Per ordinare la porzione  $a[from] \dots a[to]$  dell'array  $a$ , si dispongono dapprima gli elementi in modo che nessuno di quelli presenti nella parte  $a[from] \dots a[p]$  sia maggiore di elementi dell'altra parte,  $a[p + 1] \dots a[to]$ . Questo passo viene detto suddivisione o *partizionamento* della porzione.

Ad esempio, supponete di iniziare con questa porzione

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 5 | 3 | 2 | 6 | 4 | 1 | 3 | 7 |
|---|---|---|---|---|---|---|---|

Ecco una possibile suddivisione della porzione: notate che le due parti non sono ancora state ordinate.

|   |   |   |   |   |  |   |   |   |
|---|---|---|---|---|--|---|---|---|
| 3 | 3 | 2 | 1 | 4 |  | 6 | 5 | 7 |
|---|---|---|---|---|--|---|---|---|

Vedrete più avanti come ottenere tale suddivisione. Nel prossimo passo, ordinate ciascuna parte applicando ricorsivamente il medesimo algoritmo: questa azione ordina l'intera porzione originaria, perché il massimo elemento presente nella prima parte è al più uguale al minimo elemento presente nella seconda parte.

|   |   |   |   |   |  |   |   |   |
|---|---|---|---|---|--|---|---|---|
| 1 | 2 | 3 | 3 | 4 |  | 5 | 6 | 7 |
|---|---|---|---|---|--|---|---|---|

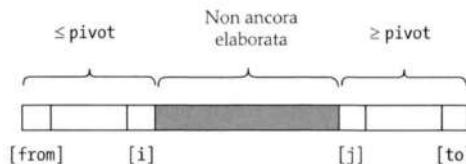
Ecco un'implementazione ricorsiva di quicksort:

```
public static void sort(int[] a, int from, int to)
{
    if (from >= to) { return; }
    int p = partition(a, from, to);
    sort(a, from, p);
    sort(a, p + 1, to);
}
```

Torniamo al problema di come suddividere in due parti una porzione di array. Scegliete un elemento e chiamatelo *pivot* (cardine). Esistono diverse varianti dell'algoritmo quicksort: nella più semplice, sceglierete come pivot il primo elemento della porzione, cioè  $a[from]$ .

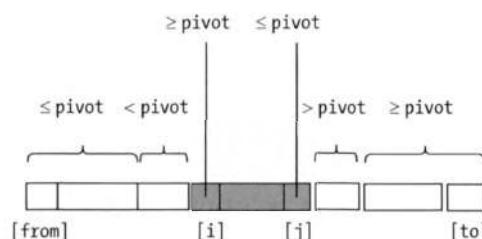
Ora create due regioni:  $a[from] \dots a[i]$ , contenente valori non maggiori del pivot, e  $a[j] \dots a[to]$ , contenente valori non minori del pivot. La regione  $a[i+1] \dots a[j-1]$  contiene valori che non sono ancora stati analizzati. All'inizio le zone di sinistra e di destra sono vuote, cioè  $i = from - 1$  e  $j = to + 1$ .

Suddivisione  
di una porzione di array  
da ordinare



A questo punto, incrementate  $i$  finché  $a[i] < \text{pivot}$  e decrementate  $j$  finché  $a[j] > \text{pivot}$ . La figura mostra  $i$  e  $j$  quando il procedimento si arresta.

Estensione delle due parti,  
sinistra e destra



Ora scambiate tra loro i valori che si trovano nelle posizioni  $i$  e  $j$ , estendendo così entrambe le zone, poi proseguite con la procedura precedente finché  $i < j$ . Ecco il codice per il metodo *partition*:

```
private static int partition(int[] a, int from, int to)
{
    int pivot = a[from];
    int i = from - 1;
    int j = to + 1;
    while (i < j)
    {
```

```

    i++; while (a[i] < pivot) { i++;}
    j--; while (a[j] > pivot) { j--; }
    if (i < j) { ArrayUtil.swap(i, j); }
}
return j;
}

```

In media, l'algoritmo quicksort ha prestazioni  $O(n \log(n))$ . C'è un solo aspetto sfortunato nell'algoritmo quicksort: il suo comportamento nel *caso peggiore* è  $O(n^2)$ . Inoltre, se come pivot viene scelto il primo elemento della regione, il comportamento di caso peggiore si ha quando l'insieme è già ordinato: una situazione, in pratica, piuttosto frequente. Scegliendo il pivot con maggior attenzione possiamo rendere estremamente improbabile l'evenienza del caso peggiore: gli algoritmi quicksort "messi a punto" in tal modo sono usati molto frequentemente, perché le loro prestazioni sono generalmente eccellenti. Ad esempio, il metodo `sort` della classe `Arrays` usa un algoritmo quicksort.

Un altro miglioramento che viene solitamente messo in atto prevede di passare all'utilizzo dell'ordinamento per inserimento quando l'array è di piccole dimensioni, perché il numero totale di operazioni richieste dall'ordinamento per inserimento è, in tali casi, inferiore. La libreria Java usa questo accorgimento quando la lunghezza dell'array è inferiore a sette.

## 13.6 Effettuare ricerche

Capita molto frequentemente di dover cercare un elemento in un array e, come visto nel problema dell'ordinamento, la scelta dell'algoritmo migliore può fare veramente la differenza.

### 13.6.1 Ricerca lineare

Immaginate di voler trovare il numero di telefono di un vostro amico. Cercate il suo nome nell'elenco telefonico e, ovviamente, lo trovate rapidamente, perché l'elenco telefonico è ordinato alfabeticamente. Pensate, ora, di avere un numero di telefono e di voler sapere a chi appartiene: potreste, naturalmente, chiamare quel numero, ma supponiamo che nessuno risponda alla chiamata; in alternativa, potreste scorrere l'elenco telefonico, un numero dopo l'altro, fino a quando trovate quello che vi interessa. Questo comporterebbe, ovviamente, un'enorme quantità di lavoro: dovreste essere davvero disperati per imbarcarvi in un'impresa del genere.

Questo ipotetico esperimento fa capire la differenza fra la ricerca effettuata in un insieme di dati ordinati e quella che opera con dati non ordinati: i prossimi due paragrafi esamineranno questa differenza in modo più formale.

Se volete trovare un numero all'interno di una sequenza di valori che si presentano in un ordine arbitrario, non potete fare nulla per accelerare la ricerca: dovete semplicemente scorrere tutti gli elementi, esaminandoli uno a uno, fino a quando trovate una corrispondenza con l'elemento cercato oppure arrivate in fondo all'elenco. Questa si chiama **ricerca lineare** (*linear search*) o **ricerca sequenziale** (*sequential search*).

La ricerca lineare esamina tutti i valori di un array fino a trovare una corrispondenza con quanto cercato o la fine dell'array.

La ricerca lineare trova un valore in un array eseguendo un numero di passi  $O(n)$ .

Quanto tempo richiede una ricerca lineare? Nell'ipotesi che l'elemento  $v$  sia presente nell'array  $a$  di lunghezza  $n$ , la ricerca richiede in media la visita di  $n/2$  elementi. Se l'elemento non è presente nell'array bisogna visitare tutti gli elementi per verificarne l'assenza. In ogni caso, la ricerca lineare è un algoritmo  $O(n)$ .

Ecco una classe che esegue la ricerca lineare in un array  $a$  di numeri interi: il metodo `search` restituisce l'indice della prima corrispondenza trovata, oppure  $-1$  se il valore non è presente in  $a$ .

### File LinearSearcher.java

```

1  /**
2   * Una classe che esegue ricerche lineari in un array.
3  */
4  public class LinearSearcher
5  {
6      /**
7       * Cerca un valore in un array usando l'algoritmo
8       * di ricerca lineare.
9       * @param a l'array in cui cercare
10      * @param value il valore da cercare
11      * @return l'indice in cui si trova il valore, oppure -1
12      *         se il valore non è presente nell'array
13     */
14    public static int search(int[] a, int value)
15    {
16        for (int i = 0; i < a.length; i++)
17        {
18            if (a[i] == value) { return i; }
19        }
20        return -1;
21    }
22 }
```

### File LinearSearchDemo.java

```

1 import java.util.Arrays;
2 import java.util.Scanner;
3
4 /**
5  * Questo programma utilizza l'algoritmo di ricerca lineare.
6  */
7 public class LinearSearchDemo
8 {
9     public static void main(String[] args)
10    {
11        int[] a = ArrayUtil.randomIntArray(20, 100);
12        System.out.println(Arrays.toString(a));
13        Scanner in = new Scanner(System.in);
14
15        boolean done = false;
16        while (!done)
17        {
18            System.out.print("Enter number to search for, -1 to quit: ");
```

```

19     int n = in.nextInt();
20     if (n == -1)
21     {
22         done = true;
23     }
24     else
25     {
26         int pos = LinearSearcher.search(a, n);
27         System.out.println("Found in position " + pos);
28     }
29 }
30 }
31 }
```

### Esempio di esecuzione del programma

```
[46, 99, 45, 57, 64, 95, 81, 69, 11, 97, 6, 85, 61, 88, 29, 65, 83, 88, 45, 88]
Enter number to search for, -1 to quit: 11
Found in position 8
Enter number to search for, -1 to quit: 12
Found in position -1
Enter number to search for, -1 to quit: -1
```

### 13.6.2 Ricerca binaria

Cerchiamo ora un elemento all'interno di una sequenza di dati che sia stata precedentemente ordinata. Naturalmente potremmo ancora eseguire una ricerca lineare, ma possiamo far di meglio, come si vedrà.

Considerando questo array ordinato, a:

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
| 1   | 4   | 5   | 8   | 9   | 12  | 17  | 20  | 24  | 32  |

vorremmo sapere se il valore 15 è presente al suo interno. Restringiamo la nostra ricerca, chiedendoci se il valore si trova nella prima o nella seconda metà dell'array. L'ultimo valore nella prima metà dell'insieme, a[4], è 9: è più piccolo del valore che stiamo cercando, quindi dovremo cercare nella seconda metà dell'insieme, cioè nella porzione qui evidenziata in grigio:

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
| 1   | 4   | 5   | 8   | 9   | 12  | 17  | 20  | 24  | 32  |

L'elemento centrale di questa sequenza è 20, quindi il valore che cerchiamo, se c'è, deve trovarsi nella sotto-sequenza qui nuovamente evidenziata in grigio:

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
| 1   | 4   | 5   | 8   | 9   | 12  | 17  | 20  | 24  | 32  |

L'ultimo valore della prima metà di questa sequenza molto breve è 12, che è minore del valore che stiamo cercando, per cui dobbiamo cercare nella seconda metà:

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
| 1   | 4   | 5   | 8   | 9   | 12  | 17  | 20  | 24  | 32  |

È banale constatare che non abbiamo trovato il numero cercato, perché  $15 \neq 17$ . Se volessimo inserire 15 nella sequenza, avremmo dovuto inserirlo appena prima di  $a[6]$ .

La ricerca binaria cerca un valore in un array ordinato determinando se si può trovare nella prima o nella seconda metà dell'array, ripetendo poi la ricerca in una delle due metà.

Questo processo di ricerca si chiama **ricerca binaria** (*binary search*) o ricerca per bisezione perché a ogni passo dimezziamo la dimensione della zona da esplorare: tale dimezzamento funziona soltanto perché sappiamo che la sequenza dei valori è ordinata.

La classe seguente realizza la ricerca binaria all'interno di un array ordinato di numeri interi. Il metodo `search` restituisce la posizione dell'elemento cercato se la ricerca ha successo, oppure  $-1$  se il valore cercato non è presente in  $a$ . L'algoritmo di ricerca binaria viene qui mostrato nella sua versione ricorsiva.

### File BinarySearcher.java

```

1  /**
2   * Una classe per eseguire ricerche binarie in un array.
3  */
4  public class BinarySearcher
5  {
6      /**
7       * Cerca un valore in un array ordinato
8       * utilizzando l'algoritmo di ricerca binaria.
9       * @param a l'array in cui cercare
10      * @param low il primo indice della zona di ricerca
11      * @param high l'ultimo indice della zona di ricerca
12      * @param value il valore da cercare
13      * @return l'indice in cui si trova il valore cercato, oppure -1
14      *         se il valore non è presente nell'array
15     */
16    public static int search(int[] a, int low, int high, int value)
17    {
18        if (low <= high)
19        {
20            int mid = (low + high) / 2;
21
22            if (a[mid] == value)
23            {
24                return mid;
25            }
26            else if (a[mid] < value)
27            {
28                return search(a, mid + 1, high, value);
29            }
30            else
31            {
32                return search(a, low, mid - 1, value);
33            }
34        }
35    }

```

```

35     else
36     {
37         return -1;
38     }
39 }
40 }
```

Proviamo ora a stabilire quante visite di elementi dell'array sono necessarie per portare a termine una ricerca binaria. Possiamo usare la stessa tecnica che abbiamo adottato per analizzare l'ordinamento per fusione e osservare che, poiché esaminiamo l'elemento centrale, che conta come una sola visita, e poi esploriamo il sotto-array di sinistra oppure quello di destra, possiamo scrivere:

$$T(n) = T(n/2) + 1$$

Utilizzando la stessa equazione, si ha:

$$T(n/2) = T(n/4) + 1$$

Inserendo questo risultato nell'equazione originale, otteniamo:

$$T(n) = T(n/4) + 2$$

e, generalizzando, si ottiene:

$$T(n) = T(n/2^k) + k$$

Come nell'analisi dell'ordinamento per fusione, facciamo l'ipotesi semplificativa che  $n$  sia una potenza di 2, cioè  $n = 2^m$ , dove  $m$  è  $\log_2(n)$ . Otteniamo, quindi

$$T(n) = 1 + \log_2(n)$$

Di conseguenza, la ricerca binaria è un algoritmo  $O(\log(n))$ .

**La ricerca binaria trova la posizione di un valore in un array eseguendo un numero di passi  $O(\log(n))$ .**

Questo risultato ha senso anche dal punto di vista intuitivo. Supponiamo che  $n$  valga 100: dopo ciascuna ricerca la dimensione dell'intervallo da esplorare viene divisa a metà, riducendosi via via a: 50, 25, 12, 6, 3 e 1. Dopo sette confronti abbiamo finito: questo coincide con la nostra formula, dal momento che  $\log_2(100) \approx 6.64386$  e, in effetti, la più piccola potenza di 2 con esponente intero il cui valore sia maggiore di 100 è  $2^7 = 128$ .

Dal momento che la ricerca binaria è tanto più veloce della ricerca lineare, vale la pena di ordinare un array non ordinato, per poi ricorrere alla ricerca binaria? Dipende. Se nell'array si effettua una sola ricerca, allora è più efficiente sostenere solo il costo  $O(n)$  di una ricerca lineare invece del costo  $O(n \log(n))$  di un ordinamento, seguito da quello  $O(\log(n))$  di una ricerca binaria. Se, però, sullo stesso array si devono eseguire molte ricerche, allora vale davvero la pena di eseguire prima l'ordinamento.

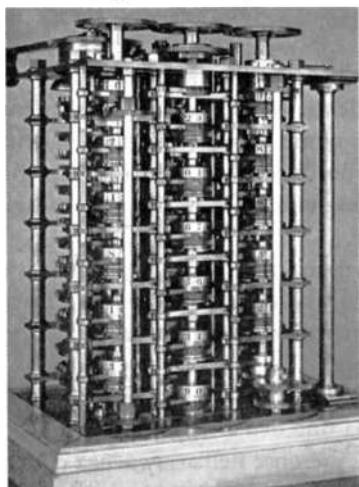


## Computer e società 13.1

### Il primo programmatore

Prima che esistessero le calcolatrici tascabili e i personal computer, navigatori e ingegneri usavano addizionatrici meccaniche, regoli calcolatori e tavole di logaritmi e di funzioni trigonometriche per accelerare i calcoli. Sfortunatamente, le tavole, i cui valori dovevano essere calcolati a mano, erano notoriamente imprecise. Il matematico Charles Babbage (1791-1871) ebbe l'intuizione che, qualora fosse stato possibile costruire una macchina che producesse automaticamente le tavole stampate, si sarebbero evitati sia gli errori di calcolo sia quelli di composizione tipografica. Babbage si dedicò al progetto di una tale macchina, che chiamò *Difference Engine*, perché utilizzava differenze consecutive per calcolare funzioni polinomiali. Per esempio, considerate la funzione  $f(x) = x^3$ . Scrivete i valori per  $f(1), f(2), f(3)$  e così di seguito, poi scrivete, più a destra, le differenze fra valori consecutivi:

Difference Engine di Babbage



|     |    |
|-----|----|
| 1   | 7  |
| 8   | 19 |
| 27  | 37 |
| 64  | 61 |
| 125 | 91 |
| 216 |    |

Ripetete il processo, scrivendo nella terza colonna le differenze fra valori consecutivi della seconda colonna, e poi ripetetelo un'altra volta:

|     |    |   |
|-----|----|---|
| 1   | 7  |   |
| 8   | 12 | 6 |
| 27  | 18 | 6 |
| 64  | 24 | 6 |
| 125 | 30 |   |
| 216 | 91 |   |

Ora le differenze ottenute sono costanti. Potete ritrovare i valori della funzione mediante una sequenza di addizioni: dovete conoscere la differenza costante e i valori che si trovano sul bordo superiore dello schema. Potete provare: scrivete su un foglio di carta i numeri evidenziati, riempiendo le posizioni rimanenti con il risultato dell'addizione dei numeri che si trovano sopra e in alto a destra rispetto al numero che si cerca.

Questo metodo era molto attraente, perché le macchine addizionatrici meccaniche già si conoscevano da tempo: erano costituite da ruote dentate, con dieci denti per ruota a rappresentare le cifre e opportuni meccanismi

per gestire il riporto da una cifra alla successiva. Al contrario, le macchine moltiplicatrici meccaniche erano fragili e poco affidabili. Babbage costruì un prototipo del *Difference Engine* che ebbe successo e, con denaro suo e alcuni fondi messi a disposizione dal governo, passò a produrre la macchina per stampare le tavole. Tuttavia, per problemi di finanziamento e per le difficoltà che si incontrarono per costruire la macchina con la precisione meccanica che era necessaria, non venne mai portata a termine.

Mentre stava lavorando al *Difference Engine*, Babbage concepì un'idea molto più grandiosa, che chiamò *Analytical Engine*. Il *Difference Engine* era stato concepito per eseguire un insieme limitato di calcoli e non era più avanzato di una calcolatrice tascabile dei nostri giorni, ma Babbage si rese conto che una macchina del genere avrebbe potuto essere resa *programmabile*, immagazzinando programmi insieme ai dati: la memoria interna dell'*Analytical Engine* doveva essere costituita da 1000 registri, ciascuno con 50 cifre decimali; programmi e costanti dovevano essere memorizzati su schede perforate, una tecnica che all'epoca era molto diffusa nei telai per tessere stoffe decorative.

Ada Augusta, contessa di Lovelace (1815-1852), unica figlia di Lord Byron, fu amica e finanziatrice di Charles Babbage e fu una delle prime persone a capire il potenziale di una macchina del genere, non soltanto per calcolare tavole matematiche, ma per elaborare dati che non fossero numeri: da molti viene considerata il primo programmatore del mondo.



## Auto-valutazione

18. Immaginate di dover cercare un numero telefonico in un insieme di un milione di dati. Quanti pensate di doverne esaminare, mediamente, per trovare il numero?
19. Perché nel metodo search non si può usare un ciclo generalizzato come `for (int element : a)?`
20. Immaginate di dover cercare un valore in un array ordinato avente un milione elementi. Usando l'algoritmo di ricerca binaria, quanti elementi pensate di dover esaminare, mediamente, per trovare il valore che cercate?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R13.14, E13.13 e E13.15, al termine del capitolo.

## 13.7 Problem Solving: stima del tempo di esecuzione di un algoritmo

In questo capitolo avete visto come si possa stimare il tempo di esecuzione degli algoritmi di ordinamento. Inoltre, come avrete certamente notato, saper distinguere un algoritmo avente prestazioni  $O(n \log(n))$  da uno  $O(n^2)$  ha grandi implicazioni pratiche. Analogamente, è molto importante saper stimare le prestazioni temporali di altri algoritmi e in questo paragrafo ci eserciteremo stimando le prestazioni di alcuni algoritmi che elaborano array.

### 13.7.1 Algoritmi lineari

Iniziamo con un esempio semplice, un algoritmo che conta il numero di elementi che hanno un determinato valore:

```
int count = 0;
for (int i = 0; i < a.length; i++)
{
    if (a[i] == value) { count++; }
```

Qual è il tempo di esecuzione in funzione di  $n$ , la lunghezza dell'array  $a$ ?

Partiamo da un'analisi dello schema di visita agli elementi dell'array. Ogni elemento viene visitato esattamente una volta e, come ausilio alla comprensione di questo schema, immaginate l'array come se fosse una serie di lampadine, disposte in fila (si veda pagina seguente). Nel momento in cui l'elemento  $i$ -esimo viene visitato, accendiamo la  $i$ -esima lampadina.

Ora pensiamo a ciò che avviene durante ciascuna visita e ci chiediamo: per una visita, è richiesto un numero fisso di azioni, indipendente da  $n$ ? In questo caso specifico, è così. Servono poche azioni, perché bisogna: leggere l'elemento, confrontarlo con il valore cercato e, quando necessario, incrementare un contatore.

Quindi, il tempo di esecuzione è pari a  $n$  volte un tempo costante, per cui è  $O(n)$ .

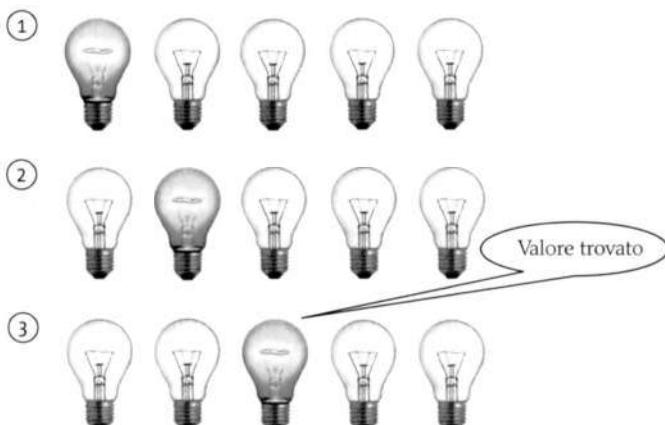
Cosa succede se l'algoritmo non arriva sempre alla fine dell'array? Supponiamo, ad esempio, di voler verificare se un determinato valore è presente nell'array, senza contare quante volte vi ricorre:

Un ciclo che esegue  $n$  iterazioni, ciascuna delle quali è costituita da un numero fisso di azioni, richiede un tempo  $O(n)$ .



```
boolean found = false;  
for (int i = 0; !found && i < a.length; i++)  
{  
    if (a[i] == value) { found = true; }  
}
```

In questo caso il ciclo può arrestarsi prima di aver esaminato l'intero array:



Anche questo è un algoritmo  $O(n)$ ? Si, perché in alcuni casi la corrispondenza potrà essere individuata alla fine dell'array. Inoltre, nel caso in cui l'elemento non sia presente nell'array, bisogna visitare tutti i suoi elementi.

### 13.7.2 Algoritmi quadratici

Passiamo ora a un caso più interessante. Cosa succede se, in corrispondenza di ogni visita, le azioni svolte sono “molte”? Ecco un esempio: vogliamo cercare l'elemento più frequente in un array.

Supponiamo che l'array sia questo:

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 8 | 7 | 5 | 7 | 7 | 5 | 4 |
|---|---|---|---|---|---|---|

Osservando i valori, è ovvio notare che quello più frequente è 7, ma se l'array avesse migliaia di valori?

|   |   |   |   |   |   |   |   |   |   |   |   |    |   |   |   |   |     |   |    |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|----|---|---|---|---|-----|---|----|---|---|---|---|---|
| 8 | 7 | 5 | 7 | 7 | 5 | 4 | 1 | 3 | 2 | 4 | 9 | 12 | 3 | 2 | 5 | { | ... | } | 11 | 9 | 2 | 3 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|---|----|---|---|---|---|-----|---|----|---|---|---|---|---|

Potremmo contare quante volte ricorre il valore 8, per poi passare a contare quante volte ricorre il valore 7, e così via. Ad esempio, nel primo array il numero 8 compare una volta e il numero 7 compare tre volte. Ma dove memorizziamo questi conteggi? Inseriamoli in un secondo array, counts, avente la stessa lunghezza del primo.

|         |   |   |   |   |   |   |   |
|---------|---|---|---|---|---|---|---|
| values: | 8 | 7 | 5 | 7 | 7 | 5 | 4 |
| counts: | 1 | 3 | 2 | 3 | 3 | 2 | 1 |

Poi, cerchiamo il valore massimo dell'array che contiene i conteggi, cioè counts. Tale valore massimo è 3, dopodiché cerchiamo la prima posizione nell'array counts in cui compare il numero 3 e identifichiamo il valore che si trova nella medesima posizione nell'array a: quello è il valore più frequente in a, in questo caso 7.

Cerchiamo, per prima cosa, di valutare il tempo necessario per calcolare i conteggi.

```
for (int i = 0; i < a.length; i++)
{
    counts[i] = . . . // conta quante volte a[i] compare in a
}
```

Un ciclo che esegue  $n$  iterazioni, ciascuna delle quali richiede un tempo  $O(n)$ , è  $O(n^2)$ .

Anche in questo caso visitiamo ciascun elemento una sola volta, ma in corrispondenza di ogni visita il lavoro svolto è decisamente maggiore, perché, come avete visto nel paragrafo precedente, ogni azione di conteggio è  $O(n)$ . Impiegando un tempo  $O(n)$  in ciascuna visita, il tempo di esecuzione complessivo dell'algoritmo è  $O(n^2)$ .

Questo algoritmo è costituito da tre fasi:

1. Calcola tutti i conteggi.
2. Trova il valore massimo tra i conteggi.
3. Trova la posizione del valore massimo tra i conteggi.

Abbiamo appena visto che la prima fase è  $O(n^2)$ . Per trovare il valore massimo tra  $n$  conteggi serve un tempo  $O(n)$ : basta ricordare l'algoritmo visto nel Paragrafo 7.3.3 e notare che ogni passo richiede un tempo di elaborazione costante. Infine, abbiamo appena visto che la ricerca di un valore richiede un tempo  $O(n)$ .

La stima  $O$ -grande del tempo necessario per eseguire più fasi in sequenza è pari al valore di  $O$ -grande avente crescita più rapida tra quelli relativi alle singole fasi.

Come possiamo stimare il tempo di esecuzione complessivo a partire dalle stime delle singole fasi? Il tempo totale è, ovviamente, uguale alla somma dei singoli tempi, ma, usando le stime  $O$ -grande, prendiamo la stima *massima* tra quelle delle singole fasi. Per capire perché facciamo questo, immaginate di aver individuato le equazioni che descrivono effettivamente l'andamento dei tempi di esecuzione delle singole fasi, in funzione di  $n$ :

$$T_1(n) = an^2 + bn + c$$

$$T_2(n) = dn + e$$

$$T_3(n) = fn + g$$

La loro somma, che rappresenta il tempo totale, è:

$$T(n) = T_1(n) + T_2(n) + T_3(n) = an^2 + (b + d + f)n + c + e + g$$

Ma è importante soltanto il termine di grado massimo, per cui  $T(n)$  è  $O(n^2)$ .

Abbiamo, quindi, scoperto che l'algoritmo che trova l'elemento più frequente in un array è  $O(n^2)$ .

### 13.7.3 Lo schema triangolare

Cerchiamo di rendere più veloce l'algoritmo visto nel paragrafo precedente, perché ci sembra che perda tempo nel contare gli stessi elementi più volte.



Riusciamo a risparmiare tempo evitando di contare più volte uno stesso elemento? Cioè, prima di iniziare a contare quante volte ricorre nell'array il valore  $a[i]$ , non dovremmo verificare che tale valore non sia già presente nella regione  $a[0] \dots a[i - 1]$ ?

Facciamo una stima del tempo necessario a fare queste verifiche aggiuntive. Al passo  $i$ -esimo la quantità di lavoro che serve è proporzionale a  $i$ . Non è proprio come nel paragrafo precedente, dove avete visto che un ciclo avente  $n$  iterazioni, ciascuna delle quali richiede un tempo  $O(n)$ , è  $O(n^2)$ . Ora ciascuna iterazione richiede (soltanto) un tempo  $O(i)$ .

Per intuire cosa significhi tutto ciò, guardiamo di nuovo alle lampadine. Nella seconda iterazione dobbiamo ispezionare di nuovo  $a[0]$ , nella terza iterazione dobbiamo ispezionare di nuovo  $a[0]$  e  $a[1]$ , e così via. Lo schema è riportato nella pagina precedente.

Un ciclo che esegue  $n$  iterazioni,  
la  $i$ -esima delle quali richiede  
un tempo  $O(i)$ , è  $O(n^2)$ .

Se ogni riga ha  $n$  lampadine, quelle accese riempiono circa metà dello schema quadrato, cioè ci sono circa  $n^2/2$  lampadine accese: sfortunatamente, questa funzione è ancora  $O(n^2)$ .

Ma abbiamo un'altra idea che forse ci farà risparmiare tempo. Quando contiamo le ricorrenze di  $a[i]$  non c'è alcun bisogno di ispezionare la regione  $a[0] \dots a[i - 1]$ . Infatti, se il valore  $a[i]$  non è mai comparso nell'array durante le iterazioni precedenti, otteniamo il conteggio corretto ispezionando soltanto  $a[i] \dots a[n - 1]$ ; se, invece, è già comparso, allora il conteggio l'abbiamo già calcolato. Ci è d'aiuto tutto questo? In realtà no, si tratta di nuovo di uno schema triangolare, anche se nell'altra direzione:



Tutto questo, però, non significa che l'implementazione di questi miglioramenti non sia interessante. Se la soluzione migliore che si riesce a ottenere per un determinato problema

è un algoritmo  $O(n^2)$ , è comunque utile cercare di rendere più veloce la sua esecuzione. Non seguiremo, però, questa strada, perché, in effetti, nel prossimo paragrafo vedremo che possiamo fare molto meglio.

### 13.7.4 Algoritmi logaritmici

Le stime temporali logaritmiche sono solitamente relative ad algoritmi che a ogni passo suddividono il problema a metà, come abbiamo visto, per esempio, nel caso degli algoritmi di ricerca binaria e di ordinamento per fusione.

In particolare, quindi, se una delle fasi di un algoritmo è una ricerca binaria o un ordinamento per fusione, nella stima  $O$ -grande del suo tempo di esecuzione comparirà un logaritmo.

Analizziamo gli effetti di questa nuova idea per migliorare l'algoritmo precedente, che cerca, in un array, l'elemento che ricorre più frequentemente. Supponiamo di *ordinare* l'array prima di eseguire l'algoritmo:



Questa fase di ordinamento richiede un tempo  $O(n \log(n))$ . Se riusciamo a completare l'algoritmo in un tempo  $O(n)$ , allora avremo trovato una soluzione migliore di quelle precedenti, che erano  $O(n^2)$ .

Per capire come questo sia possibile, immaginate di scandire l'array ordinato. Ogni volta che trovate un valore che è uguale a quello precedente, incrementate un contatore. Quando, invece, trovate un valore diverso, memorizzate il valore del contatore e fatelo ripartire da zero:

|         |                             |
|---------|-----------------------------|
| values: | [4   5   5   7   7   7   8] |
| counts: | [1   1   2   1   2   3   1] |

Vediamo il codice:

```
int count = 0;
for (int i = 0; i < a.length; i++)
{
    count++;
    if (i == a.length - 1 || a[i] != a[i + 1])
    {
        counts[i] = count;
        count = 0;
    }
}
```

In questo algoritmo l'esecuzione di ogni iterazione richiede un tempo costante, anche se visita due elementi, come si può vedere nella pagina seguente.

La funzione  $2n$  (cioè due visite per ogni iterazione) è  $O(n)$ . Possiamo, quindi, calcolare tutti i conteggi relativi a un array ordinato in un tempo  $O(n)$ . L'intero algoritmo, ora, è  $O(n \log(n))$ .

Un algoritmo che a ogni passo divide a metà il problema viene eseguito in un tempo  $O(\log(n))$ .



Notate anche che, in realtà, non c'è alcun bisogno di memorizzare tutti i conteggi, basta soltanto tenere traccia di quello massimo visto fino a quel momento (come vedrete nell'Esercizio E13.11). Anche questo è un miglioramento che vale la pena di implementare, ma non modifica la stima O-grande del tempo di esecuzione dell'algoritmo.



### Auto-valutazione

21. Qual è lo schema di visita “a lampadine” del seguente algoritmo, che verifica se un array è un palindromo?

```
for (int i = 0; i < a.length / 2; i++)
{
    if (a[i] != a[a.length - 1 - i]) { return false; }
}
return true;
```

22. Qual è la stima O-grande del tempo di esecuzione del seguente algoritmo, che verifica se il primo elemento di un array è duplicato al suo interno?

```
for (int i = 1; i < a.length; i++)
{
    if (a[0] == a[i]) { return true; }
}
return false;
```

23. Qual è la stima O-grande del tempo di esecuzione del seguente algoritmo, che verifica se un array contiene (almeno) un elemento duplicato?

```

for (int i = 0; i < a.length; i++)
{
    for (int j = i + 1; j < a.length; j++)
    {
        if (a[i] == a[j]) { return true; }
    }
}
return false;

```

24. Descrivete un algoritmo che verifichi se un array contiene (almeno) un elemento duplicato e che venga eseguito in un tempo  $O(n \log(n))$ .
25. Qual è la stima O-grande del tempo di esecuzione del seguente algoritmo, che cerca un elemento (`value`) in un array bidimensionale  $n \times n$ ?

```

for (int i = 0; i < n; i++)
{
    for (int j = 0; j < n; j++)
    {
        if (a[i][j] == value) { return true; }
    }
}
return false;

```

26. Se si esegue l'algoritmo visto nel Paragrafo 13.7.4 su un array bidimensionale  $n \times n$ , qual è la stima O-grande del tempo richiesto, in funzione di  $n$ , per trovare l'elemento che ricorre più frequentemente?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R13.9, R13.15, R13.21 e E13.11, al termine del capitolo.

## 13.8 Ordinamento e ricerca nella libreria Java

Quando scrivete programmi in Java non avete bisogno di implementare algoritmi di ordinamento: le classi `Arrays` e `Collections` contengono metodi di ordinamento e ricerca, presentati nei prossimi paragrafi.

### 13.8.1 Ordinamento

La classe `Arrays` contiene il metodo di ordinamento che dovrebbe essere normalmente utilizzato nei programmi Java.

La classe `Arrays` contiene metodi statici, `sort`, che sono in grado di ordinare array di numeri interi e di numeri in virgola mobile. Ad esempio, potete ordinare un array di numeri interi scrivendo semplicemente così:

```

int[] a = . . .;
Arrays.sort(a);

```

Tale metodo `sort` usa l'algoritmo quicksort presentato nella sezione Argomenti avanzati 13.3.

La classe Collections contiene un metodo di ordinamento che agisce su vettori.

Se i dati da ordinare sono contenuti in un oggetto di tipo `ArrayList` si usa invece il metodo `sort` della classe `Collections`, che implementa l'algoritmo di ordinamento per fusione:

```
ArrayList<String> names = . . .;
Collections.sort(names);
```

### 13.8.2 Ricerca binaria

Le classi `Arrays` e `Collections` contengono anche metodi statici, `binarySearch`, che implementano l'algoritmo di ricerca binaria, con l'aggiunta di un utile miglioramento: se il valore cercato non è presente nell'array, il valore restituito non è  $-1$ , ma  $-k - 1$ , dove  $k$  è la posizione in cui dovrebbe essere inserito il valore, se lo si volesse inserire. Ad esempio:

```
int[] a = { 1, 4, 9 };
int v = 7;
int pos = Arrays.binarySearch(a, v);
// restituisce -3: v dovrebbe essere inserito nella posizione 2
```

### 13.8.3 Confronto di oggetti

Il metodo `sort` della classe `Arrays` ordina oggetti di classi che implementano l'interfaccia `Comparable`.

Nei programmi applicativi si ha spesso bisogno di ordinare raccolte di oggetti, per poi farvi ricerche. Per questo motivo le classi `Arrays` e `Collections` contengono anche metodi `sort` e `binarySearch` che operano su raccolte di oggetti, anche se non sanno come confrontare tra loro oggetti di tipo arbitrario. Immaginiamo, ad esempio, di avere un array di oggetti di tipo `Country`. Non è per niente ovvio decidere come ordinare nazioni: per nome o sulla base della loro superficie? I metodi `sort` e `binarySearch` non possono prendere una simile decisione per conto vostro. Infatti, richiedono che gli oggetti appartengano a classi che implementano l'interfaccia `Comparable`, presentata nel Paragrafo 10.3 e dotata di un unico metodo::

```
public interface Comparable<T>
{
    int compareTo(T other);
}
```

L'invocazione

```
a.compareTo(b)
```

deve restituire: un numero negativo se `a` precede `b`; un numero positivo se `a` segue `b`; zero se `a` e `b` sono uguali.

Osservate che `Comparable` è un tipo generico, come `ArrayList`. Nel caso di un oggetto `ArrayList`, il parametro di tipo specifica il tipo degli elementi del vettore. Nel caso di `Comparable`, il parametro di tipo è il tipo del parametro ricevuto dal metodo `compareTo`, quindi solitamente una classe che implementa `Comparable` vorrà essere “confrontabile con se stessa”. Ad esempio, la classe `Country` implementa `Comparable<Country>`.

Molte classi della libreria standard di Java (come, ad esempio, la classe `String`, le classi involucro per i tipi numerici, le classi che rappresentano una data o il percorso che identifica un file) implementano l'interfaccia `Comparable`.

Potete implementare l'interfaccia `Comparable` anche nelle vostre classi. Ad esempio, per ordinare una raccolta di nazioni in base alla loro superficie, la classe `Country` dovrebbe implementare l'interfaccia `Comparable<Country>`, dichiarando un metodo `compareTo` come questo:

```
public class Country implements Comparable<Country>
{
    ...
    public int compareTo(Country other)
    {
        return Double.compare(area, other.area);
    }
}
```

Il metodo `compareTo` confronta nazioni in base alla superficie. Notate l'uso del metodo ausiliario `Double.compare` (visto nella sezione Suggerimenti per la programmazione 10.1), che restituisce un numero negativo, zero o un numero positivo. Questa soluzione è più semplice rispetto alla scrittura di una diramazione a tre uscite.

A questo punto è possibile fornire al metodo `Arrays.sort` un array di nazioni:

```
Country[] countries = new Country[n];
// aggiungi nazioni
...
Arrays.sort(countries); // ordina per superficie crescente
```

Quando dovete eseguire ordinamenti o ricerche, usate i metodi delle classi `Arrays` e `Collections` e non altri scritti da voi: gli algoritmi della libreria sono stati ben collaudati e ottimizzati. L'obiettivo principale di questo capitolo non è stato quello di insegnarvi a realizzare algoritmi di ordinamento e ricerca. Avete, invece, appreso una cosa più importante: algoritmi diversi possono avere prestazioni ben diverse, per cui è utile conoscere meglio la progettazione e l'analisi di algoritmi.



## Auto-valutazione

27. Perché il metodo `Arrays.sort` non può ordinare un array di oggetti `Rectangle`?
28. Cosa bisogna fare per mettere in ordine di saldo crescente un array di oggetti `BankAccount`?
29. Perché è utile che il metodo `Arrays.binarySearch` restituisca la posizione in cui inserire un elemento mancante?
30. Perché il metodo `Arrays.binarySearch` restituisce  $-k - 1$  e non  $-k$  per segnalare che il valore cercato non è presente e che, se lo si vuole inserire, va posto nella posizione  $k$ ?

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi E13.12, E13.16 e E13.17, al termine del capitolo.



## Errori comuni 13.1

**Il metodo compareTo può restituire qualsiasi numero intero, non solo -1, 0 o 1**

Quando l'invocazione `a.compareTo(b)` deve segnalare che `a` precede `b` può restituire *qualsiasi* numero intero negativo, non necessariamente il valore `-1`. Di conseguenza, la verifica

```
if (a.compareTo(b) == -1) // ERRORE !
```

è, in generale, errata. Dovete scrivere, invece

```
if (a.compareTo(b) < 0) // così va bene
```

Perché mai un metodo `compareTo` dovrebbe voler restituire un numero diverso da `-1`, `0` o `1`? A volte è comodo restituire semplicemente la differenza tra due numeri interi. Ad esempio, il metodo `compareTo` della classe `String` confronta i caratteri che si trovano in posizioni corrispondenti:

```
char c1 = charAt(i);
char c2 = other.charAt(i);
```

Se i caratteri sono diversi, allora il metodo può semplicemente restituire la loro differenza:

```
if (c1 != c2) { return c1 - c2; }
```

Se `c1` precede `c2` questa differenza è certamente un numero negativo, ma non è necessariamente il numero `-1`.

Si osservi che la restituzione di una differenza funziona soltanto se non si verifica una condizione di overflow (Suggerimenti per la programmazione 10.1).



## Argomenti avanzati 13.4

### L'interfaccia Comparator

A volte si vuole ordinare un array o un vettore di oggetti che non appartengono a una classe che implementa l'interfaccia `Comparable`, oppure si vuole ordinare l'array in un modo diverso da quello indotto dal metodo `compareTo`: ad esempio, può darsi che si vogliano ordinare nazioni per nome che per superficie.

Non vorreste essere costretti a modificare il codice di una classe soltanto per poter invocare `Arrays.sort` e, fortunatamente, esiste un'alternativa. Una versione del metodo `Arrays.sort` non richiede che gli oggetti da ordinare appartengano a una classe che realizza l'interfaccia `Comparable`: potete fornire oggetti di qualsiasi tipo, ma dovete anche fornire un *comparatore* di oggetti, che ha il compito, appunto, di confrontare gli oggetti che volete ordinare. L'oggetto comparatore deve appartenere a una classe che implementa l'interfaccia `Comparator`: ha un unico metodo, `compare`, che confronta due oggetti.

Come `Comparable`, l'interfaccia `Comparator` è un tipo parametrico, il cui parametro di tipo specifica di che tipo sono i due parametri ricevuti dal metodo `compare`. Ad esempio, l'interfaccia `Comparator<Country>` è questa:

```
public interface Comparator<Country>
{
    int compare(Country a, Country b);
}
```

Se `comp` è un oggetto di una classe che implementa `Comparator<Country>`, l'invocazione

```
comp.compare(a, b)
```

deve restituire: un numero negativo se `a` precede `b`; un numero positivo se `a` segue `b`; zero se `a` e `b` sono uguali.

Ad esempio, ecco una classe `Comparator` per nazioni:

```
public class CountryComparator implements Comparator<Country>
{
    public int compare(Country a, Country b)
    {
        return Double.compare(a.getArea(), b.getArea());
    }
}
```

Per ordinare un array di nazioni, `countries`, in base alla superficie, si può ora invocare:

```
Arrays.sort(countries, new CountryComparator());
```

## Note per Java 8 13.1

---

### Comparatori definiti mediante espressioni lambda

Nelle versioni precedenti a Java 8 la definizione di un comparatore era abbastanza scomoda: bisognava definire una classe che implementasse l'interfaccia `Comparator`, realizzandone il metodo `compare`, per poi costruire un esemplare di tale classe. Una vera sfortuna, perché in realtà i comparatori sono molto utili e si usano in diversi algoritmi, per fare ricerche, ordinare dati e trovare il massimo o il minimo in un insieme.

Con l'avvento delle espressioni lambda, la definizione di un comparatore è diventata molto più semplice. Ad esempio, per ordinare un array di parole sulla base della loro lunghezza, si può scrivere:

```
Arrays.sort(words, (v, w) -> v.length() - w.length());
```

Per casi come questo esiste anche una comoda abbreviazione. Osservate che il risultato del confronto dipende da una funzione che mette in corrispondenza ciascuna stringa con un valore numerico, in questo caso la sua lunghezza. Il metodo statico `Comparator.comparing` costruisce proprio un comparatore a partire da un'espressione lambda. Ad esempio, si può scrivere:

```
Arrays.sort(words, Comparator.comparing(w -> w.length()));
```

Viene così costruito un comparatore che invoca per entrambi gli oggetti da confrontare la funzione fornita come parametro, confrontando poi i risultati ottenuti.

Il metodo `Comparator.comparing` è in grado di gestire molti casi. Ad esempio, per ordinare nazioni in base alla loro superficie, è sufficiente scrivere:

```
Arrays.sort(countries, Comparator.comparing(c -> c.getArea()));
```



## Esempi completi 13.1

### Migliorare l'algoritmo di ordinamento per inserimento

**Problema.** Implementare un algoritmo (chiamato *ordinamento di Shell* dal nome del suo inventore, Donald Shell) che migliora l'algoritmo di ordinamento per inserimento visto nella sezione Argomenti avanzati 13.2.

L'ordinamento di Shell migliora l'ordinamento per inserimento sfruttando il fatto che tale algoritmo è  $O(n)$  quando l'array da ordinare è, in realtà, già ordinato. L'ordinamento di Shell ordina alcune porzioni dell'array, poi esegue l'ordinamento per inserimento sull'intero array: in questo modo, tale ordinamento conclusivo non ha poi tanto lavoro da fare.

Una fase chiave dell'ordinamento di Shell consiste nel disporre la sequenza da ordinare in una tabella, con righe e colonne, per poi ordinare separatamente ciascuna colonna. Ad esempio, se l'array è questo:

|    |    |    |    |    |   |    |    |    |    |    |   |    |    |    |    |    |    |    |    |
|----|----|----|----|----|---|----|----|----|----|----|---|----|----|----|----|----|----|----|----|
| 65 | 46 | 14 | 52 | 38 | 2 | 96 | 39 | 14 | 33 | 13 | 4 | 24 | 99 | 89 | 77 | 73 | 87 | 36 | 81 |
|----|----|----|----|----|---|----|----|----|----|----|---|----|----|----|----|----|----|----|----|

e lo disponiamo in una tabella di quattro colonne, otteniamo:

|    |    |    |    |
|----|----|----|----|
| 65 | 46 | 14 | 52 |
| 38 | 2  | 96 | 39 |
| 14 | 33 | 13 | 4  |
| 24 | 99 | 89 | 77 |
| 73 | 87 | 36 | 81 |

Ora ordiniamo le singole colonne:

|    |    |    |    |
|----|----|----|----|
| 14 | 2  | 13 | 5  |
| 24 | 33 | 14 | 39 |
| 38 | 46 | 36 | 52 |
| 65 | 87 | 89 | 77 |
| 73 | 99 | 96 | 81 |

e ricostruiamo l'array intero, leggendo i dati dalla tabella, riga per riga:

|    |   |    |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|----|---|----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 14 | 2 | 13 | 5 | 24 | 33 | 14 | 39 | 38 | 46 | 36 | 52 | 65 | 87 | 89 | 77 | 73 | 99 | 96 | 81 |
|----|---|----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|

Come si può notare, l'array non è completamente ordinato, ma molti dei numeri di valore minore si trovano nelle prime posizioni, mentre, al contrario, molti dei numeri di valore maggiore si trovano nelle ultime posizioni.

Ora ripeteremo la procedura finché l'intero array non risulterà ordinato, ogni volta usando un numero di colonne diverso. Shell usò originariamente colonne il cui numero era sempre una potenza di due: ad esempio, per ordinare un array di 20 elementi propose di usare prima 16 colonne, poi 8, 4, 2 e, infine, una sola colonna. Usando una sola colonna ci troviamo di fronte a una normale implementazione dell'algoritmo di ordinamento per inserimento, per cui siamo certi che alla fine l'array risulterà essere ordinato. La cosa in qualche modo sorprendente è il fatto che le fasi di ordinamento precedenti rendano più veloce questo ordinamento finale.

In ogni caso, sono state individuate sequenze di numeri di colonne che si comportano in modo più efficiente, per cui useremo una di queste:

$$\begin{aligned}c_1 &= 1 \\c_2 &= 4 \\c_3 &= 13 \\c_4 &= 40 \\\dots \\c_{i+1} &= 3c_i + 1\end{aligned}$$

Quindi, per ordinare un array di 20 elementi, partiamo con l'ordinamento di 13 colonne, per poi passare a 4 colonne e, infine, a una sola colonna. Questa sequenza è quasi efficiente quanto la migliore che si conosca, ma è più facile da calcolare.

In realtà, non si modifica effettivamente il contenuto dell'array durante ogni fase, bensì si manipolano le posizioni degli elementi di ciascuna colonna.

Se, ad esempio, il numero di colonne,  $c$ , è uguale a 4, gli elementi delle quattro colonne sono posizionati nell'array in questo modo:

|    |    |    |    |    |   |    |    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|---|----|----|----|----|----|----|----|----|----|----|----|
| 65 |    |    |    | 38 |   |    | 14 |    |    | 24 |    |    | 73 |    |    |    |
|    | 46 |    |    |    | 2 |    |    | 33 |    |    | 99 |    |    | 87 |    |    |
|    |    | 14 |    |    |   | 96 |    |    | 13 |    |    | 89 |    |    | 36 |    |
|    |    |    | 52 |    |   |    | 39 |    |    | 4  |    |    | 77 |    |    | 81 |

Come si può facilmente notare, i numeri appartenenti a una stessa colonna si trovano a distanza  $c$  l'uno dall'altro e la colonna  $k$ -esima è composta dagli elementi  $a[k], a[k + c], a[k + 2 * c]$ , e così via.

Ora modifichiamo l'algoritmo di ordinamento per inserimento in modo che ordini una di queste colonne. L'algoritmo originario è il seguente:

```
for (int i = 1; i < a.length; i++)
{
    int next = a[i];
    // sposta in avanti tutti gli elementi maggiori
    int j = i;
    while (j > 0 && a[j - i] > next)
    {
        a[j] = a[j - 1];
        j--;
    }
}
```

```

    }
    // inserisci l'elemento
    a[j] = next;
}

```

Il ciclo più esterno ispeziona gli elementi  $a[1]$ ,  $a[2]$ , e così via. La sequenza corrispondente alla  $k$ -esima colonna è  $a[k + c], a[k + 2 * c], \dots$ , quindi il ciclo più esterno diventa:

```
for (int i = k + c; i < a.length; i = i + c)
```

Nel ciclo interno l'algoritmo originario visitava  $a[j], a[j - 1], \dots$ , e così via. Dobbiamo fare in modo che visiti  $a[j], a[j - c], \dots$ , quindi diventa:

```

while (j >= c && a[j - c] > next)
{
    a[j] = a[j - c];
    j = j - c;
}

```

Mettendo tutto insieme, otteniamo il metodo seguente:

```

/**
 * Ordina una colonna usando l'ordinamento per inserimento.
 * @param a l'array da ordinare
 * @param k l'indice del primo elemento della colonna
 * @param c lo spazio tra elementi della colonna
 */
public static void insertionSort(int[] a, int k, int c)
{
    for (int i = k + c; i < a.length; i = i + c)
    {
        int next = a[i];
        // sposta in avanti tutti gli elementi maggiori
        int j = i;
        while (j >= c && a[j - c] > next)
        {
            a[j] = a[j - c];
            j = j - c;
        }
        // inserisci l'elemento
        a[j] = next;
    }
}

```

A questo punto tutto è pronto perché possiamo implementare l'algoritmo di ordinamento di Shell. Per prima cosa dobbiamo calcolare quanti elementi della sequenza di numeri di colonne ci servono. Quindi, generiamo i numeri di tale sequenza, uno dopo l'altro, fino a quando non viene superata la dimensione della lista da ordinare:

```

ArrayList<Integer> columns = new ArrayList<Integer>();
int c = 1;
while (c < a.length)
{
    columns.add(c);
    c = 3 * c + 1;
}

```

Per ogni valore di `columns` dobbiamo ordinare il contenuto di tutte le singole colonne:

```
for (int s = columns.size() - 1; s >= 0; s--)  
{  
    c = columns.get(s);  
    for (int k = 0; k < c; k++)  
    {  
        insertionSort(a, k, c);  
    }  
}
```

Le prestazioni sono buone? Confrontiamole con quelle del metodo `Arrays.sort` della libreria di Java:

```
int[] a = ArrayUtil.randomIntArray(n, 100);  
int[] a2 = Arrays.copyOf(a, a.length);  
int[] a3 = Arrays.copyOf(a, a.length);  
  
StopWatch timer = new StopWatch();  
  
timer.start();  
ShellSorter.sort(a);  
timer.stop();  
  
System.out.println("Elapsed time with Shell sort: "  
    + timer.getElapsedTime() + " milliseconds");  
  
timer.reset();  
timer.start();  
Arrays.sort(a2);  
timer.stop();  
  
System.out.println("Elapsed time with Arrays.sort: "  
    + timer.getElapsedTime() + " milliseconds");  
  
if (!Arrays.equals(a, a2))  
{  
    throw new IllegalStateException("Incorrect sort result");  
}  
  
timer.reset();  
timer.start();  
InsertionSorter.sort(a3);  
timer.stop();  
  
System.out.println("Elapsed time with insertion sort: "  
    + timer.getElapsedTime() + " milliseconds");
```

Ci siamo assicurati che i tre algoritmi ordinino lo stesso array, facendo copie di quello riempito di valori casuali; inoltre, abbiamo verificato che il risultato prodotto dall'ordinamento di Shell sia corretto, confrontandolo con quello prodotto da `Arrays.sort`. Infine, lo confrontiamo anche con l'ordinamento per inserimento.

I risultati mostrano che l'ordinamento di Shell è decisamente più veloce dell'ordinamento per inserimento:

```
Enter array size: 1000000
Elapsed time with Shell sort: 205 milliseconds
Elapsed time with Arrays.sort: 101 milliseconds
Elapsed time with insertion sort: 148196 milliseconds
```

L'algoritmo quicksort (usato da `Arrays.sort`) batte, però, l'ordinamento di Shell e per questo motivo quest'ultimo praticamente non viene utilizzato, anche se è un algoritmo interessante, con un'efficienza che lascia un po' sorpresi.

Potete fare qualche esperimento anche usando i numeri di colonne individuati nel lavoro originale di Shell. Basta sostituire, nel metodo `ShellSorter.sort`, l'enunciato:

```
c = 3 * c + 1;
```

con:

```
c = 2 * c;
```

Scoprirete che, in questo caso, l'algoritmo è circa tre volte più lento rispetto a quello che usa la sequenza migliore, ma rimane molto più veloce dell'ordinamento per inserimento.

Il programma completo si trova nella cartella `worked_example_1` del Capitolo 13 del pacchetto dei file scaricabili per questo libro.

## Riepilogo degli obiettivi di apprendimento

### Descrizione dell'algoritmo di ordinamento per selezione

- L'algoritmo di ordinamento per selezione ordina un array cercando ripetutamente l'elemento minimo della zona terminale non ancora ordinata, spostandolo all'inizio della zona stessa.

### Misurazione del tempo di esecuzione di un metodo

- Per misurare il tempo di esecuzione di un metodo, chiedete l'ora al sistema subito prima e subito dopo l'invocazione del metodo stesso.

### Utilizzo della notazione O-grande per descrivere il tempo di esecuzione di un algoritmo

- Nell'informatica teorica si descrive il tasso di crescita di una funzione usando la notazione O-grande.
- L'ordinamento per selezione è un algoritmo  $O(n^2)$ : il raddoppio della dimensione dell'insieme di dati quadruplica il tempo di elaborazione.
- L'ordinamento per inserimento è un algoritmo  $O(n^2)$ .

### Descrizione dell'algoritmo di ordinamento per fusione (*mergesort*)

- L'algoritmo di ordinamento per fusione ordina un array dividendolo a metà, ordinando ricorsivamente ciascuna delle due parti e fondendo, poi, le due metà ordinate.

### Tempi di esecuzione degli algoritmi di ordinamento per fusione e per selezione

- L'ordinamento per fusione è un algoritmo  $O(n \log(n))$ . La funzione  $n \log(n)$  cresce molto più lentamente di  $n^2$ .

### Tempi di esecuzione degli algoritmi di ricerca lineare e di ricerca binaria

- La ricerca lineare esamina tutti i valori di un array fino a trovare una corrispondenza con quanto cercato o la fine dell'array.
- La ricerca lineare trova un valore in un array eseguendo un numero di passi  $O(n)$ .
- La ricerca binaria cerca un valore in un array ordinato determinando se si può trovare nella prima o nella seconda metà dell'array, ripetendo poi la ricerca in una delle due metà.
- La ricerca binaria trova la posizione di un valore in un array eseguendo un numero di passi  $O(\log(n))$ .

### Stime dell'andamento O-grande delle prestazioni di algoritmi

- Un ciclo che esegue  $n$  iterazioni, ciascuna delle quali è costituita da un numero fisso di azioni, richiede un tempo  $O(n)$ .
- Un ciclo che esegue  $n$  iterazioni, ciascuna delle quali richiede un tempo  $O(n)$ , è  $O(n^2)$ .
- La stima O-grande del tempo necessario per eseguire più fasi in sequenza è pari al valore di O-grande avente crescita più rapida tra quelli relativi alle singole fasi.
- Un ciclo che esegue  $n$  iterazioni, la  $i$ -esima delle quali richiede un tempo  $O(i)$ , è  $O(n^2)$ .
- Un algoritmo che a ogni passo divide a metà il problema viene eseguito in un tempo  $O(\log(n))$ .

### Utilizzo dei metodi della libreria di Java per ordinare dati e fare ricerche

- La classe `Arrays` contiene il metodo di ordinamento che dovrebbe essere normalmente utilizzato nei programmi Java.
- La classe `Collections` contiene un metodo di ordinamento che agisce su vettori.
- Il metodo `sort` della classe `Arrays` ordina oggetti di classi che implementano l'interfaccia `Comparable`.

## Elementi di libreria presentati in questo capitolo

`java.lang.System`  
`currentTimeMillis`  
`java.util.Arrays`  
`binarySearch`  
`sort`

`java.util.Collections`  
`binarySearch`  
`sort`  
`java.util.Comparator<T>`  
`compare`  
`comparing`

## Esercizi di riepilogo e approfondimento

- ★ **R13.1.** Qual è la differenza fra cercare e ordinare?
- ★★ **R13.2.** *Attenzione al rischio di errori per scarto di uno.* Scrivendo l'algoritmo di ordinamento per selezione visto nel Paragrafo 13.1, occorre, come al solito, scegliere tra `<` e `<=`, tra `a.length` e `a.length - 1`, e tra `from` e `from + 1`: un terreno fertile per la proliferazione degli errori per scarto di uno. Eseguite passo dopo passo il codice dell'algoritmo applicato ad array di lunghezza 0, 1, 2 e 3, controllando accuratamente che tutti i valori degli indici siano corretti.
- ★★ **R13.3.** Qual è l'andamento di crescita, in termini di O-grande, di queste funzioni?
  - $n^2 + 2n + 1$
  - $n^{10} + 9n^9 + 20n^8 + 145n^7$
  - $(n + 1)^4$
  - $(n^2 + n)^2$

- e.  $n + 0.001n^3$
- f.  $n^3 - 1000n^2 + 10^9$
- g.  $n + \log(n)$
- h.  $n^2 + n \log(n)$
- i.  $2^n + n^2$
- j.  $(n^3 + 2n)/(n^2 + 0.75)$

- \* **R13.4.** Abbiamo calcolato che il numero effettivo di visite richieste dall'algoritmo di ordinamento per selezione è

$$T(n) = (1/2)n^2 + (5/2)n - 3$$

Abbiamo poi stabilito che questo metodo è caratterizzato da una crescita  $O(n^2)$ . Calcolate i rapporti effettivi

$$\begin{aligned} T(2000)/T(1000) \\ T(4000)/T(1000) \\ T(10000)/T(1000) \end{aligned}$$

e, posto  $f(n) = n^2$ , confrontateli con

$$\begin{aligned} f(2000)/f(1000) \\ f(4000)/f(1000) \\ f(10000)/f(1000) \end{aligned}$$

- \* **R13.5.** Supponiamo che l'algoritmo  $A$  impieghi 5 secondi per elaborare un insieme di 1000 dati. Se l'algoritmo  $A$  ha prestazioni  $O(n)$ , quanto tempo impiegherà approssimativamente per elaborare un insieme di 2000 dati? E uno di 10000?
- \*\* **R13.6.** Supponiamo che un algoritmo impieghi 5 secondi per elaborare un insieme di 1000 dati. Riempite la tabella seguente, che mostra approssimativamente la crescita del tempo di esecuzione in funzione della complessità dell'algoritmo.

|       | $O(n)$ | $O(n^2)$ | $O(n^3)$ | $O(n \log n)$ | $O(2^n)$ |
|-------|--------|----------|----------|---------------|----------|
| 1000  | 5      | 5        | 5        | 5             | 5        |
| 2000  |        |          |          |               |          |
| 3000  |        |          | 45       |               |          |
| 10000 |        |          |          |               |          |

Per esempio, dal momento che  $3000^2/1000^2 = 9$ , se l'algoritmo fosse  $O(n^2)$ , per elaborare un insieme di 3000 dati impiegherebbe un tempo 9 volte superiore a quello necessario per elaborare un insieme di 1000 dati, cioè 45 secondi.

- \*\* **R13.7.** Ordinate le seguenti espressioni O-grande in ordine crescente.

- $O(n)$
- $O(n^3)$
- $O(n^n)$
- $O(\log(n))$
- $O(n^2 \log(n))$
- $O(n \log(n))$
- $O(2^n)$

$O(\sqrt{n})$   
 $O(n\sqrt{n})$   
 $O(n^{\log(n)})$

- \* **R13.8.** Qual è l'andamento del tempo di esecuzione dell'algoritmo standard che trova il valore minimo in un array? E di quello che trova sia il minimo che il massimo?
- \* **R13.9.** Qual è l'andamento del tempo di esecuzione del seguente metodo, in funzione di  $n$ , la lunghezza di  $a$ ? Visualizzate il risultato usando il “metodo delle lampadine” visto nel Paragrafo 13.7.

```
public static void swap(int[] a)
{
    int i = 0;
    int j = a.length - 1;
    while (i < j)
    {
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
        i++;
        j--;
    }
}
```

- \* **R13.10.** Una “serie” (*run*) è una sequenza di valori adiacenti ripetuti (si veda l'Esercizio R7.23). Descrivete un algoritmo  $O(n)$  che trovi la lunghezza della serie più lunga in un array.
- \*\* **R13.11.** Considerate il problema di trovare l'elemento più frequente in un array di lunghezza  $n$ . Ecco tre possibili approcci:
  - Ordinare l'array, poi trovare la serie più lunga.
  - Creare un array di contatori avente la stessa dimensione dell'array originario. Per ciascun elemento, scandire l'intero array contando il numero di elementi uguali a quello in esame, aggiornando il relativo contatore. Quindi, trovare il contatore che ha il valore massimo.
  - Usare variabili per memorizzare l'elemento visto più frequentemente finora e la sua frequenza. Per ogni indice  $i$ , verificare se  $a[i]$  è presente nelle posizioni  $a[0] \dots a[i - 1]$ . Se non è presente, contare quante volte è presente in  $a[i + 1] \dots a[n - 1]$ . Se  $a[i]$  ricorre più volte dell'elemento visto più frequentemente finora, aggiornare le variabili.
 Descrivete l'efficienza dei tre approcci in termini di O-grande.
- \* **R13.12.** Seguite, passo dopo passo, l'esecuzione dell'algoritmo di ordinamento per selezione applicato a questi due array:
  - 4, 7, 11, 4, 9, 5, 11, 7, 3, 5
  - 7, 6, 8, 7, 5, 9, 0, 11, 10, 5, 8
- \* **R13.13.** Seguite, passo dopo passo, l'esecuzione dell'algoritmo di ordinamento per fusione applicato a questi due array:
  - 5, 11, 7, 3, 5, 4, 7, 11, 4, 9
  - 9, 0, 11, 10, 5, 8, -7, 6, 8, 7, 5
- \* **R13.14.** Seguite, passo dopo passo, l'esecuzione dei seguenti algoritmi:
  - Ricerca lineare di 7 in -7, 1, 3, 3, 4, 7, 11, 13
  - Ricerca binaria di 8 in -7, 2, 2, 3, 4, 7, 8, 11, 13
  - Ricerca binaria di 8 in -7, 1, 2, 3, 5, 7, 10, 13

- \*\* **R13.15.** Il vostro compito consiste nel togliere tutti i duplicati da un array. Per esempio, se l'array contiene i valori

4 7 11 4 9 5 11 7 3 5

allora deve essere modificato in modo da contenere

4 7 11 9 5 3

Ecco un semplice algoritmo per risolvere il problema. Esaminate  $a[i]$  e contate quante volte ricorre in  $a$ : se il conteggio è maggiore di 1, eliminatelo. Qual è l'andamento del tempo di esecuzione di questo algoritmo?

- \*\*\* **R13.16.** Modificate l'algoritmo di ordinamento per fusione in modo che elimini gli elementi duplicati durante la fase di fusione, ottenendo così un algoritmo che elimina gli elementi duplicati da un array. Si osservi che, alla fine, l'array non ha necessariamente lo stesso ordinamento dell'array originario. Qual è l'efficienza di questo algoritmo?

- \*\* **R13.17.** Considerate il seguente algoritmo che elimina tutti i duplicati da un array. Ordinate l'array; poi, esaminando ciascuno dei suoi elementi, per vedere se è presente più di una volta esaminate l'elemento seguente e, in caso affermativo, eliminate lo. Questo algoritmo è più veloce di quello dell'Esercizio R13.15?

- \*\*\* **R13.18.** Mettete a punto un algoritmo  $O(n \log(n))$  per eliminare i duplicati da un array nel caso in cui l'array risultante debba avere lo stesso ordinamento di quello originale. Quando un valore è presente più volte, devono essere eliminate tutte le sue occorrenze tranne la prima.

- \*\*\* **R13.19.** Perché, quando l'array è già ordinato, l'algoritmo di ordinamento per inserimento è significativamente più veloce dell'ordinamento per selezione?

- \*\*\* **R13.20.** Considerate la seguente modifica, che migliora le prestazioni dell'algoritmo di ordinamento per inserimento visto nella sezione Argomenti avanzati 13.2: per ogni elemento dell'array, invocate `Arrays.binarySearch` per determinare la posizione in cui va inserito. Questo miglioramento ha un impatto significativo sull'efficienza dell'algoritmo?

- \*\* **R13.21.** Considerate il seguente algoritmo, noto come *ordinamento a bolle (bubble sort)*:

Finché l'array non è ordinato  
Per ogni coppia di elementi adiacenti  
    Se la coppia non è ordinata  
        Scambia i suoi elementi.

Qual è la sua efficienza, in termini di O-grande?

- \*\* **R13.22.** L'algoritmo di *ordinamento per radice (radix sort)* ordina un array di  $n$  numeri interi, ciascuno avente  $d$  cifre in formato decimale, usando dieci array ausiliari associati agli indici da 0 a 9. Per prima cosa, inserisce ciascun valore  $v$  dell'array nell'array ausiliario il cui indice corrisponde all'ultima cifra di  $v$ , quindi trasferisce nuovamente tutti gli elementi nell'array originale, preservando il loro ordine (cioè prima trasferisce gli elementi dall'array ausiliario 0, poi quelli dall'array ausiliario 1, e così via). Quindi la procedura viene ripetuta usando la penultima cifra di ciascun valore come indice per decidere l'array ausiliario di destinazione, e così via. Qual è l'efficienza di questo algoritmo in termini di O-grande, in funzione di  $n$  e di  $d$ ? In quali casi questo algoritmo è da preferirsi rispetto all'ordinamento per fusione?

- \*\* **R13.23.** Un algoritmo di ordinamento *stabile (stable sort)* non modifica l'ordine relativo di elementi che hanno lo stesso valore, una caratteristica utile e richiesta in molte applicazioni. Considerate,

infatti, una sequenza di messaggi di posta elettronica (*e-mail*): se prima li ordinate in base alla data e, poi, li ordinate di nuovo in base al nome del mittente, sarebbe davvero opportuno che la seconda procedura di ordinamento preservasasse l'ordine relativo tra gli elementi indotto dalla prima procedura, in modo che l'utente possa vedere consecutivamente tutti i messaggi che hanno un certo mittente, ordinati tra loro in base alla data. L'algoritmo di ordinamento per selezione è stabile? E l'algoritmo di ordinamento per inserimento? Perché?

- \*\* **R13.24.** Descrivete un algoritmo che in un tempo  $O(n)$  ordini un array di  $n$  byte (cioè di numeri compresi tra -128 e 127). *Suggerimento:* usate un array di contatori.
- \*\* **R13.25.** Dopo aver rappresentato le pagine di un libro con una sequenza di array di parole, dovere costruire il relativo indice analitico (che è un array di parole ordinato), ciascun elemento del quale è associato a un array ordinato di numeri: le pagine in cui tale parola compare. Descrivete un algoritmo che costruisca tale indice e fate una stima  $O$ -grande del suo tempo di esecuzione in funzione del numero totale di parole.
- \*\* **R13.26.** Dati due array, ciascuno dei quali contiene  $n$  numeri interi, descrivete un algoritmo  $O(n \log(n))$  che determini se hanno (almeno) un elemento in comune.
- \*\*\* **R13.27.** Dato un array contenente  $n$  numeri interi e un valore  $v$ , descrivete un algoritmo  $O(n \log(n))$  che determini se nell'array sono presenti due valori,  $x$  e  $y$ , la cui somma sia uguale a  $v$ .
- \*\* **R13.28.** Dati due array, ciascuno dei quali contiene  $n$  numeri interi, descrivete un algoritmo  $O(n \log(n))$  che determini tutti gli elementi che hanno in comune.
- \*\* **R13.29.** Immaginate di modificare l'algoritmo quicksort, descritto nella sezione Argomenti avanzati 13.3, in modo che selezioni come pivot l'elemento centrale dell'array anziché il suo primo elemento. Quali sono le prestazioni di questa variante dell'algoritmo quando viene eseguito su un array già ordinato?
- \*\* **R13.30.** Immaginate di modificare l'algoritmo quicksort, descritto nella sezione Argomenti avanzati 13.3, in modo che selezioni come pivot l'elemento centrale dell'array anziché il suo primo elemento. Individuate una sequenza di valori per il cui ordinamento questo algoritmo richieda un tempo  $O(n^2)$ .

## Esercizi di programmazione

- \* **E13.1.** Modificate l'algoritmo di ordinamento per selezione in modo che ordini un array di numeri interi in ordine decrescente.
- \* **E13.2.** Modificate l'algoritmo di ordinamento per selezione in modo che ordini un array di monete in base al loro valore.
- \*\* **E13.3.** Scrivete un programma che generi automaticamente la tabella dei tempi di esecuzione dell'ordinamento per selezione. Il programma deve chiedere i valori minimo e massimo di  $n$  e il numero di misurazioni da effettuare, per poi attivare tutte le esecuzioni.
- \* **E13.4.** Modificate l'algoritmo di ordinamento per fusione in modo che ordini un array di stringhe in ordine lessicografico.
- \*\* **E13.5.** Modificate l'algoritmo di ordinamento per selezione in modo che ordini un array di oggetti che implementano l'interfaccia `Measurable` vista nel Capitolo 10.
- \*\* **E13.6.** Modificate l'algoritmo di ordinamento per selezione in modo che ordini un array di oggetti che implementano l'interfaccia `Comparable` (nella versione non generica, cioè senza parametro di tipo).

- \*\* **E13.7.** Modificate l'algoritmo di ordinamento per selezione in modo che ordini un array di oggetti, ricevendo un parametro di tipo `Comparator` (nella versione non generica, cioè senza parametro di tipo).
- \*\*\* **E13.8.** Scrivete un programma per consultare l'elenco del telefono. Leggete un insieme di dati contenente 1000 nomi e i relativi numeri di telefono, memorizzati in un file che contiene i dati in ordine casuale. Gestite la ricerca sia in base al nome sia in base al numero di telefono, utilizzando una ricerca binaria per entrambe le modalità di consultazione.
- \*\* **E13.9.** Scrivete un programma che misuri le prestazioni dell'algoritmo di ordinamento per inserimento descritto nella sezione Argomenti avanzati 13.2.
- \* **E13.10.** Implementate l'algoritmo di ordinamento a bolle descritto nell'Esercizio R13.21.
- \*\* **E13.11.** Implementate l'algoritmo descritto nel Paragrafo 13.7.4, tenendo però traccia solamente del valore avente la frequenza più elevata fino a quel momento:

```

int mostFrequent = 0;
int highestFrequency = -1;
for (int i = 0; i < a.length; i++)
    Conta le occorrenze di a[i] in a[i + 1] . . . a[n - 1]
    Se ricorre più di highestFrequency volte
        highestFrequency = Quel conteggio
        mostFrequent = a[i]
    }
}

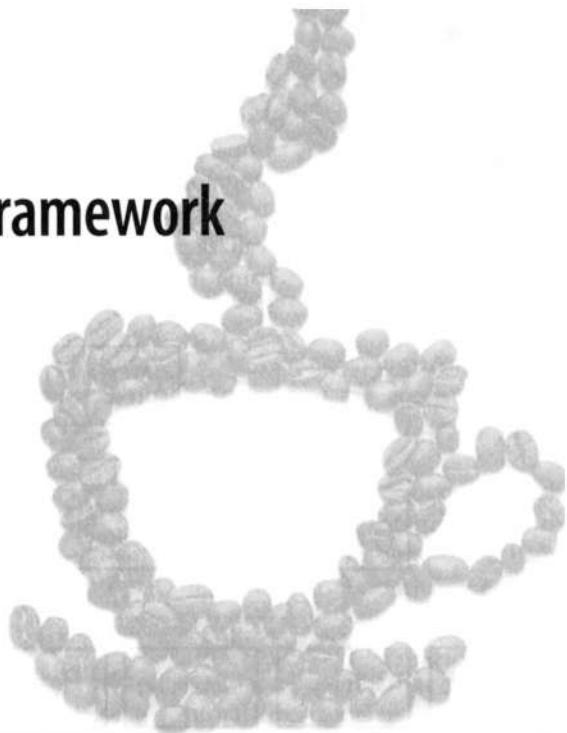
```

- \*\*\* **E13.12.** Scrivete un programma che ordini un esemplare di `ArrayList<Country>` in ordine decrescente, in modo che la nazione di superficie maggiore si trovi all'inizio del vettore. Usate un `Comparator`.
- \*\* **E13.13.** Considerate l'algoritmo di ricerca binaria del Paragrafo 13.6: se non trova l'elemento cercato, il metodo `search` restituisce `-1`. Modificatelo in modo che, se l'elemento non viene trovato, venga invece restituito il valore `-k - 1`, dove `k` è la posizione in cui l'elemento dovrebbe essere inserito (che è il comportamento di `Arrays.binarySearch`).
- \*\* **E13.14.** Realizzate senza ricorsione il metodo `sort` dell'algoritmo di ordinamento per fusione, nell'ipotesi che la lunghezza dell'array sia una potenza di 2. Prima fondete le regioni adiacenti di dimensione 1, poi le regioni adiacenti di dimensione 2, quindi le regioni adiacenti di dimensione 4 e così via.
- \*\*\* **E13.15.** Usate l'ordinamento per inserimento e la ricerca binaria dell'Esercizio E13.13 per ordinare un array secondo quanto descritto nell'Esercizio R13.20. Realizzate tale algoritmo e misuratene le prestazioni.
- \* **E13.16.** Scrivete una classe `Person` che implementi l'interfaccia `Comparable`, confrontando persone in base al loro nome. Chiedete all'utente di inserire dieci nomi e generate dieci oggetti di tipo `Person`. Usando il metodo `compareTo`, determinate la prima e l'ultima persona dell'insieme e visualizzatele.
- \*\* **E13.17.** Ordinate un vettore di stringhe in ordine crescente di *lunghezza*. Suggerimento: progettate un apposito `Comparator`.
- \*\*\* **E13.18.** Ordinate un vettore di stringhe in ordine crescente di lunghezza, in modo che stringhe della stessa lunghezza vengano disposte in ordine lessicografico. Suggerimento: progettate un apposito `Comparator`.

Sul sito web dedicato al libro si trova una raccolta di progetti di programmazione più complessi.

# 14

## Java Collections Framework



### Obiettivi del capitolo

- Imparare a utilizzare le raccolte di dati della libreria standard
- Saper usare gli iteratori per scandire raccolte
- Scegliere la raccolta più adeguata alla soluzione di ciascun problema
- Studiare applicazioni di pile e code

Se scrivete un programma che deve usare un contenitore di oggetti, potete scegliere tra diverse alternative. Ovviamente potete usare un array o un vettore, ma gli scienziati dell'informazione hanno sviluppato molti diversi meccanismi che possono svolgere il compito richiesto in modo migliore. In questo capitolo presenteremo le classi e le interfacce messe a disposizione dalla libreria di Java per descrivere e realizzare raccolte di dati. Imparerete a usarle e a scegliere quella più adeguata a ciascun particolare problema.

## 14.1 Una panoramica del Collections Framework

Una raccolta (*collection*) raggruppa insieme elementi e ne consente il recupero successivo.

Quando in un programma si ha la necessità di organizzare la memorizzazione di più oggetti, li si può inserire in una **raccolta** (*collection*). La classe `ArrayList` presentata nel Capitolo 7 è una delle molte classi che implementano raccolte e che sono messe a disposizione dei programmatore dalla libreria standard di Java. In questo capitolo studierete, in particolare, il *Collections Framework* di Java, una gerarchia di interfacce e classi aventi lo scopo di realizzare contenitori di oggetti. Ogni interfaccia è implementata da una o più classi, come si può vedere nella Figura 1.

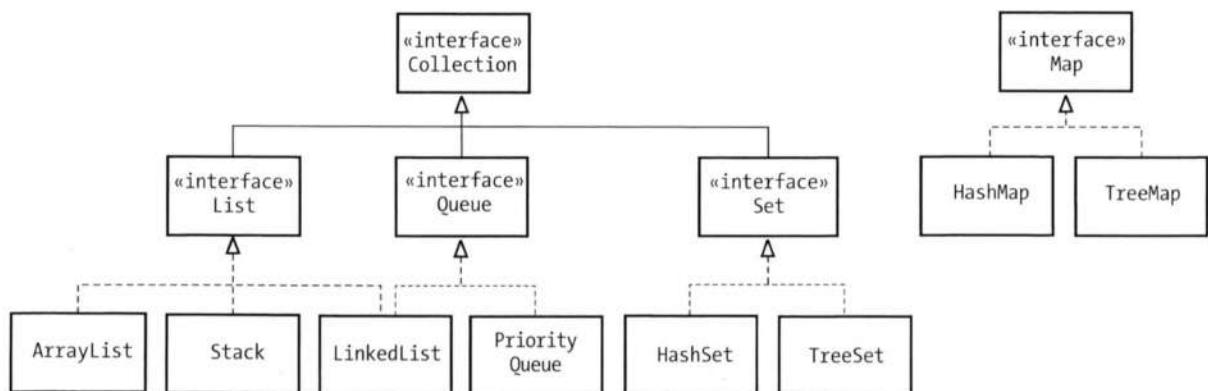


Figura 1 Interfacce e classi del Java Collections Framework

Alla radice della gerarchia si trova l'interfaccia `Collection`, che contiene metodi per aggiungere e rimuovere elementi dalla raccolta, oltre ad altri, come elencato nella Tabella 1. Dato che tutte le raccolte implementano questa interfaccia, i suoi metodi sono disponibili in tutte le classi che rappresentano raccolte di vario tipo. Ad esempio, il metodo `size` restituisce il numero di elementi presenti in *qualsiasi* raccolta.

L'interfaccia `List` descrive un'importante categoria di raccolte. In Java, una *lista* (*list*) è una raccolta che ricorda l'ordine relativo tra i propri elementi (come nella Figura 2). La classe `ArrayList`, ad esempio, implementa l'interfaccia `List`: un oggetto `ArrayList` contiene semplicemente un array che si espande quando serve. Se non siete particolarmente preoccupati dell'efficienza dei vostri programmi, potete usare un `ArrayList` ogni volta che vi serve una raccolta di oggetti, ma molte operazioni abbastanza comuni sono inefficienti quando si usano tali vettori: in particolare, per aggiungere o eliminare un elemento bisogna spostare tutti gli elementi che si trovano in posizioni di indice maggiore.

La libreria di Java mette a disposizione un'altra classe, `LinkedList`, che implementa a sua volta l'interfaccia `List`. Diversamente da un vettore, una *lista concatenata* (in inglese, *linked list*) consente di inserire e rimuovere elementi in posizioni intermedie della lista in modo efficiente: ne parleremo nel prossimo paragrafo.



**Figura 2** Una lista di libri



**Figura 3** Un insieme di libri



**Figura 4** Una pila di libri

Una lista (*list*) è una raccolta che ricorda l'ordine relativo tra i propri elementi.

Un insieme (*set*) è una raccolta non ordinata di elementi non duplicati.

Si usa una lista quando si vuole preservare l'ordine relativo tra gli elementi. Ad esempio, i libri posti su uno scaffale della libreria possono essere ordinati per argomento: una lista è una struttura adeguata per una tale raccolta, perché in questo caso l'ordine tra gli elementi è importante.

In molte applicazioni, però, non siamo affatto interessati all'ordine tra gli elementi di una raccolta. Considerate un venditore di libri per corrispondenza, che gestisce soltanto ordini postali: senza clienti che girano tra gli scaffali, non c'è alcun bisogno di ordinare i libri per argomento. Una raccolta di elementi privi di un ordinamento intrinseco è un **insieme** (*set*, Figura 3).

Dal momento che un insieme non tiene traccia dell'ordine tra i propri elementi, li può disporre in modo da aumentare l'efficienza di operazioni come la ricerca, l'inserimento e la rimozione di elementi, e gli informatici teorici hanno ideato meccanismi adatti a questo. La libreria di Java contiene classi che sono basate su due di tali strategie: le *tabelle hash* e gli *alberi di ricerca binari*. In questo capitolo scoprirete come scegliere una o l'altra.

Un altro modo per aumentare l'efficienza di una raccolta consiste nel ridurre il numero di operazioni che può compiere. Una **pila** (*stack*) ricorda l'ordine tra i propri elementi, ma non consente l'inserimento di nuovi elementi in posizioni arbitrarie: si possono aggiungere elementi soltanto in cima (Figura 4).

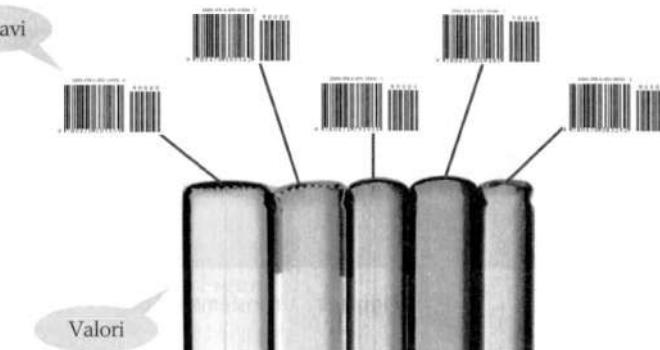
In una **coda** (*queue*) gli elementi si aggiungono alla fine (cioè "in coda", *tail*) e si eliminano all'inizio (*head*). Ad esempio, potete gestire una coda di libri, aggiungendo alla fine della coda i libri che dovete leggere e prendendo il libro che si trova all'inizio della coda ogni volta che avete tempo di iniziare uno. Una **coda prioritaria** (*priority queue*) è una raccolta che non ricorda l'ordine tra i propri elementi (cioè "una raccolta non ordinata") ma esegue in modo molto efficiente l'operazione di eliminazione dell'elemento avente priorità massima. Una coda prioritaria può essere utile per organizzare i vostri impegni di lettura: ogni volta che avete tempo di leggere un libro, estraete dalla raccolta quello avente la massima priorità e leggetelo. Nel Paragrafo 14.5 parleremo di pile, code e code prioritarie.

Infine, una **mappa** (*map*) gestisce associazioni tra **chiavi** (*key*) e **valori** (*value*): ogni chiave della mappa è associata a un valore, come nella Figura 5. La mappa memorizza al proprio interno le chiavi, i valori e le relative associazioni.

Una mappa (*map*) gestisce associazioni tra chiavi e valori.

**Figura 5**

Una mappa che associa a ogni libro (il valore) il relativo codice a barre (la chiave).



Come esempio consideriamo una biblioteca che assegna a ogni libro un codice a barre. Il programma utilizzato per tenere traccia dei libri che escono e rientrano nella biblioteca ha bisogno di cercare il libro associato a un particolare codice a barre e una mappa che associa i codici a barre ai libri può risolvere questo problema. Parleremo di mappe nel Paragrafo 14.4.

In questo capitolo useremo la “sintassi a diamante” per costruire esemplari di classi generiche (si veda la sezione Argomenti avanzati 7.5). Ad esempio, per costruire un vettore di stringhe scriveremo:

```
ArrayList<String> coll = new ArrayList<>();
```

Osservate che dopo `new ArrayList`, nella parte destra, c’è una coppia di parentesi angolari priva di contenuto: il compilatore deduce che si voglia costruire un vettore di stringhe analizzando la parte sinistra dell’enunciato.

**Tabella 1** I metodi principali dell’interfaccia Collection

|                                                                       |                                                                                                                                           |
|-----------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Collection&lt;String&gt; coll = new ArrayList&lt;&gt;();</code> | La classe <code>ArrayList</code> implementa l’interfaccia <code>Collection</code> .                                                       |
| <code>coll = new TreeSet&lt;&gt;();</code>                            | Anche la classe <code>TreeSet</code> (Paragrafo 14.3) implementa l’interfaccia <code>Collection</code> .                                  |
| <code>int n = coll.size();</code>                                     | Restituisce la dimensione della raccolta: ora <code>n</code> vale 0.                                                                      |
| <code>coll.add("Harry");</code>                                       | Aggiunge elementi alla collezione.                                                                                                        |
| <code>coll.add("Sally");</code>                                       |                                                                                                                                           |
| <code>String s = coll.toString();</code>                              | Restituisce una stringa contenente tutti gli elementi della raccolta: ora <code>s</code> è uguale a <code>[Harry, Sally]</code> .         |
| <code>System.out.println(coll);</code>                                | Invoca il metodo <code>toString</code> e visualizza <code>[Harry, Sally]</code> .                                                         |
| <code>coll.remove("Harry");</code>                                    | Elimina un elemento dalla raccolta, restituendo <code>false</code> se l’elemento non è presente; <code>b</code> vale <code>false</code> . |
| <code>boolean b = coll.remove("Tom");</code>                          |                                                                                                                                           |
| <code>b = coll.contains("Sally");</code>                              | Verifica se la raccolta contiene l’elemento cercato; <code>b</code> vale <code>true</code> .                                              |
| <code>for (String s : coll)</code>                                    | Con qualsiasi raccolta si può usare il ciclo <code>for</code> esteso. Questo ciclo visualizza gli elementi su righe distinte.             |
| <code>{</code>                                                        |                                                                                                                                           |
| <code>    System.out.println(s);</code>                               |                                                                                                                                           |
| <code>}</code>                                                        |                                                                                                                                           |
| <code>Iterator&lt;String&gt; iter = coll.iterator();</code>           | Per visitare gli elementi di una raccolta si può usare un iteratore (Paragrafo 14.2.3).                                                   |



## Auto-valutazione

- Un'applicazione che gestisce il registro di classe memorizza una raccolta di valutazioni: dovrebbe usare una lista o un insieme?
- Un sistema informativo studentesco memorizza una raccolta di informazioni relative a ciascuno studente di un ateneo: dovrebbe usare una lista o un insieme?
- Per quale motivo per organizzare i libri che dovete leggere è meglio usare una coda piuttosto che una pila?
- Come potete vedere nella Figura 1, nel Java Collections Framework le mappe non sono raccolte: per quale motivo?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R14.1, R14.2 e R14.3, al termine del capitolo.

## 14.2 Liste concatenate

Una **lista concatenata** (*linked list*) o *catena* è una struttura che memorizza una sequenza di oggetti e che consente di aggiungere e di rimuovere in modo efficiente elementi in qualsiasi posizione, anche intermedia, della sequenza. Nei prossimi paragrafi vedrete come la lista concatenata gestisce i propri elementi e imparerete a usare tale struttura nei vostri programmi.

### 14.2.1 La struttura delle liste concatenate

Per capire per quale motivo gli array siano inefficienti e ci sia bisogno di una struttura più efficiente, immaginate un programma che gestisca una sequenza di oggetti che rappresentano dipendenti, ordinati in base al cognome. Se un dipendente lascia il lavoro, i suoi dati devono essere eliminati e, in un array, il buco che si crea nella sequenza deve essere chiuso spostando tutti i dati che lo seguono. Viceversa, quando viene assunto un nuovo dipendente bisogna inserire un oggetto nella sequenza: il nuovo oggetto deve essere probabilmente inserito nella sequenza in qualche posizione intermedia, per cui tutti gli oggetti che lo seguono devono essere spostati verso la fine della sequenza. Spostare un elevato numero di elementi può comportare un notevole dispendio di tempo di elaborazione: una lista concatenata evita questi spostamenti.

Una lista concatenata è composta da un certo numero di nodi, ciascuno dei quali contiene un riferimento al nodo successivo.

Una lista concatenata usa una sequenza di *nodi*, ciascuno dei quali memorizza un valore e un riferimento al nodo successivo nella sequenza (si veda la Figura 6).

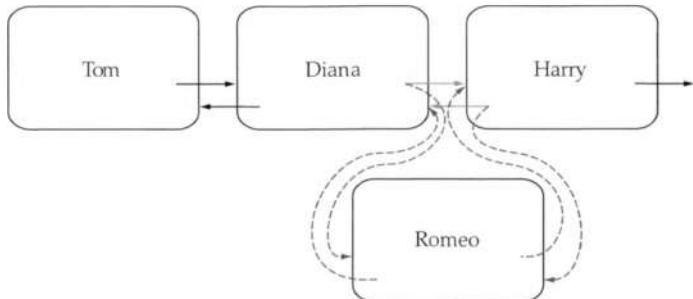
**Figura 6**

Una lista concatenata

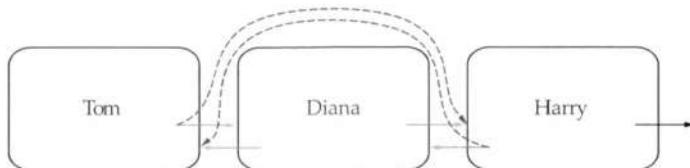


**Figura 7**

Inserimento di un nodo in una lista concatenata

**Figura 8**

Eliminazione di un nodo da una lista concatenata



**Aggiungere e rimuovere elementi in una data posizione di una lista concatenata è un'operazione efficiente.**

Quando viene inserito un nuovo nodo in una lista concatenata, devono essere aggiornati soltanto i riferimenti nei nodi vicini (si veda la Figura 7). Lo stesso accade quando si elimina un nodo (si veda la Figura 8). Dove sta l'insidia? Le liste concatenate consentono di eseguire in modo efficiente inserimenti e rimozioni, ma l'accesso agli elementi può essere poco efficiente.

Se, ad esempio, volete localizzare il quinto elemento, dovete necessariamente visitare i primi quattro: se il vostro algoritmo ha la necessità di accedere agli elementi secondo la modalità di accesso casuale, questo è un problema. Il termine “accesso casuale” (*random access*) viene usato in informatica per descrivere uno schema di accesso in cui gli elementi vengono visitati in ordine arbitrario (non necessariamente casuale); al contrario, con l’accesso sequenziale si visitano gli elementi in sequenza.

Ovviamente, se visitate gli elementi principalmente in successione (per esempio, per visualizzarli o per stamparli), la scarsa efficienza dell’accesso casuale non è un problema. Ricorrete alle liste concatenate quando siete interessati soprattutto all’efficienza degli inserimenti e delle eliminazioni e non avete bisogno di accedere agli elementi in modo casuale.

**Visitare in sequenza gli elementi di una lista concatenata è efficiente, ma accedervi secondo la modalità di accesso casuale non lo è.**

### 14.2.2 La classe `LinkedList` del Java Collections Framework

La libreria di Java mette a disposizione una classe che realizza una lista concatenata: `LinkedList` nel pacchetto `java.util`. È una **classe generica**, proprio come la classe `ArrayList`, per cui il tipo degli elementi contenuti nella lista va specificato tra parentesi angolari, ad esempio `LinkedList<String>` oppure `LinkedList<Employee>`.

La Tabella 2 presenta i metodi principali della classe `LinkedList` (ricordate che la classe `LinkedList` eredita anche i metodi dell’interfaccia `Collection`, elencati nella Tabella 1).

Come potete vedere, ci sono metodi che consentono l’accesso diretto sia al primo sia all’ultimo elemento della lista, mentre per visitare gli altri elementi c’è bisogno di un **iteratore**, di cui parleremo nel prossimo paragrafo.

**Tabella 2** Alcuni metodi di `LinkedList`

|                                                                        |                                                                                                                                                                                                                                                               |
|------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>LinkedList&lt;String&gt; list = new LinkedList&lt;&gt;();</code> | Una lista vuota.                                                                                                                                                                                                                                              |
| <code>list.addLast("Harry");</code>                                    | Aggiunge un elemento alla fine della lista, esattamente come <code>add</code> .                                                                                                                                                                               |
| <code>list.addFirst("Sally");</code>                                   | Aggiunge un elemento all'inizio della lista, dopodiché il contenuto di <code>list</code> è <code>[Sally, Harry]</code> .                                                                                                                                      |
| <code>list.getFirst();</code>                                          | Restituisce l'elemento memorizzato all'inizio della lista, in questo caso <code>"Sally"</code> .                                                                                                                                                              |
| <code>list.getLast();</code>                                           | Restituisce l'elemento memorizzato alla fine della lista, in questo caso <code>"Harry"</code> .                                                                                                                                                               |
| <code>String removed = list.removeFirst();</code>                      | Elimina il primo elemento della lista e lo restituisce, dopodiché il contenuto di <code>list</code> è <code>[Harry]</code> e <code>removed</code> vale <code>"Sally"</code> . Per eliminare l'ultimo elemento si usa, analogamente, <code>removeLast</code> . |
| <code>ListIterator&lt;String&gt; iter = list.listIterator();</code>    | Restituisce un iteratore per visitare tutti gli elementi della lista (per il suo funzionamento si veda la Tabella 3).                                                                                                                                         |

### 14.2.3 Iteratori per liste

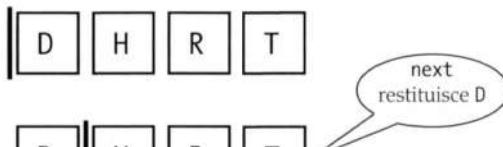
Per accedere agli elementi presenti in una lista concatenata si usa un iteratore (o cursore).

Un **iteratore** o cursore (*iterator*) rappresenta il concetto di posizione in un qualsiasi punto all'interno della lista concatenata. Concettualmente dovete considerare l'iteratore come qualcosa che punta fra due elementi, esattamente come il cursore di un elaboratore di testi punta fra due caratteri (Figura 9). In termini astratti, pensate a ciascun elemento della lista come se fosse un carattere in un elaboratore di testi, mentre l'iteratore è il cursore intermittente che si trova fra due caratteri.

**Figura 9**

Una visione astratta di un iteratore operante su una lista

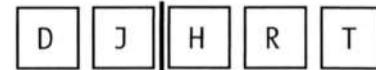
Posizione iniziale di `ListIterator`



Dopo l'invocazione di `next`



Dopo l'inserimento di J



Per creare un iteratore che operi su una lista utilizzate il metodo `listIterator` della classe `LinkedList`:

```
LinkedList<String> employeeNames = . . .;
ListIterator<String> iterator = employeeNames.listIterator();
```

Notate che anche la classe iteratore è una classe generica: un oggetto `ListIterator<String>` scandisce le posizioni all'interno di una lista concatenata di stringhe, mentre un `ListIterator<Book>` visita gli elementi di una `LinkedList<Book>`.

L'iteratore punta inizialmente alla posizione che precede il primo elemento, poi lo potete spostare invocando il suo metodo `next`:

```
iterator.next();
```

Il metodo `next` lancia l'eccezione `NoSuchElementException` se l'iteratore si trova già oltre la fine della lista, per cui dovreste sempre invocare il metodo `hasNext` prima di invocare `next`: restituisce `true` se c'è un elemento successivo.

```
if (iterator.hasNext())
{
    iterator.next();
}
```

Il metodo `next` restituisce l'elemento sopra cui transita l'iteratore durante il suo avanzamento: usando un iteratore di tipo `ListIterator<String>`, il metodo `next` restituisce un riferimento di tipo `String` e, in generale, il tipo di dato restituito dal metodo `next` corrisponde al tipo parametrico usato nella lista (che, a sua volta, ovviamente corrisponde al tipo di oggetti contenuti nella lista stessa).

Potete visitare tutti gli elementi di una lista concatenata di stringhe usando questo ciclo:

```
while (iterator.hasNext())
{
    String name = iterator.next();
    Elabora name
}
```

Se dovete semplicemente visitare tutti gli elementi di una lista concatenata, potete più sinteticamente usare un ciclo `for` esteso:

```
for (String name : employeeNames)
{
    Elabora name
}
```

In questo caso non avete nemmeno bisogno di pensare che esistano gli iteratori: dietro le quinte, il ciclo `for` esteso usa un iteratore per visitare tutti gli elementi della lista.

I nodi della classe `LinkedList` contengono due collegamenti, uno verso l'elemento successivo e uno verso il precedente: una lista di questo genere è detta **lista doppiamente concatenata** (*doubly-linked list*) e potete anche usare i metodi `previous` e `hasPrevious` dell'interfaccia `ListIterator` per spostare l'iteratore all'indietro.

Il metodo `add` aggiunge un oggetto subito dopo la posizione attuale dell'iteratore, quindi sposta la posizione dell'iteratore in modo che si venga a trovare dopo il nuovo elemento.

```
iterator.add("Juliet");
```

Potete visualizzare l'inserimento come se fosse la digitazione di testo in un elaboratore di testi: ciascun carattere viene inserito dopo il cursore e, successivamente, il cursore si sposta in modo da trovarsi dopo il carattere appena inserito (osservate la Figura 9). Gran parte delle persone non ci fa caso: fate una prova e osservate attentamente in che modo il vostro elaboratore di testi inserisce i caratteri.

Il metodo `remove` elimina l'oggetto che era stato restituito dall'ultima invocazione di `next` o `previous`. Per esempio, questo ciclo elimina tutti gli oggetti che soddisfano una determinata condizione:

```
while (iterator.hasNext())
{
    String name = iterator.next();
    if (name soddisfa la condizione)
        iterator.remove();
}
```

Quando invocate `remove` dovete fare molta attenzione, perché può essere invocato una sola volta dopo aver invocato `next` o `previous`, e non lo si può invocare subito dopo aver invocato `add`. Se invocato in modo improprio, il metodo lancia `IllegalStateException`.

La Tabella 3 riassume i metodi dell'interfaccia `ListIterator`, che estende la più generica interfaccia `Iterator`, che è adatta alla scansione di qualunque raccolta, non soltanto una lista. La tabella specifica quali metodi sono disponibili nella sola interfaccia `ListIterator`.

**Tabella 3** Alcuni metodi delle interfacce `Iterator` e `ListIterator`

|                                                                                     |                                                                                                                                                                                                                                                                               |
|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>String s = iter.next();</code>                                                | Ipotizziamo che, prima di questa invocazione di <code>next</code> , <code>iter</code> puntasse all'inizio della lista, il cui contenuto è [Sally]. Dopo l'esecuzione, <code>s</code> vale "Sally" e l'iteratore punta alla fine della lista.                                  |
| <code>iter.previous();</code><br><code>iter.set("Juliet");</code>                   | Il metodo <code>set</code> aggiorna l'ultimo elemento restituito da <code>next</code> o <code>previous</code> , dopodiché il contenuto della lista è [Juliet]. I metodi <code>previous</code> e <code>set</code> sono dichiarati nell'interfaccia <code>ListIterator</code> . |
| <code>iter.hasNext()</code>                                                         | Restituisce <code>false</code> perché l'iteratore si trova alla fine della raccolta..                                                                                                                                                                                         |
| <code>if (iter.hasPrevious())</code><br>{<br><code>s = iter.previous();</code><br>} | Il metodo <code>hasPrevious</code> restituisce <code>true</code> perché l'iteratore non si trova all'inizio della lista. I metodi <code>previous</code> e <code>hasPrevious</code> sono dichiarati nell'interfaccia <code>ListIterator</code> .                               |
| <code>iter.add("Diana");</code>                                                     | Aggiunge un elemento prima della posizione dell'iteratore, dopodiché il contenuto della lista è [Diana, Sally]. Il metodo <code>add</code> è dichiarato nell'interfaccia <code>ListIterator</code> .                                                                          |
| <code>iter.next();</code><br><code>iter.remove();</code>                            | Il metodo <code>remove</code> elimina l'ultimo elemento restituito da <code>next</code> o <code>previous</code> , dopodiché il contenuto della lista è [Sally].                                                                                                               |

Ecco un programma dimostrativo che inserisce stringhe in una lista e, quindi, la percorre iterativamente, aggiungendo e rimuovendo elementi. Alla fine viene visualizzato il contenuto dell'intera lista. I commenti segnalano la posizione dell'iteratore.

### File `ListDemo.java`

```
1 import java.util.LinkedList;
2 import java.util.ListIterator;
3
4 /**
5     Programma che illustra il funzionamento della classe LinkedList.
```

```

6  */
7  public class ListDemo
8 {
9     public static void main(String[] args)
10    {
11        LinkedList<String> staff = new LinkedList<>();
12        staff.addLast("Diana");
13        staff.addLast("Harry");
14        staff.addLast("Romeo");
15        staff.addLast("Tom");
16
17        // il segno | nei commenti indica la posizione dell'iteratore
18
19        ListIterator<String> iterator = staff.listIterator(); // |DHRT
20        iterator.next(); // D|HRT
21        iterator.next(); // DH|RT
22
23        // aggiunge altri elementi dopo il secondo
24
25        iterator.add("Juliet"); // DHJ|RT
26        iterator.add("Nina"); // DHJN|RT
27
28        iterator.next(); // DHJNR|T
29
30        // toglie l'ultimo elemento restituito
31
32        iterator.remove(); // DHJN|T
33
34        // visualizza tutti gli elementi
35
36        System.out.println(staff);
37        System.out.println("Expected: [Diana, Harry, Juliet, Nina, Tom]");
38    }
39 }

```

### Esecuzione del programma

[Diana, Harry, Juliet, Nina, Tom]  
Expected: [Diana, Harry, Juliet, Nina, Tom]



### Auto-valutazione

5. Le liste concatenate occupano più spazio in memoria degli array che contengono lo stesso numero di elementi?

6. Perché con gli array non abbiamo bisogno di iteratori?

7. Nell'ipotesi che la lista concatenata letters contenga inizialmente gli elementi "A", "B", "C" e "D", disegnate il contenuto della lista e la posizione dell'iteratore iter dopo ciascuna delle seguenti operazioni:

```

ListIterator<String> iter = letters.listIterator();
iter.next();
iter.next();
iter.remove();
iter.next();

```

```
iter.add("E");
iter.next();
iter.add("F");
```

8. Scrivete un ciclo che elimini dalla lista concatenata di stringhe `words` tutte le stringhe aventi lunghezza inferiore a quattro.
9. Scrivete un ciclo che visualizzi tutti gli elementi di posizione dispari (contando le posizioni da zero come gli indici, cioè il secondo elemento, il quarto, ecc.) della lista concatenata di stringhe `words`.

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R14.5, R14.8 e E14.1, al termine del capitolo.

## 14.3 Insiemi

Come detto nel Paragrafo 14.1, un **insieme** (*set*) organizza i propri valori in modo da ottimizzare l'efficienza delle operazioni e questo potrebbe non coincidere affatto con l'ordine in cui gli elementi vi sono stati inseriti. L'inserimento e la rimozione di elementi sono operazioni più efficienti in un insieme di quanto non lo siano in una lista concatenata.

Nei paragrafi che seguono imparerete a scegliere una delle implementazioni di insieme disponibili e a scrivere codice che usa insiemi.

### 14.3.1

#### Scegliere un'implementazione di insieme

Le classi HashSet e TreeSet implementano l'interfaccia Set.

Le implementazioni di insieme dispongono gli elementi in modo da poterli ritrovare velocemente.

L'interfaccia `Set` della libreria standard di Java ha gli stessi metodi dell'interfaccia `Collection`, elencati nella Tabella 1, ma tra raccolte generiche e insiemi c'è una differenza essenziale: un insieme non può contenere elementi duplicati. Se aggiungete a un insieme un elemento già presente al suo interno, l'azione di inserimento viene ignorata.

Le classi `HashSet` e `TreeSet` implementano l'interfaccia `Set` e sono basate su due meccanismi diversi: la prima è una **tavella hash** (*hash table*), mentre la seconda è un **albero di ricerca binario** (*binary search tree*). Entrambe le implementazioni dispongono gli elementi al proprio interno in modo da rendere efficienti le operazioni di ricerca, di inserimento e di rimozione di elementi, ma usano due strategie diverse.

L'idea su cui si basano le tabelle hash è semplice. Gli elementi dell'insieme sono raggruppati in raccolte più piccole di elementi che condividono una stessa caratteristica: potete immaginare un insieme di libri come costituito da un gruppo per ciascun possibile colore, in modo che libri dello stesso colore vengano posti nello stesso gruppo. Per scoprire se un determinato libro è presente nell'insieme, è sufficiente verificare la sua presenza tra i libri che appartengono al gruppo corrispondente al suo colore. Le tabelle hash non usano proprio i colori, ma numeri interi (chiamati *codici di hash*) che possono essere calcolati a partire dagli elementi.



### Standardizzazione

Ogni giorno potete constatare i vantaggi della standardizzazione. Quando comprate una lampadina, potete essere certi che si avvierà correttamente nel portalampada senza averlo misurato a casa e senza misurare la lampadina nel negozio. Al contrario, potete aver conosciuto la scomodità della mancanza di standard se avete comprato una torcia elettrica con una lampadina non standard: le lampadine di ricambio per tale torcia sono solitamente difficili da trovare e costose.

I programmati hanno la stessa voglia di standardizzazione. Consideriamo l'importante obiettivo dell'indipendenza dalla piattaforma per i programmi Java: dopo aver compilato un programma Java, potete eseguire il risultato prodotto dalla compilazione in qualsiasi computer che disponga di una macchina virtuale Java. Perché questo possa accadere, la macchina virtuale Java deve essere definita in modo molto preciso: se tutte le macchine virtuali non si comportassero esattamente allo stesso modo, il motto "scrivi una volta, esegui dappertutto" (*write once, run anywhere*) si tramuterebbe in "scrivi una volta, collauda dappertutto". Perché diversi gruppi di programmati possano realizzare macchine virtuali compatibili, la macchina virtuale deve essere *standardizzata*, cioè c'è bisogno di qualcuno che dia una definizione della macchina virtuale e del suo comportamento previsto.

Chi crea gli standard? Alcuni degli standard di maggiore successo sono stati creati da gruppi di volon-

tari, come Internet Engineering Task Force (IETF) e World Wide Web Consortium (W3C). IETF si occupa di rendere standard molti protocolli di Internet, come quello che riguarda lo scambio di messaggi di posta elettronica, mentre W3C definisce lo standard per il linguaggio HTML (Hypertext Markup Language), che descrive il formato delle pagine Web. Questi standard sono stati fondamentali per la creazione del World Wide Web come una piattaforma aperta, non controllata da nessuna azienda.

Molti linguaggi di programmazione, come C++ e Scheme, sono stati standardizzati da organizzazioni indipendenti, come ANSI (American National Standards Institute) e ISO (International Organization for Standardization). ANSI e ISO sono associazioni di professionisti dell'industria che sviluppano standard per qualsiasi cosa, dalle dimensioni e forme di pneumatici e carte di credito ai linguaggi di programmazione.

Molti standard sono stati sviluppati da esperti che lavorano in molte aziende diverse, oltre che da progettisti privati, con l'obiettivo di creare un insieme di regole che codifichino le strategie migliori, ma a volte gli standard sono frutto di lunghi contenzi. Nel 2005 Microsoft iniziò a perdere contratti governativi perché i suoi clienti si preoccupavano del fatto che molti dei loro documenti venivano archiviati usando formati proprietari e non descritti pubblicamente. Invece di fornire conseguentemente supporto a formati standard già esistenti o collaborare con un gruppo di lavoro per migliorarli, Microsoft

scrisse un proprio standard che codificava semplicemente ciò che i suoi prodotti facevano in quel momento, anche se tale formato era diffusamente ritenuto incoerente e troppo complesso (la sua descrizione occupava circa 6000 pagine). Inizialmente l'azienda sottopose lo standard a ECMA (European Computer Manufacturers Association), che lo approvò dopo una discussione assai contenuta, dopodiché ISO lo rese operativo come "standard già esistente", evitando di sottoporlo alla consueta procedura di revisione tecnica.

Per motivi analoghi, Sun Microsystems, dopo aver inventato Java, non acconsentì mai a una sua standardizzazione da parte di un'organizzazione terza, bensì iniziò una propria procedura, coinvolgendo altre aziende ma rifiutandosi di perdere il controllo sullo standard stesso.

Ovviamente molte importanti tecnologie non sono affatto standardizzate. Si consideri il sistema operativo Windows: sebbene venga definito come uno standard di fatto (*de facto*), non è affatto uno standard. Nessuno ha mai tentato di definire formalmente cosa dovrebbe essere il sistema operativo Windows, il cui comportamento cambia secondo il volere del suo produttore: ciò è perfetto per Microsoft, perché rende impossibile per altri creare la propria versione di Windows.

Nella vostra carriera di informatici ci saranno molte occasioni in cui dovrete prendere una decisione sull'aderenza a un particolare standard. Facciamo un semplice esempio: in questo capitolo abbiamo

usato le raccolte di elementi definite nella libreria standard di Java, ma a molti informatici queste classi non piacciono, per molti motivi legati a problemi di progettazione. Dovreste

usare queste classi nel vostro codice o sarebbe meglio realizzare un insieme di raccolte migliori? Se scegliete la prima opportunità dovete fare i conti con una realizzazione non ottimale,

mentre nel secondo caso altri programmati potrebbero avere difficoltà nella comprensione del vostro codice, perché non hanno familiarità con le vostre classi.

Per poter usare una tabella hash gli elementi devono disporre di un metodo, che si chiama `hashCode`, che calcoli tali numeri interi. Inoltre, gli elementi devono appartenere a una classe che abbia definito in modo appropriato il metodo `equals` (si veda il Paragrafo 9.5.2).

Molte classi della libreria standard, tra cui `String`, `Integer`, `Double`, `Point`, `Rectangle`, `Color` e tutte le classi di tipo raccolta implementano tali metodi. Di conseguenza, potete certamente costruire oggetti di tipo `HashSet<String>`, `HashSet<Rectangle>` o anche `HashSet<HashSet<Integer>>`.

Se volete usare un insieme di elementi che siano esemplari di una classe progettata da voi, ad esempio un insieme di libri di tipo `HashSet<Book>`, dovete definire, nella classe `Book`, i metodi `hashCode` e `equals`. C'è, però, un'eccezione a questa regola: se tutti gli elementi sono distinti (ad esempio, se il programma non userà mai due oggetti `Book` che rappresentino libri con lo stesso titolo e lo stesso autore), allora potete semplicemente ereditare i metodi `hashCode` e `equals` della classe `Object`.

La classe `TreeSet` usa una strategia differente per disporre gli elementi al proprio interno: li tiene ordinati. Ad esempio, in un insieme di libri questi potrebbero essere ordinati in base all'altezza, oppure in ordine alfabetico per autore o titolo. Gli elementi non vengono memorizzati in un array, perché questo renderebbe troppo inefficiente l'inserimento e la rimozione di elementi: li memorizza in nodi, come in una lista concatenata. I nodi, però, non vengono disposti in una sequenza lineare, bensì in una struttura a forma di albero.

Per poter usare un `TreeSet`, deve essere possibile confrontare gli elementi tra loro, per determinare quale sia il più "grande". Si può, quindi, usare un `TreeSet` per contenere oggetti di tipo `String` o `Integer`, cioè classi che implementano l'interfaccia `Comparable`, di cui abbiamo parlato nel paragrafo 10.3 (dove abbiamo anche discusso come si possa realizzare il metodo di confronto anche in classi progettate autonomamente).

Come regola di base, usate un insieme di tipo `TreeSet` quando avete la necessità di visitare gli elementi dell'insieme in ordine, altrimenti usate un `HashSet`: se la funzione di hash è stata progettata bene, l'insieme risulta un po' più efficiente.

Quando costruite un insieme di tipo `HashSet` o `TreeSet`, memorizzatene il riferimento in una variabile di tipo `Set`, così

```
Set<String> names = new HashSet<>();
```

oppure così

```
Set<String> names = new TreeSet<>();
```

Dopo aver costruito l'oggetto che funge da raccolta, non importa più sapere quale realizzazione dell'insieme venga usata, ci serve solo conoscere la sua interfaccia.

Si possono usare insiemi realizzati mediante tabella hash quando devono, ad esempio, contenere oggetti di tipo `String`, `Integer`, `Double`, `Point`, `Rectangle` o `Color`.

Si possono usare insiemi realizzati mediante albero di ricerca binario con elementi di qualunque classe che implementi l'interfaccia `Comparable`, come `String` o `Integer`.

### 14.3.2 Lavorare con insiemi

Per aggiungere e rimuovere elementi in un insieme si usano i metodi `add` e `remove`:

```
names.add("Romeo");
names.remove("Juliet");
```

Gli insiemi non contengono elementi duplicati. L'aggiunta all'insieme di un elemento duplicato di un altro elemento già presente nell'insieme viene semplicemente ignorata.

Come in matematica, in Java una raccolta di tipo insieme rifiuta l'inserimento di elementi duplicati. L'aggiunta di un elemento non ha alcun effetto se l'elemento è già presente nell'insieme. Analogamente, il tentativo di rimozione di un elemento che non appartiene all'insieme viene ignorato.

Il metodo `contains` verifica se un elemento appartiene all'insieme:

```
if (names.contains("Juliet")) . . .
```

Il metodo `contains` fa i confronti usando il metodo `equals` dell'elemento. Se l'insieme contiene oggetti di tipo `String` o `Integer` non c'è da preoccuparsi, perché tali classi mettono a disposizione un metodo `equals` adeguato, ma se il tipo dell'elemento è una classe realizzata in proprio, questa deve aver definito il metodo `equals`, come descritto nel Paragrafo 9.5.2.

Infine, per elencare tutti gli elementi presenti nell'insieme si usa un iteratore. Come per gli iteratori delle liste concatenate, si usano i metodi `next` e `hasNext` per scandire l'insieme, un elemento per volta.

```
Iterator<String> iter = names.iterator();
while (iter.hasNext())
{
    String name = iter.next();
    Elabora name
}
```

Invece di usare esplicitamente un iteratore si può usare un ciclo `for` esteso:

```
for(String name : names)
{
    Elabora name
}
```

Un iteratore visita gli elementi di un insieme nell'ordine in cui questi sono memorizzati all'interno dell'insieme stesso, in relazione al tipo di implementazione.

Non si può aggiungere un elemento a un insieme usando una specifica posizione di un iteratore.

Un iteratore visita gli elementi di un insieme nell'ordine in cui questi sono memorizzati all'interno dell'insieme e questo non coincide necessariamente con l'ordine in cui gli elementi vi sono stati inseriti. In un insieme realizzato mediante tabella hash gli elementi vengono visitati in ordine apparentemente casuale, perché il codice di hash li distribuisce tra i diversi gruppi in tal modo; visitando, invece, gli elementi di un insieme realizzato mediante albero di ricerca binario, questi appaiono in ordine, anche se sono stati inseriti in ordine diverso.

C'è una differenza importante tra l'oggetto di tipo `Iterator` che viene fornito da un insieme e quello di tipo `ListIterator` restituito da una lista. L'oggetto `ListIterator` ha un metodo `add` per aggiungere un elemento alla lista nella posizione in cui si trova l'iteratore, mentre l'interfaccia `Iterator` non ha tale metodo: non ha senso aggiungere un elemento in una specifica posizione di un insieme, perché l'insieme può ordinare gli elementi al proprio interno come preferisce. Quindi, si aggiunge sempre un elemento direttamente all'insieme, mai a un iteratore dell'insieme.

Tuttavia, è possibile eliminare dall'insieme l'elemento che si trova nella posizione dell'iteratore, proprio come si fa con gli iteratori di lista.

Ancora, l'interfaccia `Iterator` non ha il metodo `previous` per tornare indietro nella scansione degli elementi: dato che gli elementi non sono ordinati, non ha senso distinguere tra "andare avanti" e "tornare indietro".

**Tabella 4** Lavorare con insiemi

|                                             |                                                                                                                                        |
|---------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| <code>Set&lt;String&gt; names;</code>       | Per dichiarare variabili usate il tipo interfaccia.                                                                                    |
| <code>names = new HashSet&lt;&gt;();</code> | Se dovete visitare gli elementi dell'insieme in ordine usate <code>TreeSet</code> .                                                    |
| <code>names.add("Romeo");</code>            | Ora <code>names.size()</code> vale 1.                                                                                                  |
| <code>names.add("Fred");</code>             | Ora <code>names.size()</code> vale 2.                                                                                                  |
| <code>names.add("Romeo");</code>            | <code>names.size()</code> vale ancora 2: non vengono aggiunti duplicati.                                                               |
| <code>if (names.contains("Fred"))</code>    | Il metodo <code>contains</code> verifica se un valore appartiene all'insieme. In questo caso il metodo restituisce <code>true</code> . |
| <code>System.out.println(names);</code>     | Visualizza l'insieme nel formato [Fred, Romeo]. Gli elementi non compaiono necessariamente nell'ordine in cui sono stati inseriti.     |
| <code>for (String name : names)</code>      | Usate questo ciclo per visitare tutti gli elementi dell'insieme.                                                                       |
| <code>{</code>                              |                                                                                                                                        |
| <code>    ...</code>                        |                                                                                                                                        |
| <code>}</code>                              |                                                                                                                                        |
| <code>names.remove("Romeo");</code>         | Ora <code>names.size()</code> vale 1.                                                                                                  |
| <code>names.remove("Juliet");</code>        | Rimuovere un elemento che non è presente non è un errore: l'invocazione del metodo non ha nessun effetto.                              |

Il programma seguente mostra un'applicazione degli insiemi. Legge tutte le parole di un dizionario, contenuto in un file, e le inserisce in un insieme; poi, legge tutte le parole di un documento (in questo caso, il libro "Alice in Wonderland", *Alice nel paese delle meraviglie*), inserendole in un secondo insieme; infine, visualizza tutte le parole di tale secondo insieme che non figurano nell'insieme contenente il dizionario delle parole corrette, identificando in questo modo le parole potenzialmente scritte in modo errato (come potrete vedere, usiamo un dizionario americano, quindi alcune parole inglesi, come *clamour*, vengono segnalate come potenziali errori).

### File SpellCheck.java

```

1 import java.util.HashSet;
2 import java.util.Scanner;
3 import java.util.Set;
4 import java.io.File;
5 import java.io.FileNotFoundException;
6
7 /**
8      Verifica quali parole di un file non sono presenti in un dizionario.
9 */
10 public class SpellCheck
11 {

```

```
12  public static void main(String[] args)
13      throws FileNotFoundException
14  {
15      // Legge il dizionario e il documento
16
17      Set<String> dictionaryWords = readWords("words");
18      Set<String> documentWords = readWords("alice30.txt");
19
20      // Visualizza le parole presenti nel documento ma non nel dizionario
21
22      for (String word : documentWords)
23      {
24          if (!dictionaryWords.contains(word))
25          {
26              System.out.println(word);
27          }
28      }
29  }
30
31 /**
32  Legge tutte le parole contenute in un file.
33  @param filename il nome del file
34  @return un insieme con tutte le parole del file, convertite in minuscolo;
35  una parola è una sequenza di lettere, maiuscole o minuscole.
36 */
37 public static Set<String> readWords(String filename)
38     throws FileNotFoundException
39 {
40     Set<String> words = new HashSet<>();
41     Scanner in = new Scanner(new File(filename));
42     // delimitatore: carattere che non appartiene all'intervallo a-z né A-Z
43     in.useDelimiter("[^a-zA-Z]+");
44     while (in.hasNext())
45     {
46         words.add(in.next().toLowerCase());
47     }
48     return words;
49 }
50 }
```

### Esecuzione del programma

```
neighbouring
croqueted
pennyworth
dutchess
comfits
xii
dinn
clamour
...
...
```



## Auto-valutazione

10. Gli array e le liste memorizzano l'ordine in cui vengono aggiunti gli elementi, mentre gli insiemi non lo fanno. Per quale motivo in alcune situazioni si preferisce utilizzare insiemi invece di array o liste?
11. Perché gli iteratori per insiemi sono diversi dagli iteratori per liste?
12. Cosa c'è di sbagliato nel codice seguente che vorrebbe verificare se l'insieme `Set<String> s` contiene gli elementi "Tom", "Diana" e "Harry"?
 

```
if (s.toString().equals("[Tom, Diana, Harry]")) . . .
```
13. Come realizzereste correttamente la verifica descritta nella domanda precedente?
14. Scrivete un ciclo che visualizzi tutti gli elementi che si trovano sia in `Set<String> s` sia in `Set<String> t`.
15. Modificando la riga 40 del programma `SpellCheck` in modo che usi un `TreeSet` invece di un `HashSet`, come cambiano le informazioni che vengono visualizzate?

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi E14.3, E14.12 e E14.13, al termine del capitolo.



## Suggerimenti per la programmazione 14.1

### Usare riferimenti a interfaccia per manipolare strutture dati

Memorizzare in una variabile di tipo `Set` un riferimento a un oggetto di tipo `HashSet` o `TreeSet` è considerato un buono stile di programmazione.

```
Set<String> names = new HashSet<>();
```

In questo modo, se deciderete in seguito di usare un oggetto di tipo `TreeSet`, dovrete modificare soltanto una riga di codice.

Se un metodo può elaborare una raccolta di qualsiasi tipo, definite la sua variabile parametro di tipo `Collection`:

```
public static void removeLongWords(Collection<String> words)
```

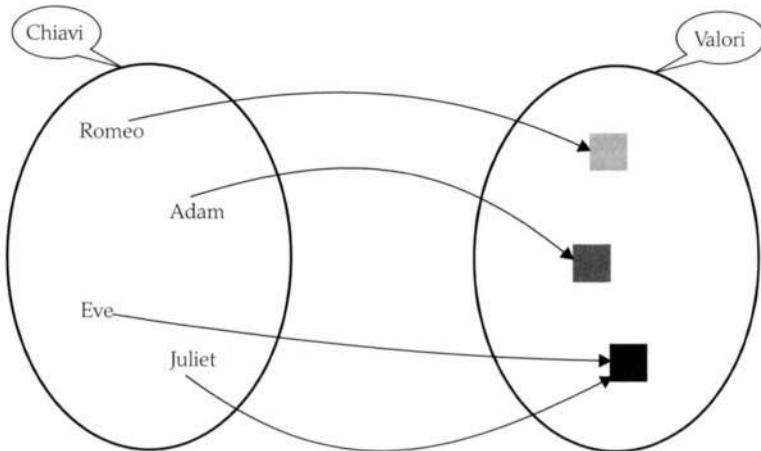
In teoria dovremmo fare la stessa raccomandazione per l'interfaccia `List`, suggerendo di memorizzare in variabili di tipo `List` i riferimenti a oggetti `LinkedList` o `ArrayList`. Tuttavia, nella libreria di Java l'interfaccia `List` ha i metodi `get` e `set` per l'accesso casuale, nonostante questi metodi siano molto inefficienti per le liste concatenate. Non potete scrivere codice efficiente se non sapete se i metodi che state invocando sono efficienti oppure no. Questo è evidentemente un serio errore di progettazione della libreria standard e per tale motivo non si può raccomandare l'uso dell'interfaccia `List`.

## 14.4 Mappe

Le classi `HashMap` e `TreeMap` implementano l'interfaccia `Map`.

Una **mappa** (*map*) memorizza associazioni tra un insieme di chiavi (*key*) e una raccolta di valori (*value*) e si usa quando si vogliono cercare oggetti (i valori) usando una chiave. La Figura 10 mostra una mappa che associa nomi di persone al loro colore preferito.

**Figura 10**  
Una mappa



Così come ci sono due realizzazioni diverse di insieme, la libreria di Java fornisce due implementazioni anche per l'interfaccia Map: `HashMap` e `TreeMap`.

Dopo aver costruito un oggetto di tipo `HashMap` o `TreeMap`, dovreste memorizzarne il riferimento in una variabile di tipo `Map`:

```
Map<String, Color> favoriteColors = new HashMap<>();
```

Per aggiungere un'associazione alla mappa si usa il metodo `put`:

```
favoriteColors.put("Juliet", Color.RED);
```

Potete modificare il valore di un'associazione già esistente semplicemente invocando di nuovo il metodo `put`:

```
favoriteColors.put("Juliet", Color.BLUE);
```

Il metodo `get` restituisce il valore associato a una chiave:

```
Color juliet'sFavoriteColor = favoriteColors.get("Juliet");
```

Se chiedete informazioni su una chiave che non è associata ad alcun valore, il metodo `get` restituisce `null`.

Per eliminare una chiave e il valore associato si invoca il metodo `remove` con la chiave:

```
favoriteColors.remove("Juliet");
```

Per trovare tutte le chiavi e i valori presenti in una mappa, si scandisce l'insieme delle chiavi e si cerca il valore corrispondente a ciascuna chiave.

A volte capita di voler esaminare tutte le chiavi di una mappa, una dopo l'altra. Il metodo `keySet` restituisce un insieme contenente le chiavi, dopodiché potete chiedere un iteratore a tale insieme e, da questo, ottenere tutte le chiavi, una dopo l'altra. Infine, per ogni chiave si può invocare il metodo `get`, ottenendo il valore associato. Quindi, le istruzioni seguenti visualizzano tutte le coppie chiave/valore presenti nella mappa `m`:

**Tabella 5** Lavorare con mappe

|                                                   |                                                                                                                                                                                   |
|---------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Map&lt;String, Integer&gt; scores;</code>   | Qui le chiavi sono stringhe e i valori sono involucri di tipo <code>Integer</code> . Per dichiarare variabili si usa il tipo interfaccia.                                         |
| <code>scores = new TreeMap&lt;&gt;();</code>      | Se non avete bisogno di visitare le chiavi della mappa in ordine usate <code>HashMap</code> .                                                                                     |
| <code>scores.put("Harry", 90);</code>             | Aggiunge chiavi e valori alla mappa.                                                                                                                                              |
| <code>scores.put("Sally", 95);</code>             |                                                                                                                                                                                   |
| <code>scores.put("Sally", 100);</code>            | Modifica il valore associato a una chiave esistente.                                                                                                                              |
| <code>int n = scores.get("Sally");</code>         | Ispeziona il valore associato a una chiave, ottenendo <code>null</code> se la chiave non è presente. Alla fine <code>n</code> vale 100 e <code>n2</code> vale <code>null</code> . |
| <code>System.out.println(scores);</code>          | Visualizza <code>scores.toString()</code> , in questo caso la stringa <code>{Harry=90, Sally=100}</code> .                                                                        |
| <code>for (String key : scores.keySet())</code>   | Il ciclo scandisce tutte le chiavi della mappa e nel suo corpo si ispezionano tutti i valori corrispondenti.                                                                      |
| <code>{</code>                                    |                                                                                                                                                                                   |
| <code>    Integer value = scores.get(key);</code> |                                                                                                                                                                                   |
| <code>    . . .</code>                            |                                                                                                                                                                                   |
| <code>}</code>                                    |                                                                                                                                                                                   |
| <code>scores.remove("Sally");</code>              | Elimina la chiave e il suo valore.                                                                                                                                                |

```
Set<String> keySet = m.keySet();
for (String key : keySet)
{
    Color value = m.get(key);
    System.out.println(key + " : " + value);
}
```

Il programma che segue mostra una mappa in azione.

### File MapDemo.java

```
1 import java.awt.Color;
2 import java.util.HashMap;
3 import java.util.Map;
4 import java.util.Set;
5
6 /**
7     Questo programma collauda una mappa che associa nomi a colori.
8 */
9 public class MapDemo
{
10     public static void main(String[] args)
11     {
12         Map<String, Color> favoriteColors = new HashMap<>();
13         favoriteColors.put("Juliet", Color.BLUE);
14         favoriteColors.put("Romeo", Color.GREEN);
15         favoriteColors.put("Adam", Color.RED);
16         favoriteColors.put("Eve", Color.BLUE);
17
18         // visualizza tutte le chiavi e i valori presenti nella mappa
19
20         Set<String> keySet = favoriteColors.keySet();
21         for (String key : keySet)
```

```

23     {
24         Color value = favoriteColors.get(key);
25         System.out.println(key + " : " + value);
26     }
27 }
28 }
```

### Esecuzione del programma

```

Adam : java.awt.Color[r=255,g=0,b=0]
Eve : java.awt.Color[r=0,g=0,b=255]
Juliet : java.awt.Color[r=0,g=0,b=255]
Romeo : java.awt.Color[r=0,g=255,b=0]
```



### Auto-valutazione

16. Che differenza c'è tra un insieme e una mappa?
17. Perché la raccolta delle chiavi di una mappa è un insieme e non una lista?
18. Perché la raccolta dei valori di una mappa non è un insieme?
19. Supponete di voler tenere traccia del numero di ripetizioni di ciascuna parola all'interno di un documento. Dichiarate una variabile con il tipo di mappa adatta.
20. Che tipo di oggetto è un `Map<String, HashSet<String>>`? Descrivete un possibile utilizzo di una tale struttura.

### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R14.20, E14.4 e E14.5, al termine del capitolo.



### Note per Java 8 14.1

#### Aggiornare le associazioni in una mappa

Le mappe vengono spesso utilizzate per contare le ripetizioni di vari dati. Ad esempio, la sezione Esempi completi 14.1 usa un oggetto di tipo `Map<String, Integer>` per tenere traccia del numero di ripetizioni di ciascuna parola in un file.

La gestione del caso speciale relativo all'inserimento del primo valore di ciascuna coppia è un po' noiosa. Consideriamo il frammento di codice seguente, preso dalla sezione Esempi completi 14.1:

```

Integer count = frequencies.get(word); // recupera il conteggio attuale
// se word non c'era, poni il conteggio a 1, altrimenti incrementalo
if (count == null) { count = 1; }
else { count = count + 1; }
frequencies.put(word, count);
```

Java 8 ha aggiunto all'interfaccia `Map` l'utile metodo `merge`, al quale occorre specificare:

- una chiave;
- un valore da usare se la chiave non è ancora presente;
- una funzione per calcolare il valore aggiornato se la chiave è già presente.

La funzione va specificata sotto forma di espressione lambda (si veda la sezione Note per Java 8 10.4). Ad esempio, l'enunciato seguente

```
frequencies.merge(word, 1, (oldValue, notPresentValue) -> oldValue + 1);
```

completa le medesime azioni delle quattro righe di codice precedenti. Se la stringa `word` non è presente nella mappa, vi viene inserita, associata al valore 1, altrimenti il suo vecchio valore (`oldValue`) viene incrementato di un'unità.

Il metodo è utile anche quando i valori della mappa sono insiemi o stringhe con componenti separate da virgole, come negli Esercizi E14.6 e E14.7.



## Consigli pratici 14.1

### Scegliere una raccolta

Immaginate di dover memorizzare oggetti in una raccolta; avete già visto un certo numero di diverse strutture per i dati e questi “Consigli pratici” riassumono come si possa scegliere la raccolta più appropriata per una determinata applicazione.

#### Fase 1. Determinate le modalità di accesso agli elementi

Gli elementi vengono memorizzati in un contenitore per poterli recuperare in seguito. Come volete accedere ai singoli elementi? Ci sono diverse possibilità.

- Si accede agli elementi mediante una posizione, rappresentata da un numero intero: usate un `ArrayList`.
- Si accede agli elementi mediante una chiave che non fa parte dell'oggetto: usate una mappa.
- Non avete bisogno di accedere ai singoli elementi sulla base della loro posizione: prenderete una decisione nelle Fasi 3 e 4.

#### Fase 2. Determinate il tipo degli elementi o i tipi di chiavi e valori

In una lista o in un insieme determinate il tipo degli elementi che volete memorizzare. Ad esempio, se volete rappresentare un insieme di libri, il tipo degli elementi potrebbe essere `Book`.

Analogamente, nel caso di una mappa determinate il tipo delle chiavi e il tipo dei valori a esse associati. Se volete cercare libri in base a un codice identificativo (ID), potete usare mappe di tipo `Map<Integer, Book>` oppure `Map<String, Book>`, in relazione al tipo di ID che intendete usare.

#### Fase 3. Determinate se l'ordine degli elementi o delle chiavi è importante

Quando visitate gli elementi della raccolta o le chiavi della mappa, vi interessa l'ordine con il quale avviene l'operazione? Ci sono diverse possibilità.

- Gli elementi o le chiavi devono essere in ordine: usate `TreeSet` o `TreeMap` e passate alla Fase 6.

- Gli elementi devono essere nello stesso ordine in cui sono stati inseriti: la scelta si restringe a `LinkedList` o `ArrayList`.
- Non importa. A condizione di poter visitare tutti gli elementi, non vi importa in quale ordine questo avviene. Se nella Fase 1 avete scelto di usare una mappa, usate `HashMap` e passate alla Fase 5.

**Fase 4.** Determinate quali operazioni devono essere efficienti

Ci sono diverse possibilità.

- La ricerca di elementi deve essere efficiente: usate `HashSet`.
- Devono essere veloci l'inserimento e la rimozione di elementi nella posizione iniziale o in posizioni intermedie, nelle quali si è già ispezionato l'elemento: usate `LinkedList`.
- Effettuate inserimenti soltanto nella posizione terminale e usate così pochi elementi che la velocità non vi interessa: usate `ArrayList`.

**Fase 5.** Per insiemi e mappe con tabella hash, decidete se occorre realizzare i metodi `equals` e `hashCode`

- Se i vostri elementi (o chiavi) appartengono a una classe realizzata da altri, controllate se ha i propri metodi `hashCode` e `equals`: se è così, siete a posto. Questo accade per la maggior parte delle classi della libreria standard di Java, come `String`, `Integer`, `Rectangle` e così via.
- In caso contrario, decidete se potete confrontare gli elementi per identità, cosa che avviene se non costruire mai due elementi distinti che abbiano lo stesso contenuto. In tal caso non avete bisogno di far nulla: i metodi `hashCode` e `equals` della classe `Object` sono adeguati.
- Altrimenti dovete realizzare vostri metodi `hashCode` e `equals`, quindi consultate il Paragrafo 9.5.2 e la sezione Argomenti avanzati 14.1.

**Fase 6.** Se usate un albero, decidete se serve un comparatore

Osservate la classe degli elementi dell'insieme o delle chiavi della mappa: implementa l'interfaccia `Comparable`? In tal caso, se l'ordinamento indotto dal metodo `compareTo` è quello che volete, allora non avete bisogno di far nulla. Questo accade per molte classi della libreria standard, in particolare per `String` e `Integer`.

In caso contrario, la classe dei vostri elementi deve implementare l'interfaccia `Comparable` (come descritto nel Paragrafo 10.3) oppure dovete definire una classe che implementi l'interfaccia `Comparator`, come visto nella sezione Argomenti avanzati 13.4.



---

## Esempi completi 14.1

### Determinare la frequenza di parole in un testo

**Problema.** Scrivere un programma che legge un file di testo e visualizza un elenco contenente, in ordine alfabetico, tutte le parole presenti nel file, seguite da un conteggio che indica il numero di ripetizioni di ciascuna parola.

Ecco, ad esempio, la parte iniziale del risultato prodotto dall'elaborazione del libro "Alice in Wonderland" (*Alice nel paese delle meraviglie*):

|         |     |
|---------|-----|
| a       | 653 |
| abide   | 1   |
| able    | 1   |
| about   | 97  |
| above   | 4   |
| absence | 1   |
| absurd  | 2   |

**Fase 1.** Determinate le modalità di accesso agli elementi

In questo caso i valori sono le occorrenze delle parole e abbiamo un valore per ciascuna parola, per cui vogliamo usare una mappa che gestisca il conteggio associato a una parola.

**Fase 2.** Determinate il tipo degli elementi o i tipi di chiavi e valori

Le parole sono di tipo `String` e i conteggi sono `Integer` (non possiamo usare `int` come tipo parametrico perché è un tipo primitivo), quindi ci serve una mappa di tipo `Map<String, Integer>`.

**Fase 3.** Determinate se l'ordine degli elementi o delle chiavi è importante

Ci viene chiesto di visualizzare le parole in ordine alfabetico, quindi useremo `TreeMap`.

**Fase 4.** Determinate quali operazioni devono essere efficienti

Saltiamo questa fase, perché abbiamo già deciso di usare una mappa.

**Fase 5.** Per insiemi e mappe con tabella hash, decidete se occorre realizzare i metodi `equals` e `hashCode`

Saltiamo questa fase, perché abbiamo già deciso di usare `TreeMap`.

**Fase 6.** Se usate un albero, decidete se serve un comparatore

Il tipo delle chiavi della nostra mappa è `String`, che implementa l'interfaccia `Comparable`, per cui non c'è altro da fare.

Abbiamo quindi scelto la nostra struttura di memorizzazione dei dati. Lo pseudocodice per descrivere la soluzione del problema è veramente semplice:

Per ogni parola presente nel file d'ingresso  
    Elimina dalla parola i caratteri che non sono lettere.  
    Se la parola è già presente nella mappa dei conteggi  
        Incrementa il conteggio associato.  
    Altrimenti  
        Imposta il conteggio al valore 1.

Ecco il codice del programma.

## File WordFrequency.java

```

1 import java.util.Map;
2 import java.util.Scanner;
3 import java.util.TreeMap;
4 import java.io.File;
5 import java.io.FileNotFoundException;
6
7 /**
8  * Visualizza il numero di ripetizioni di tutte le parole in "Alice in Wonderland".
9 */
10 public class WordFrequency
11 {
12     public static void main(String[] args)
13         throws FileNotFoundException
14     {
15         Map<String, Integer> frequencies = new TreeMap<>();
16         Scanner in = new Scanner(new File("alice30.txt"));
17         while (in.hasNext())
18         {
19             String word = clean(in.next());
20
21             // ispeziona il conteggio attuale della parola
22
23             Integer count = frequencies.get(word);
24
25             // se la parola non c'era usa 1, altrimenti incrementa il conteggio
26
27             if (count == null) { count = 1; }
28             else { count = count + 1; }
29
30             frequencies.put(word, count);
31         }
32
33         // visualizza tutte le parole e il relativo conteggio
34
35         for (String key : frequencies.keySet())
36         {
37             System.out.printf("%-20s%10d\n", key, frequencies.get(key));
38         }
39     }
40
41 /**
42  * Elimina da una stringa i caratteri che non sono lettere.
43  * @param s una stringa
44  * @return una stringa contenente tutte le lettere presenti in s
45 */
46 public static String clean(String s)
47 {
48     String r = "";
49     for (int i = 0; i < s.length(); i++)
50     {
51         char c = s.charAt(i);
52         if (Character.isLetter(c))
53         {

```

```

54         r = r + c;
55     }
56 }
57 return r.toLowerCase();
58 }
59 }
```

## Argomenti avanzati 14.1

### Funzioni di hash

Se usate un insieme o una mappa che siano realizzati mediante una tabella hash, è possibile che dobbiate realizzare una funzione di hash. Una **funzione di hash** (*hash function*) è una funzione che, a partire da un oggetto, calcola un valore intero, il **codice di hash** (*hash code*), facendo in modo che oggetti diversi abbiano codici di hash diversi con elevata probabilità. Dal momento che la procedura di hashing è così importante, la classe `Object` ha un metodo `hashCode`. L'invocazione

```
int h = x.hashCode();
```

calcola il codice di hash dell'oggetto `x` di tipo qualsiasi. Se volete memorizzare oggetti di una determinata classe come elementi di un insieme di tipo `HashSet` o usarli come chiavi in una mappa di tipo `HashMap`, la classe dovrebbe sovrascrivere questo metodo, con un'implementazione che faccia in modo che oggetti diversi abbiano codici di hash diversi con elevata probabilità.

Ad esempio, la classe `String` definisce una funzione di hash che, per le stringhe, svolge bene il suo compito, generando quasi sempre numeri interi diversi in corrispondenza di stringhe diverse. La Tabella 6 mostra alcune stringhe e i loro codici di hash.

Una funzione di hash calcola un numero intero a partire da un oggetto.

**Tabella 6**  
Codici di hash di alcune stringhe

|  | Stringa  | Codice di hash |
|--|----------|----------------|
|  | "eat"    | 100184         |
|  | "tea"    | 114704         |
|  | "Juliet" | -2065036585    |
|  | "Ugh"    | 84982          |
|  | "VII"    | 84982          |

Una buona funzione di hash minimizza le *collisioni*, che avvengono quando a oggetti diversi vengono associati codici di hash identici.

Può succedere che due o più oggetti diversi abbiano lo stesso codice di hash, dando luogo a una *collisione*. Ad esempio, per le stringhe "VII" e "Ugh" viene calcolato lo stesso codice di hash, ma si tratta di un evento molto raro (si veda l'Esercizio P14.5).

Il metodo `hashCode` della classe `String` combina i caratteri della stringa che elabora, generando un codice numerico che non è semplicemente la somma dei valori dei singoli caratteri, perché questo non mescolerebbe a sufficienza i valori stessi: stringhe che sono permutazioni di un'altra (come "eat" e "tea") avrebbero lo stesso codice di hash.

Questo è il metodo usato nella libreria standard per calcolare il codice di hash di una stringa:

```
final int HASH_MULTIPLIER = 31;
int h = 0;
for (int i = 0; i < s.length(); i++)
{
    h = HASH_MULTIPLIER * h + s.charAt(i);
}
```

Ad esempio, usando ovviamente i valori dei caratteri definiti dal codice Unicode, il codice di hash di "eat" è:

$$31 * (31 * 'e' + 'a') + 't' = 100184$$

Il codice di hash di "tea" è abbastanza diverso:

$$31 * (31 * 't' + 'e') + 'a' = 114704$$

Nelle vostre classi, sovrascrivete il metodo hashCode combinando i codici di hash delle variabili di esemplare.

Nelle vostre classi dovreste definire un codice di hash che combini in modo analogo i codici di hash delle variabili di esemplare. Ad esempio, definiamo il metodo hashCode per la classe Country vista nel Paragrafo 10.1.

La classe ha due variabili di esemplare: il nome della nazione e la sua superficie. Per prima cosa calcoliamo i loro codici di hash: sapete già come calcolare il codice di hash di una stringa; per calcolare il codice di hash di un numero in virgola mobile, costruite dapprima un involucro di tipo Double che lo contenga, poi calcolate il suo codice di hash.

```
public class Country
{
    ...
    public int hashCode()
    {
        int h1 = name.hashCode();
        int h2 = new Double(area).hashCode();
        ...
    }
}
```

Combinate quindi i due codici di hash.

```
final int HASH_MULTIPLIER = 31;
int h = HASH_MULTIPLIER * h1 + h2;
return h;
```

In ogni caso, è più semplice usare il metodo Objects.hash, che calcola i codici di hash di tutti i parametri ricevuti e li combina usando un moltiplicatore:

```
public int hashCode()
{
    return Objects.hash(name, area);
}
```

Quando in una vostra classe definite il metodo `hashCode`, dovete definire anche un metodo `equals` compatibile, perché questo viene utilizzato per distinguere tra loro due oggetti che hanno lo stesso codice di hash.

Il metodo `hashCode` deve essere compatibile con il metodo `equals`.

I metodi `hashCode` e `equals` devono essere reciprocamente *compatibili*: due oggetti uguali devono avere lo stesso codice di hash.

Se una classe sovrscrive il metodo `equals` ma non il metodo `hashCode` ci saranno dei problemi. Supponiamo di aver definito nella classe `Country` il metodo `equals` che dichiari uguali due nazioni dopo averne confrontato il nome e la superficie, senza però sovrscrivere il metodo `hashCode`: di conseguenza, la classe lo eredita dalla superclasse `Object`, che calcola un codice di hash a partire dall'*indirizzo in memoria* dell'oggetto. L'effetto è una probabilità molto alta che due qualsiasi oggetti abbiano codici di hash diversi pur avendo lo stesso contenuto, nel qual caso un `HashSet` li memorizzerà come se fossero due oggetti distinti.

Tuttavia, se non definite né `equals` né `hashCode`, non ci sono problemi. Il metodo `equals` della classe `Object` considera due oggetti uguali solo se hanno lo stesso indirizzo in memoria e, quindi, sono in realtà lo stesso oggetto e hanno, conseguentemente, lo stesso codice di hash. Perciò, la classe `Object` ha metodi `equals` e `hashCode` compatibili. Naturalmente in questo caso il concetto di uguaglianza è molto limitato: due oggetti vengono considerati uguali soltanto se sono lo stesso oggetto, un concetto di uguaglianza peraltro perfettamente valido, dipende dall'applicazione.

## 14.5 Pile, code e code prioritarie

In questo paragrafo prenderemo in considerazione pile, code e code prioritarie, strutture di memorizzazione dei dati che manifestano strategie diverse in relazione all'operazione di rimozione, che elimina, rispettivamente, l'elemento inserito più recentemente, quello inserito meno recentemente e quello avente la massima priorità.

### 14.5.1 Pile

Una pila è una raccolta di elementi con modalità di rimozione "last-in, first-out".

Una **pila** (*stack*) consente l'inserimento e la rimozione di elementi a una sola estremità, che viene tradizionalmente chiamata *cima* (*top*) della pila. Nuovi oggetti vengono inseriti in cima alla pila e gli oggetti possono essere rimossi soltanto dalla cima della pila. Di conseguenza, gli oggetti vengono rimossi in ordine inverso rispetto a come sono stati inseriti, cioè "l'ultimo inserito è il primo a essere estratto" ("last-in, first-out", modalità LIFO). Ad esempio, se inserite gli elementi A, B e C in questo ordine e poi li estraete, otterrete, nell'ordine, C, B e A. Le operazioni di inserimento e di rimozione vengono tradizionalmente chiamate `push` e `pop`.

```
Stack<String> s = new Stack<>();
s.push("A");
s.push("B");
s.push("C");
while (s.size() > 0)
{
    System.out.print(s.pop() + " ");
} // visualizza C B A
```

Ci sono molti possibili utilizzi per le pile nell'informatica: pensate alla possibilità, negli elaboratori di testo, di annullare l'ultima operazione eseguita (“annulla” o “undo”), anche ripetutamente, procedendo a ritroso. Per consentire questo, il programma memorizza in una pila i comandi che vengono eseguiti. Quando richiedete un'operazione di annullamento, viene annullato l'*ultimo* comando eseguito, poi il penultimo, e così via.

Un altro esempio significativo è la **pila di esecuzione** (*run-time stack*) gestita da un processore o da una macchina virtuale per memorizzare le variabili di metodi annidati: ogni volta che viene invocato un metodo, si inseriscono in cima a una pila i suoi parametri e le sue variabili locali; quando il metodo termina, si estraggono dalla pila parametri e variabili.

Nel Paragrafo 14.6 vedrete altre possibili applicazioni.

Nella libreria di Java è presente una semplice classe `Stack` dotata dei metodi `push`, `pop` e `peek`, l'ultimo dei quali restituisce l'elemento che si trova in cima alla pila senza eliminarlo dalla struttura (come riportato nella Tabella 7).

**Tabella 7**  
Lavorare con pile

|                                                            |                                                                                                                                          |
|------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Stack&lt;Integer&gt; s = new Stack&lt;&gt;();</code> | Costruzione di una pila vuota.                                                                                                           |
| <code>s.push(1);</code>                                    | Aggiunta di elementi in cima alla pila; ora s contiene [1, 2, 3]                                                                         |
| <code>s.push(2);</code>                                    | (come il metodo <code>toString</code> della classe <code>Stack</code> , mostriamo la cima della pila in fondo all'elenco del contenuto). |
| <code>s.push(3);</code>                                    |                                                                                                                                          |
| <code>int top = s.pop();</code>                            | Eliminazione dell'elemento che si trova in cima alla pila; top assume il valore 3 e ora s contiene [1, 2].                               |
| <code>head = s.peek();</code>                              | Ispezione dell'elemento presente in cima alla pila, senza rimuoverlo; top vale 2.                                                        |

### 14.5.2 Code

Una coda è una raccolta di elementi con modalità di rimozione “first-in, first-out”.

Una **coda** (*queue*) consente l'inserimento di elementi a un'estremità della coda (fine della coda, *tail*) e la rimozione all'altra estremità (inizio della coda, *head*). Gli oggetti vengono rimossi seguendo la modalità così descritta: “il primo inserito è il primo a essere estratto” (“*first-in, first-out*”, modalità FIFO). Gli elementi vengono, quindi, rimossi nello stesso ordine in cui sono stati inseriti.

Una tipica applicazione è la coda di stampa. Una stampante può essere disponibile per diverse applicazioni, eventualmente eseguite su diversi calcolatori: se ciascuna applicazione tentasse di accedere alla stampante nello stesso momento, la stampa risultante sarebbe una totale confusione; al contrario, ogni applicazione inserisce in un file tutti i byte che deve inviare alla stampante e aggiunge tale file alla coda di stampa. Quando la stampante ha terminato la stampa di un file estrae il successivo dalla coda, per cui i lavori vengono stampati secondo la regola “il primo entrato è il primo a uscire”, che è una strategia accettabile per gli utenti della stampante condivisa.

Come riportato nella Tabella 8, l'interfaccia `Queue` della libreria standard di Java ha i metodi `add` per inserire un elemento alla fine della coda, `remove` per eliminare l'elemento che si trova all'inizio della coda e `peek` per ispezionare tale elemento senza eliminarlo.

La classe `LinkedList` implementa l'interfaccia `Queue`, per cui, quando vi serve una coda, potete semplicemente assegnare un oggetto di tipo `LinkedList` a una variabile di tipo `Queue`:

```
Queue<String> q = new LinkedList<>();
q.add("A");
q.add("B");
q.add("C");
while (q.size() > 0)
{
    System.out.print(q.remove() + " ");
} // visualizza A B C
```

**Tabella 8**  
Lavorare con code

|                                                                                                                                     |                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Queue<Integer> q = new LinkedList<>();<br>q.add(1);<br>q.add(2);<br>q.add(3);<br><br>int head = q.remove();<br><br>head = q.peek(); | La classe <code>LinkedList</code> implementa l'interfaccia <code>Queue</code> .<br>Aggiunta di elementi alla fine della coda; ora <code>q</code> contiene [1, 2, 3].<br><br>Eliminazione dell'elemento che si trova all'inizio della coda; <code>head</code> assume il valore 1 e ora <code>q</code> contiene [2, 3].<br>Ispezione dell'elemento presente all'inizio della coda, senza rimuoverlo; <code>head</code> vale 2. |
|-------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### 14.5.3

#### Code prioritarie

Quando si estraе un elemento da una coda prioritaria viene eliminato quello aente la prioritа massima.

Una **coda prioritaria** (*priority queue*) è una raccolta di elementi, ciascuno dei quali ha una *prioritа*. Un esempio tipico di coda prioritaria è una raccolta di richieste di esecuzione di lavori, alcune delle quali possono essere più urgenti di altre. Diversamente da una coda ordinaria, la coda prioritaria non gestisce le rimozioni di elementi con una strategia di tipo FIFO: gli elementi vengono rimossi sulla base della loro prioritа. In altre parole, si possono inserire nuovi elementi in qualunque ordine, ma quando si richiede la rimozione di un elemento viene estratto quello aente la prioritа massima.

C'è la consuetudine di attribuire valori più bassi alle prioritа più elevate, con la prioritа 1 che indica la massima urgenza. Di conseguenza, ogni operazione di rimozione estraе dalla coda prioritaria l'elemento *minimo*.

Consideriamo, ad esempio, il codice seguente, nel quale aggiungiamo a una coda prioritaria oggetti di tipo `WorkOrder`, che rappresentano richieste di esecuzione di lavori, ognuna delle quali ha una prioritа e una descrizione:

```
PriorityQueue<WorkOrder> q = new PriorityQueue<>();
q.add(new WorkOrder(3, "Shampoo carpets"));
q.add(new WorkOrder(1, "Fix broken sink"));
q.add(new WorkOrder(2, "Order cleaning supplies"));
```

Quando si invoca `q.remove()` per la prima volta viene eliminata la richiesta che ha prioritа 1. La successiva invocazione di `q.remove()` elimina la richiesta che ha la prioritа più elevata tra quelle rimaste: nel nostro esempio, la prioritа di valore minimo (cioè la più elevata) è 2. Se sono presenti due elementi che hanno la stessa prioritа, la coda prioritaria puо risolvere la situazione di paritа in modo arbitrario.

Dato che la coda prioritaria deve essere in grado di stabilire quale elemento ha la prioritа minima, gli elementi inseriti nella raccolta devono essere esemplari di una classe che implementa l'interfaccia `Comparable` (descritta nel Paragrafo 10.3).

La Tabella 9 mostra i metodi principali della classe `PriorityQueue` presente nella libreria standard.

**Tabella 9**  
Lavorare con code prioritarie

|                                                                                                                                                                                                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>PriorityQueue&lt;Integer&gt; q =<br/>new PriorityQueue&lt;&gt;();<br/>q.add(3); q.add(1); q.add(2);<br/>int first = q.remove();<br/>int second = q.remove();<br/>int next = q.peek();</code> | Questa coda prioritaria contiene oggetti di tipo <code>Integer</code> . Di solito, invece, si usano oggetti che descrivono compiti da svolgere.<br>Aggiunta di elementi alla coda prioritaria.<br>Ogni invocazione di <code>remove</code> elimina l'elemento più urgente: a <code>first</code> viene assegnato il valore 1, a <code>second</code> il valore 2.<br>Ispezione dell'elemento con priorità di valore minimo tra quelli della coda prioritaria, senza eliminarlo. |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|



## Auto-valutazione

21. Perché, invece di dichiararla semplicemente come lista concatenata, si dichiara una variabile in questo modo?

```
Queue<String> q = new LinkedList<>();
```

22. Per quale motivo un vettore non è adatto all'implementazione di una coda?

23. Cosa visualizza questo frammento di codice?

```
Queue<String> q = new LinkedList<>();  
q.add("A");  
q.add("B");  
q.add("C");  
while (q.size() > 0) { System.out.print(q.remove() + " "); }
```

24. Per quale motivo una pila non è adatta alla gestione dei file inviati a una stampante?

25. Nel codice visto come esempio di utilizzo di una coda prioritaria abbiamo usato la classe `WorkOrder`: avremmo potuto invece usare stringhe, in questo modo?

```
PriorityQueue<String> q = new PriorityQueue<>();  
q.add("3 - Shampoo carpets");  
q.add("1 - Fix broken sink ");  
q.add("2 - Order cleaning supplies ");
```

## Per far pratica

A questo punto si consiglia di svolgere gli esercizi R14.15, E14.8 e E14.9, al termine del capitolo.



## Argomenti avanzati 14.2

### Notazione polacca inversa

Negli Anni Venti il matematico polacco Jan Łukasiewicz comprese che era possibile evitare l'utilizzo di parentesi nelle espressioni aritmetiche scrivendo gli operatori *prima* dei propri operandi, ad esempio `+ 3 4` invece di `3 + 4`. Trent'anni più tardi l'informatico australiano Charles Hamblin notò che si poteva ottenere una sintassi ancora migliore

scrivendo gli operatori *dopo* i relativi operandi, introducendo quella che venne chiamata *notazione polacca inversa* (RPN, *reverse Polish notation*).

| Notazione standard       | Notazione polacca inversa (RPN) |
|--------------------------|---------------------------------|
| $3 + 4$                  | $3 \ 4 \ +$                     |
| $3 + 4 \times 5$         | $3 \ 4 \ 5 \times \ +$          |
| $3 \times (4 + 5)$       | $3 \ 4 \ 5 \ + \times$          |
| $(3 + 4) \times (5 + 6)$ | $3 \ 4 \ + \ 5 \ 6 \ + \times$  |
| $3 + 4 + 5$              | $3 \ 4 \ + \ 5 \ +$             |

La notazione RPN può sembrare strana ma si tratta solo di un caso, dovuto agli avvenimenti storici: se i matematici ne avessero compreso i vantaggi fin dall'inizio, gli studenti di oggi la userebbero con naturalezza e non si preoccuperebbero più di parentesi e regole di precedenza.

Nel 1972 Hewlett-Packard presentò la calcolatrice HP 35, che usava la *notazione polacca inversa* o RPN (Reverse Polish Notation): non aveva tasti per le parentesi o per il segno di uguale, ma si usava il tasto ENTER (“Invio”) per inserire un numero sulla pila, perciò la divisione marketing di Hewlett-Packard aveva creato per tale prodotto lo slogan “la calcolatrice che non ha uguale”, giocando sul doppio senso della frase.



Con il passare del tempo gli sviluppatori di calcolatrici hanno deciso di adottare la normale notazione algebrica, piuttosto che costringere gli utilizzatori a imparare una nuova notazione, ma quegli utenti che avevano fatto lo sforzo di imparare RPN tendono a esserne fanatici sostenitori e anche oggi alcuni modelli di Hewlett-Packard gestiscono tale notazione.

## 14.6 Applicazioni di pile e code

Nonostante la loro semplicità, le pile e le code sono strutture estremamente versatili. Nei paragrafi che seguono vedremo alcune delle loro applicazioni più utili.

### 14.6.1 Accoppiamento delle parentesi

Per verificare se in un'espressione le parentesi sono accoppiate correttamente si può usare una pila.

Nella sezione Errori comuni 4.2 avete visto un semplice trucco che consente di individuare in una espressione la presenza di parentesi non accoppiate correttamente, ad esempio così:

$$\begin{array}{ccccccc} -(b * b - (4 * a * c)) / (2 * a) \\ 1 & & 2 & & 1 & 0 & 1 & 0 \end{array}$$

Si incrementa un contatore ogni volta che si incontra una parentesi aperta e lo si decrementa quando si incontra una parentesi chiusa. Il contatore non deve mai diventare negativo e al termine dell'espressione deve essere uguale a zero.

Questo trucco funziona per la verifica delle espressioni in Java, perché si usa un unico tipo di parentesi, ma nella notazione matematica si possono avere più tipi di parentesi, come in questo esempio:

$$-\{ [b \cdot b - (4 \cdot a \cdot c)] / (2 \cdot a) \}$$

Per vedere se una tale espressione è composta in modo corretto, si inseriscono le sue parentesi in una pila:

- Quando si trova una parentesi aperta, la si inserisce nella pila.
- Quando si trova una parentesi chiusa, si rimuove un elemento dalla pila.
- Se le parentesi aperta e chiusa non corrispondono
  - Le parentesi non sono accoppiate correttamente. Fine.
- Se alla fine la pila è vuota
  - Le parentesi sono accoppiate correttamente.
- Altrimenti
  - Le parentesi non sono accoppiate correttamente.

Ecco un'esecuzione dell'algoritmo passo dopo passo, nell'analisi della semplice espressione precedente:

| Pila             | Espressione da esaminare           | Commenti               |
|------------------|------------------------------------|------------------------|
| Vuota            | $-[b * b - (4 * a * c)] / (2 * a)$ |                        |
| {                | $[b * b - (4 * a * c)] / (2 * a)$  |                        |
| { [              | $b * b - (4 * a * c)] / (2 * a)$   |                        |
| { [ (            | $4 * a * c)] / (2 * a)$            |                        |
| { [ ] / (2 * a)} |                                    | Tonde accoppiate       |
| { / (2 * a) }    |                                    | Quadre accoppiate      |
| { ( 2 * a) }     |                                    |                        |
| { }              |                                    | Graffe accoppiate      |
| Vuota            | Dati terminati                     | Graffe accoppiate      |
|                  |                                    | Accoppiamento corretto |

### 14.6.2 Valutazione di espressioni RPN

Per valutare espressioni in notazione polacca inversa si può usare una pila.

Pensate a come si scrivono le espressioni aritmetiche, come  $(3 + 4) \times 5$ : le parentesi sono necessarie perché 3 e 4 vengano sommati prima di moltiplicare per 5 il risultato dell'addizione.

Se, però, si scrivono gli operatori *dopo* i loro operandi si possono eliminare le parentesi e l'espressione precedente diventa  $3\ 4 + 5 \times$  (come visto nella sezione Argomenti avanzati 14.2). Per valutare questa espressione bisogna applicare l'operatore di addizione,  $+$ , a 3 e 4, ottenendo 7, per poi valutare l'espressione  $7\ 5 \times$ , che genera il risultato 35. Per espressioni più complesse, il problema si complica. Ad esempio,  $3\ 4\ 5 + \times$  richiede il calcolo di  $4\ 5 +$  (il cui risultato è 9), per poi valutare l'espressione  $3\ 9 \times$ . Se valutiamo questa espressione da sinistra a destra, come si fa normalmente, dobbiamo lasciare in sospeso l'operando 3 da qualche parte mentre valutiamo la sotto-espressione  $4\ 5 +$ . Dove lo parcheggiamo? Lo mettiamo in una pila. In effetti, l'algoritmo che valuta espressioni in notazione polacca inversa è piuttosto semplice:

- Se è stato letto un numero
  - Inseriscilo nella pila.
- Altrimenti se è stato letto un operatore
  - Estrai due valori dalla pila.
  - Applica l'operatore ai due valori estratti.
  - Inserisci nella pila il risultato ottenuto.
- Altrimenti se non ci sono più dati da leggere
  - Estrai dalla pila il risultato e visualizzalo.

La figura mostra, passo dopo passo, la valutazione dell'espressione  $3\ 4\ 5 + \times$ .

| Pila  | Espressione da esaminare | Commenti                                               |
|-------|--------------------------|--------------------------------------------------------|
| Vuota | $3\ 4\ 5 + \times$       |                                                        |
| 3     | $4\ 5 + \times$          | Numeri inseriti nella pila                             |
| 3 4   | $5 + \times$             |                                                        |
| 3 4 5 | $+ \times$               |                                                        |
| 3 9   | $\times$                 | Estrai 4 e 5 e impila 9, il risultato di $4 + 5$       |
| 27    | Dati terminati           | Estrai 3 e 9 e impila 27, il risultato di $3 \times 9$ |
| Vuota |                          | Estrai e visualizza il risultato, 27                   |

Il programma seguente simula il funzionamento di una calcolatrice in notazione polacca inversa.

### File Calculator.java

```

1 import java.util.Scanner;
2 import java.util.Stack;
3
4 /**
5  * Questa calcolatrice usa la notazione polacca inversa.
6 */
7 public class Calculator
8 {
9     public static void main(String[] args)
10    {
11        Scanner in = new Scanner(System.in);
12        Stack<Integer> results = new Stack<>();
13        System.out.println("Enter one number or operator per line, Q to quit. ");

```

```

14     boolean done = false;
15     while (!done)
16     {
17         String input = in.nextLine();
18
19         // se è un operatore, estrai gli operandi e impila il risultato
20
21         if (input.equals("+"))
22         {
23             results.push(results.pop() + results.pop());
24         }
25         else if (input.equals("-"))
26         {
27             Integer arg2 = results.pop();
28             results.push(results.pop() - arg2);
29         }
30         else if (input.equals("*") || input.equals("x"))
31         {
32             results.push(results.pop() * results.pop());
33         }
34         else if (input.equals("/"))
35         {
36             Integer arg2 = results.pop();
37             results.push(results.pop() / arg2);
38         }
39         else if (input.equals("Q") || input.equals("q"))
40         {
41             done = true;
42         }
43         else
44         {
45             // non è un operatore, impila il valore
46
47             results.push(Integer.parseInt(input));
48         }
49     System.out.println(results);
50 }
51 }
52 }
```

### 14.6.3 Valutazione di espressioni algebriche

Usando due pile si possono valutare espressioni nella notazione algebrica convenzionale.

Nel paragrafo precedente avete visto come sia possibile valutare espressioni in notazione polacca inversa usando una pila. Se quella notazione non vi è piaciuta, sarete lieti di sapere che si possono valutare anche espressioni nella notazione algebrica convenzionale usando due pile, una per gli operandi e una per gli operatori.

Consideriamo prima un esempio semplice, come l'espressione  $3 + 4$ . Inseriamo gli operandi, 3 e 4, nella pila degli operandi e l'operatore,  $+$ , nella pila degli operatori. Poi, estraiamo due operandi dalla pila degli operandi e un operatore dalla pila degli operatori, eseguiamo l'operazione e inseriamo il risultato nella pila degli operandi.

La procedura appena descritta è il fulcro dell'algoritmo di valutazione di un'espressione generica: la chiamiamo "valutazione della cima" (lasciando sottinteso "della pila").

Nella notazione algebrica ogni operatore ha una ben definita *precedenza*: gli operatori  $+$  e  $-$  hanno la precedenza più bassa (e uguale tra loro), mentre  $\times$  e  $/$  hanno la precedenza più alta (e uguale tra loro).

| Pila operandi<br>Vuota                                             | Pila operatori<br>Vuota | Espressione<br>da elaborare<br>$3 + 4$ | Commenti |                  |                           |
|--------------------------------------------------------------------|-------------------------|----------------------------------------|----------|------------------|---------------------------|
| 1 <table border="1"><tr><td>3</td></tr></table>                    | 3                       |                                        | + 4      |                  |                           |
| 3                                                                  |                         |                                        |          |                  |                           |
| 2 <table border="1"><tr><td>3</td></tr></table>                    | 3                       | +                                      | 4        |                  |                           |
| 3                                                                  |                         |                                        |          |                  |                           |
| 3 <table border="1"><tr><td>4</td></tr><tr><td>3</td></tr></table> | 4                       | 3                                      | +        | Dati terminati   | Valuta la cima della pila |
| 4                                                                  |                         |                                        |          |                  |                           |
| 3                                                                  |                         |                                        |          |                  |                           |
| 4 <table border="1"><tr><td>7</td></tr></table>                    | 7                       |                                        |          | Il risultato è 7 |                           |
| 7                                                                  |                         |                                        |          |                  |                           |

Consideriamo l'espressione  $3 \times 4 + 5$ . Ecco i primi passi della sua valutazione:

| Pila operandi<br>Vuota                                             | Pila operatori<br>Vuota | Espressione<br>da elaborare<br>$3 \times 4 + 5$ | Commenti       |     |                            |
|--------------------------------------------------------------------|-------------------------|-------------------------------------------------|----------------|-----|----------------------------|
| 1 <table border="1"><tr><td>3</td></tr></table>                    | 3                       |                                                 | $\times 4 + 5$ |     |                            |
| 3                                                                  |                         |                                                 |                |     |                            |
| 2 <table border="1"><tr><td>3</td></tr></table>                    | 3                       | $\times$                                        | $4 + 5$        |     |                            |
| 3                                                                  |                         |                                                 |                |     |                            |
| 3 <table border="1"><tr><td>4</td></tr><tr><td>3</td></tr></table> | 4                       | 3                                               | $\times$       | + 5 | Valuta $\times$ prima di + |
| 4                                                                  |                         |                                                 |                |     |                            |
| 3                                                                  |                         |                                                 |                |     |                            |

Dato che l'operatore  $\times$  ha la precedenza sull'operatore  $+$ , siamo pronti a valutare la cima:

| Pila operandi                                                       | Pila operatori | Commenti |                       |                |
|---------------------------------------------------------------------|----------------|----------|-----------------------|----------------|
| 4 <table border="1"><tr><td>12</td></tr></table>                    | 12             | +        | 5                     |                |
| 12                                                                  |                |          |                       |                |
| 5 <table border="1"><tr><td>5</td></tr><tr><td>12</td></tr></table> | 5              | 12       | +                     | Dati terminati |
| 5                                                                   |                |          |                       |                |
| 12                                                                  |                |          |                       |                |
| 6 <table border="1"><tr><td>17</td></tr></table>                    | 17             |          | Questo è il risultato |                |
| 17                                                                  |                |          |                       |                |

Nel valutare, invece, l'espressione  $3 + 4 \times 5$ , inseriamo l'operatore  $\times$  nella pila degli operatori, perché prima dobbiamo leggere il suo secondo operando, dopodiché potremo eseguire la moltiplicazione e, infine, l'addizione.

| Pila operandi<br>Vuota                          | Pila operatori<br>Vuota | Espressione<br>da elaborare<br>$3 + 4 \times 5$ | Commenti       |  |
|-------------------------------------------------|-------------------------|-------------------------------------------------|----------------|--|
| 1 <table border="1"><tr><td>3</td></tr></table> | 3                       |                                                 | $+ 4 \times 5$ |  |
| 3                                               |                         |                                                 |                |  |
| 2 <table border="1"><tr><td>3</td></tr></table> | 3                       | +                                               | $4 + 5$        |  |
| 3                                               |                         |                                                 |                |  |

|          |                                                                  |   |   |                                                                                    |                                                                                    |          |   |                              |
|----------|------------------------------------------------------------------|---|---|------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|----------|---|------------------------------|
| 3        | <table border="1"><tr><td>4</td></tr><tr><td>3</td></tr></table> | 4 | 3 | .                                                                                  | <table border="1"><tr><td><math>\times</math></td></tr><tr><td>5</td></tr></table> | $\times$ | 5 | Non valutare $\times$ adesso |
| 4        |                                                                  |   |   |                                                                                    |                                                                                    |          |   |                              |
| 3        |                                                                  |   |   |                                                                                    |                                                                                    |          |   |                              |
| $\times$ |                                                                  |   |   |                                                                                    |                                                                                    |          |   |                              |
| 5        |                                                                  |   |   |                                                                                    |                                                                                    |          |   |                              |
| 4        | <table border="1"><tr><td>4</td></tr><tr><td>3</td></tr></table> | 4 | 3 | <table border="1"><tr><td><math>\times</math></td></tr><tr><td>+</td></tr></table> | $\times$                                                                           | +        | 5 |                              |
| 4        |                                                                  |   |   |                                                                                    |                                                                                    |          |   |                              |
| 3        |                                                                  |   |   |                                                                                    |                                                                                    |          |   |                              |
| $\times$ |                                                                  |   |   |                                                                                    |                                                                                    |          |   |                              |
| +        |                                                                  |   |   |                                                                                    |                                                                                    |          |   |                              |

In altre parole, lasciamo gli operatori nella relativa pila fino a quando non siamo pronti a valutarli. Ecco la parte rimanente della valutazione:

|          | Pila operandi                                                                       | Pila operatori | Commenti |                                               |                                                                                    |                                       |   |                                                |
|----------|-------------------------------------------------------------------------------------|----------------|----------|-----------------------------------------------|------------------------------------------------------------------------------------|---------------------------------------|---|------------------------------------------------|
| 5        | <table border="1"><tr><td>5</td></tr><tr><td>4</td></tr><tr><td>3</td></tr></table> | 5              | 4        | 3                                             | <table border="1"><tr><td><math>\times</math></td></tr><tr><td>+</td></tr></table> | $\times$                              | + | Dati terminati<br>Valuta la cima<br>della pila |
| 5        |                                                                                     |                |          |                                               |                                                                                    |                                       |   |                                                |
| 4        |                                                                                     |                |          |                                               |                                                                                    |                                       |   |                                                |
| 3        |                                                                                     |                |          |                                               |                                                                                    |                                       |   |                                                |
| $\times$ |                                                                                     |                |          |                                               |                                                                                    |                                       |   |                                                |
| +        |                                                                                     |                |          |                                               |                                                                                    |                                       |   |                                                |
| 6        | <table border="1"><tr><td>20</td></tr><tr><td>3</td></tr></table>                   | 20             | 3        | <table border="1"><tr><td>+</td></tr></table> | +                                                                                  | Valuta di nuovo<br>la cima della pila |   |                                                |
| 20       |                                                                                     |                |          |                                               |                                                                                    |                                       |   |                                                |
| 3        |                                                                                     |                |          |                                               |                                                                                    |                                       |   |                                                |
| +        |                                                                                     |                |          |                                               |                                                                                    |                                       |   |                                                |
| 7        | <table border="1"><tr><td>23</td></tr></table>                                      | 23             |          | Questo è il risultato                         |                                                                                    |                                       |   |                                                |
| 23       |                                                                                     |                |          |                                               |                                                                                    |                                       |   |                                                |

Per vedere come si gestiscono le parentesi, consideriamo l'espressione  $3 \times (4 + 5)$ . Per prima cosa la parentesi aperta viene inserita in cima alla pila degli operatori, dove viene poi inserito anche l'operatore  $+$ . Quando incontriamo la parentesi chiusa, sappiamo di essere pronti a valutare la cima della pila, proseguendo fino a quando non compare la parentesi aperta corrispondente:

|          | Pila operandi                                                                       | Pila operatori | Espressione da elaborare                                                           | Commenti                                                                                              |                                                                                                       |          |                              |          |   |                              |
|----------|-------------------------------------------------------------------------------------|----------------|------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------|----------|------------------------------|----------|---|------------------------------|
|          | Vuota                                                                               | Vuota          | $3 \times (4 + 5)$                                                                 |                                                                                                       |                                                                                                       |          |                              |          |   |                              |
| 1        | <table border="1"><tr><td>3</td></tr></table>                                       | 3              |                                                                                    | $\times (4 + 5)$                                                                                      |                                                                                                       |          |                              |          |   |                              |
| 3        |                                                                                     |                |                                                                                    |                                                                                                       |                                                                                                       |          |                              |          |   |                              |
| 2        | <table border="1"><tr><td>3</td></tr></table>                                       | 3              | <table border="1"><tr><td><math>\times</math></td></tr></table>                    | $\times$                                                                                              | $(4 + 5)$                                                                                             |          |                              |          |   |                              |
| 3        |                                                                                     |                |                                                                                    |                                                                                                       |                                                                                                       |          |                              |          |   |                              |
| $\times$ |                                                                                     |                |                                                                                    |                                                                                                       |                                                                                                       |          |                              |          |   |                              |
| 3        | <table border="1"><tr><td>3</td></tr></table>                                       | 3              | <table border="1"><tr><td>(</td></tr><tr><td><math>\times</math></td></tr></table> | (                                                                                                     | $\times$                                                                                              | $4 + 5)$ | Non valutare $\times$ adesso |          |   |                              |
| 3        |                                                                                     |                |                                                                                    |                                                                                                       |                                                                                                       |          |                              |          |   |                              |
| (        |                                                                                     |                |                                                                                    |                                                                                                       |                                                                                                       |          |                              |          |   |                              |
| $\times$ |                                                                                     |                |                                                                                    |                                                                                                       |                                                                                                       |          |                              |          |   |                              |
| 4        | <table border="1"><tr><td>4</td></tr><tr><td>3</td></tr></table>                    | 4              | 3                                                                                  | <table border="1"><tr><td>(</td></tr><tr><td><math>\times</math></td></tr></table>                    | (                                                                                                     | $\times$ | $+ 5)$                       |          |   |                              |
| 4        |                                                                                     |                |                                                                                    |                                                                                                       |                                                                                                       |          |                              |          |   |                              |
| 3        |                                                                                     |                |                                                                                    |                                                                                                       |                                                                                                       |          |                              |          |   |                              |
| (        |                                                                                     |                |                                                                                    |                                                                                                       |                                                                                                       |          |                              |          |   |                              |
| $\times$ |                                                                                     |                |                                                                                    |                                                                                                       |                                                                                                       |          |                              |          |   |                              |
| 5        | <table border="1"><tr><td>4</td></tr><tr><td>3</td></tr></table>                    | 4              | 3                                                                                  | <table border="1"><tr><td>+</td></tr><tr><td>(</td></tr><tr><td><math>\times</math></td></tr></table> | +                                                                                                     | (        | $\times$                     | $5)$     |   |                              |
| 4        |                                                                                     |                |                                                                                    |                                                                                                       |                                                                                                       |          |                              |          |   |                              |
| 3        |                                                                                     |                |                                                                                    |                                                                                                       |                                                                                                       |          |                              |          |   |                              |
| +        |                                                                                     |                |                                                                                    |                                                                                                       |                                                                                                       |          |                              |          |   |                              |
| (        |                                                                                     |                |                                                                                    |                                                                                                       |                                                                                                       |          |                              |          |   |                              |
| $\times$ |                                                                                     |                |                                                                                    |                                                                                                       |                                                                                                       |          |                              |          |   |                              |
| 6        | <table border="1"><tr><td>5</td></tr><tr><td>4</td></tr><tr><td>3</td></tr></table> | 5              | 4                                                                                  | 3                                                                                                     | <table border="1"><tr><td>+</td></tr><tr><td>(</td></tr><tr><td><math>\times</math></td></tr></table> | +        | (                            | $\times$ | ) | Valuta la cima<br>della pila |
| 5        |                                                                                     |                |                                                                                    |                                                                                                       |                                                                                                       |          |                              |          |   |                              |
| 4        |                                                                                     |                |                                                                                    |                                                                                                       |                                                                                                       |          |                              |          |   |                              |
| 3        |                                                                                     |                |                                                                                    |                                                                                                       |                                                                                                       |          |                              |          |   |                              |
| +        |                                                                                     |                |                                                                                    |                                                                                                       |                                                                                                       |          |                              |          |   |                              |
| (        |                                                                                     |                |                                                                                    |                                                                                                       |                                                                                                       |          |                              |          |   |                              |
| $\times$ |                                                                                     |                |                                                                                    |                                                                                                       |                                                                                                       |          |                              |          |   |                              |

|   |        |        |                |                                       |
|---|--------|--------|----------------|---------------------------------------|
| 7 | 9<br>3 | (<br>x | Dati terminati | Estrai (                              |
| 8 | 9<br>3 | x      |                | Valuta di nuovo<br>la cima della pila |
| 9 | 27     |        |                | Questo è il risultato                 |

Ecco l'algoritmo:

- Se è stato letto un numero
  - Inseriscilo nella pila degli operandi.
- Altrimenti se è stata letta una parentesi aperta
  - Inseriscila nella pila degli operandi.
- Altrimenti se è stato letto l'operatore *op*
  - Finché l'operatore in cima alla pila ha la precedenza su *op*
    - Valuta la cima.
  - Inserisci *op* nella pila degli operatori.
- Altrimenti se è stata letta una parentesi chiusa
  - Finché in cima alla pila non c'è una parentesi aperta
    - Valuta la cima.
  - Estrai la parentesi aperta dalla pila degli operatori.
- Altrimenti se non ci sono più dati da leggere
  - Finché la pila degli operatori non è vuota
    - Valuta la cima.

Al termine, il valore rimasto in cima alla pila degli operandi è il valore dell'espressione.

L'algoritmo utilizza il seguente metodo ausiliario, che valuta l'operatore che si trova in cima alla pila degli operatori usando come operandi i due che si trovano in cima alla pila degli operandi:

- Valuta la cima:
  - Estrai due operandi dalla pila degli operandi.
  - Estrai un operatore dalla pila degli operatori.
  - Esegui l'operazione tra l'operatore e gli operandi.
  - Inserisci il risultato nella pila degli operandi.

#### 14.6.4

#### Backtracking

Per poter tornare sui propri passi e prendere una strada diversa, un algoritmo di backtracking usa una pila per ricordare le alternative ancora inesplorate.

Immaginate di essere all'interno di un labirinto e di dover trovare l'uscita. Cosa dovete fare quando vi trovate in un incrocio? Potete proseguire esplorando uno dei percorsi che si diramano, ma volette ricordare i percorsi alternativi: se il percorso scelto non è quello giusto, potrete tornare indietro e tentare una delle alternative che non avete ancora provato.

Ovviamente, seguendo il percorso potrete trovare altri incroci, le cui alternative vanno analogamente memorizzate. Una soluzione semplice consiste nell'utilizzare una pila, in cui inserire i percorsi che devono ancora essere esplorati. La procedura che prevede di tornare indietro a un punto in cui si è effettuata una scelta per provare una diversa

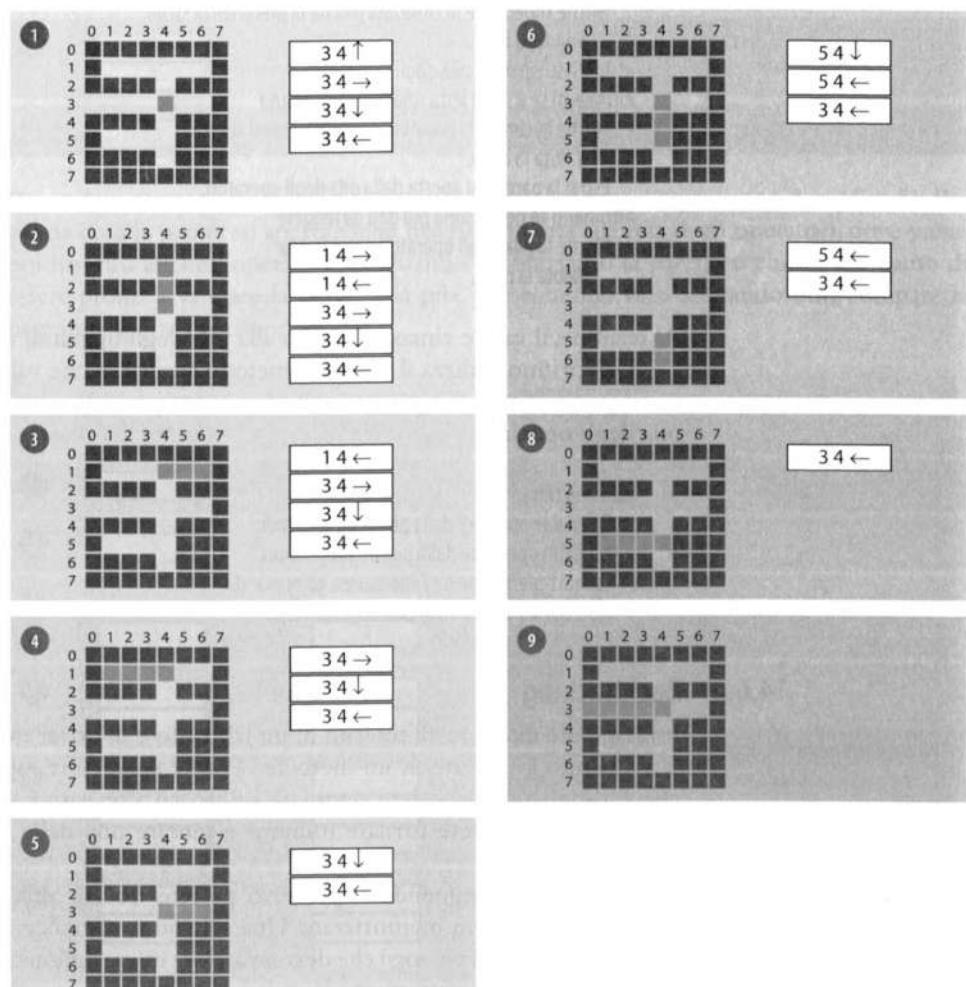
alternativa si chiama *backtracking* (“procedere a ritroso”). Usando una pila tornerete alla scelta compiuta più di recente prima di ritornare a quelle più lontane nel passato.

La Figura 11 mostra un esempio. Partiamo da un punto all'interno del labirinto, la posizione (3, 4). Ci sono quattro percorsi possibili e li inseriamo tutti in una pila (riquadro 1) indicando con una freccia la direzione da prendere a partire dal punto memorizzato. Estraiamo l'indicazione che si trova in cima alla pila e ci spostiamo verso nord a partire dalla posizione (3, 4), arrivando nella posizione (1, 4). A questo punto inseriamo due alternative nella pila, indecisi se proseguire verso ovest o verso est (riquadro 2). Entrambe le strade, però, portano a un vicolo cieco (riquadri 3 e 4).

Ora estraiamo dalla pila l'indicazione di procedere verso est a partire dalla posizione (3, 4): questo ci porta di nuovo in un vicolo cieco (riquadro 5). Il prossimo tentativo è il percorso che da (3, 4) si dirige verso sud. Nel punto (5, 4) troviamo un incrocio e inseriamo nella pila le due alternative possibili (riquadro 6): portano entrambe in un vicolo cieco (riquadri 7 e 8). Entrambe le strade, però, portano a un vicolo cieco (riquadro 9).

**Figura 11**

Uscire da un labirinto  
usando il backtracking



Infine, il percorso che da (3, 4) va verso ovest ci porta all'uscita (riquadro 9).

Abbiamo trovato un percorso per uscire dal labirinto usando una pila. Ecco, infatti, lo pseudocodice che descrive l'algoritmo di uscita da un labirinto:

```

Impila tutti i percorsi che partono dal punto in cui ti trovi.
Finché la pila non è vuota
    Estrai un percorso dalla pila.
    Segui il percorso fino a un'uscita, un incrocio o un vicolo cieco.
    Se hai trovato un'uscita
        Congratulazioni!
    Altrimenti se hai trovato un incrocio
        Impila tutti i percorsi che partono dal punto in cui ti trovi,
        tranne il percorso da cui provieni.
```

Se il labirinto non ha *percorsi ciclici*, questo algoritmo troverà un'uscita. Se, invece, è possibile che si segua un percorso ciclico e si torni a un incrocio già visitato seguendo una sequenza di percorsi diversi, allora per trovare un'uscita bisogna lavorare più duramente, come delineato nell'Esercizio E14.21.

L'implementazione di questo algoritmo è strettamente correlata al modo in cui si descrive il labirinto. Ad esempio, questo si può fare con un array bidimensionale di caratteri, con spazi che rappresentano i corridoi di passaggio e asterischi che indicano le barriere invalicabili:

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

Per rappresentare una posizione di partenza e una direzione (nord, est, sud o ovest) si può progettare una classe `Path`, mentre la classe `Maze` avrà un metodo che estende un percorso lungo la direzione attuale fino a raggiungere un incrocio o un'uscita, oppure finché non viene bloccato da una barriera, oltre a un metodo che individua tutti i percorsi che si diramano da un incrocio.

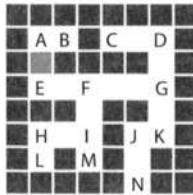
Si noti che in questo algoritmo si può usare una coda al posto della pila: di conseguenza, verranno esplorate le alternative meno recenti prima di quelle più recenti. Per trovare una soluzione del problema questo approccio è altrettanto efficace, ma, pensando all'esplorazione concreta di un labirinto, è decisamente meno intuitivo: bisogna immaginare di poter essere teletrasportati indietro all'incrocio iniziale invece di camminare a ritroso fino all'ultimo incrocio visitato.



## Auto-valutazione

26. Qual è il valore dell'espressione  $2 \ 3 \ 4 + 5 \times \times$  in notazione polacca inversa?
27. Perché nel programma `Calculator` la gestione dell'operatore di sottrazione non esegue semplicemente l'enunciato seguente?

- results.push(results.pop() - results.pop());
28. Valutando l'espressione  $3 - 4 + 5$  con l'algoritmo visto nel Paragrafo 14.6.3, quale operatore viene valutato per primo?
29. Nell'algoritmo visto nel Paragrafo 14.6.3 gli operatori, all'interno della loro pila, sono sempre disposti in ordine di precedenza crescente?
30. Considerate il seguente esempio di labirinto. Immaginando di partire dal punto indicato e inserendo nella pila i percorsi alternativi nell'ordine ovest, sud, est e nord, in quale ordine vengono visitati i punti contrassegnati dalle lettere eseguendo l'algoritmo presentato nel Paragrafo 14.6.4?



### Per far pratica

A questo punto si consiglia di svolgere gli esercizi R14.25, E14.18, E14.20, E14.21 e E14.22, al termine del capitolo.

## Riepilogo degli obiettivi di apprendimento

### Comprendere l'architettura del Java Collections Framework

- Una raccolta (*collection*) raggruppa insieme elementi e ne consente il recupero successivo.
- Una lista (*list*) è una raccolta che ricorda l'ordine relativo tra i propri elementi.
- Un insieme (*set*) è una raccolta non ordinata di elementi non duplicati.
- Una mappa (*map*) gestisce associazioni tra chiavi e valori.

### Utilizzare liste concatenate

- Una lista concatenata è composta da un certo numero di nodi, ciascuno dei quali contiene un riferimento al nodo successivo.
- Aggiungere e rimuovere elementi in una data posizione di una lista concatenata è un'operazione efficiente.
- Visitare in sequenza gli elementi di una lista concatenata è efficiente, ma accedervi secondo la modalità di accesso casuale non lo è.
- Per accedere agli elementi presenti in una lista concatenata si usa un iteratore (o cursore).

### Scegliere un'implementazione di insieme e usarla per gestire insiemi di valori

- Le classi `HashSet` e `TreeSet` implementano l'interfaccia `Set`.
- Le implementazioni di insieme dispongono gli elementi in modo da poterli ritrovare velocemente.
- Si possono usare insiemi realizzati mediante tabella hash quando devono, ad esempio, contenere oggetti di tipo `String`, `Integer`, `Double`, `Point`, `Rectangle` o `Color`.
- Si possono usare insiemi realizzati mediante albero di ricerca binario con elementi di qualche classe che implementi l'interfaccia `Comparable`, come `String` o `Integer`.
- Gli insiemi non contengono elementi duplicati. L'aggiunta all'insieme di un elemento duplicato di un altro elemento già presente nell'insieme viene semplicemente ignorata.
- Un iteratore visita gli elementi di un insieme nell'ordine in cui questi sono memorizzati

all'interno dell'insieme stesso, in relazione al tipo di implementazione.

- Non si può aggiungere un elemento a un insieme usando una specifica posizione di un iteratore.

### Usare mappe per rappresentare associazioni tra chiavi e valori

- Le classi `HashMap` e `TreeMap` implementano l'interfaccia `Map`.
- Per trovare tutte le chiavi e i valori presenti in una mappa, si scandisce l'insieme delle chiavi e si cerca il valore corrispondente a ciascuna chiave.
- Una funzione di hash calcola un numero intero a partire da un oggetto `p`.
- Una buona funzione di hash minimizza le *collisioni*, che avvengono quando a oggetti diversi vengono associati codici di hash identici.
- Nelle vostre classi, sovrascrivete il metodo `hashCode` combinando i codici di hash delle variabili di esemplare.
- Il metodo `hashCode` deve essere compatibile con il metodo `equals`.

### Pile, code e code prioritarie

- Una pila è una raccolta di elementi con modalità di rimozione “last-in, first-out”.
- Una coda è una raccolta di elementi con modalità di rimozione “first-in, first-out”.
- Quando si estrae un elemento da una coda prioritaria viene eliminato quello avente la priorità massima.

### Risolvere problemi usando pile e code

- Per verificare se in un'espressione le parentesi sono accoppiate correttamente si può usare una pila.
- Per valutare espressioni in notazione polacca inversa si può usare una pila.
- Usando due pile si possono valutare espressioni nella notazione algebrica convenzionale.
- Per poter tornare sui propri passi e prendere una strada diversa, un algoritmo di backtracking usa una pila per ricordare le alternative ancora inesplorate.

## Elementi di libreria presentati in questo capitolo

|                                              |                                               |
|----------------------------------------------|-----------------------------------------------|
| <code>java.util.Collection&lt;E&gt;</code>   | <code>add</code>                              |
| <code>add</code>                             |                                               |
| <code>contains</code>                        |                                               |
| <code>iterator</code>                        |                                               |
| <code>remove</code>                          |                                               |
| <code>size</code>                            |                                               |
| <code>java.util.HashMap&lt;K, V&gt;</code>   |                                               |
| <code>java.util.HashSet&lt;K, V&gt;</code>   |                                               |
| <code>java.util.Iterator&lt;E&gt;</code>     |                                               |
| <code>hasNext</code>                         |                                               |
| <code>next</code>                            |                                               |
| <code>remove</code>                          |                                               |
| <code>java.util.LinkedList&lt;E&gt;</code>   |                                               |
| <code>addFirst</code>                        |                                               |
| <code>addLast</code>                         |                                               |
| <code>getFirst</code>                        |                                               |
| <code>getLast</code>                         |                                               |
| <code>removeFirst</code>                     |                                               |
| <code>removeLast</code>                      |                                               |
| <code>java.util.List&lt;E&gt;</code>         |                                               |
| <code>listIterator</code>                    |                                               |
| <code>java.util.ListIterator&lt;E&gt;</code> |                                               |
|                                              | <code>java.util.Map&lt;K, V&gt;</code>        |
|                                              | <code>get</code>                              |
|                                              | <code>keyset</code>                           |
|                                              | <code>put</code>                              |
|                                              | <code>remove</code>                           |
|                                              | <code>java.util.Objects</code>                |
|                                              | <code>hash</code>                             |
|                                              | <code>java.util.PriorityQueue&lt;E&gt;</code> |
|                                              | <code>remove</code>                           |
|                                              | <code>java.util.Queue&lt;E&gt;</code>         |
|                                              | <code>peek</code>                             |
|                                              | <code>java.util.Set&lt;E&gt;</code>           |
|                                              | <code>java.util.Stack&lt;E&gt;</code>         |
|                                              | <code>peek</code>                             |
|                                              | <code>pop</code>                              |
|                                              | <code>push</code>                             |
|                                              | <code>java.util.TreeMap&lt;K, V&gt;</code>    |
|                                              | <code>java.util.TreeSet&lt;K, V&gt;</code>    |

## Esercizi di riepilogo e approfondimento

- ★★ **R14.1.** Una fattura contiene una raccolta di articoli acquistati. Dovrebbe essere implementata mediante una lista o un insieme? Fornite spiegazioni.
- ★★ **R14.2.** Considerate un programma che gestisca un'agenda di appuntamenti. Dovrebbe inserirli in una lista, in una pila, in una coda o in una coda prioritaria? Fornite spiegazioni.
- ★★ **R14.3.** Una possibile implementazione di un'agenda prevede l'utilizzo di una mappa che metta in corrispondenza oggetti di tipo "data", usati come chiavi, e oggetti di tipo "evento". Questo, però, funziona soltanto se è previsto un unico evento per ogni possibile data. Quale altro tipo di raccolta si può utilizzare per consentire la presenza di più eventi associati a una stessa data?
- ★★ **R14.4.** Analizzate, nella documentazione dell'interfaccia `Collection`, le descrizioni dei metodi `addAll`, `removeAll`, `retainAll` e `containsAll`. Descrivete come li si possa utilizzare per implementare le comuni operazioni tra insiemi (unione, intersezione, differenza e sottoinsieme).
- \* **R14.5.** Spiegate che cosa viene visualizzato dal codice seguente. Tracciate uno schema della lista concatenata dopo ciascun passo.

```
LinkedList<String> staff = new LinkedList<>();
staff.addFirst("Harry");
staff.addFirst("Diana");
staff.addFirst("Tom");
System.out.println(staff.removeFirst());
System.out.println(staff.removeFirst());
System.out.println(staff.removeFirst());
```

- \* **R14.6.** Spiegate che cosa viene visualizzato dal codice seguente. Tracciate uno schema della lista concatenata dopo ciascun passo.

```
LinkedList<String> staff = new LinkedList<>();
staff.addFirst("Harry");
staff.addFirst("Diana");
staff.addFirst("Tom");
System.out.println(staff.removeLast());
System.out.println(staff.removeFirst());
System.out.println(staff.removeLast());
```

- \* **R14.7.** Spiegate che cosa viene visualizzato dal codice seguente. Tracciate uno schema della lista concatenata dopo ciascun passo.

```
LinkedList<String> staff = new LinkedList<>();
staff.addFirst("Harry");
staff.addLast("Diana");
staff.addFirst("Tom");
System.out.println(staff.removeLast());
System.out.println(staff.removeFirst());
System.out.println(staff.removeLast());
```

- \* **R14.8.** Spiegate che cosa viene visualizzato dal codice seguente. Tracciate uno schema della lista concatenata e la posizione dell'iteratore dopo ciascun passo.

```
LinkedList<String> staff = new LinkedList<>();
ListIterator<String> iterator = staff.listIterator();
iterator.add("Tom");
iterator.add("Diana");
iterator.add("Harry");
```

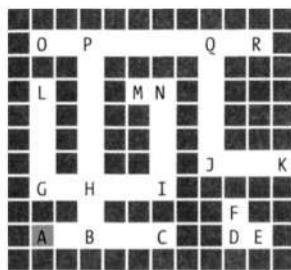
```
iterator = staff.listIterator();
if (iterator.next().equals("Tom")) { iterator.remove(); }
while (iterator.hasNext()) { System.out.println(iterator.next()); }
```

- \* **R14.9.** Spiegate che cosa viene visualizzato dal codice seguente. Tracciate uno schema della lista concatenata e la posizione dell'iteratore dopo ciascun passo.

```
LinkedList<String> staff = new LinkedList<>();
ListIterator<String> iterator = staff.listIterator();
iterator.add("Tom");
iterator.add("Diana");
iterator.add("Harry");
iterator = staff.listIterator();
iterator.next();
iterator.next();
iterator.add("Romeo");
iterator.next();
iterator.add("Juliet");
iterator = staff.listIterator();
iterator.next();
iterator.remove();
while (iterator.hasNext()) { System.out.println(iterator.next()); }
```

- \*\* **R14.10.** Data una lista concatenata di stringhe, come vi si eliminano tutti gli elementi che hanno lunghezza non superiore a tre?
- \*\* **R14.11 (per Java 8).** Risolvete nuovamente l'esercizio precedente usando il metodo `removeIf`, dopo averne letto la descrizione nella documentazione API dell'interfaccia `Collection`. Usate un'espressione lambda (descritta nella sezione Note per Java 8 10.4).
- \*\* **R14.12.** Quali vantaggi e svantaggi hanno le liste concatenate rispetto agli array?
- \*\* **R14.13.** Supponete di dover organizzare un elenco di numeri di telefono per un reparto di una società. Al momento vi sono circa 6000 dipendenti, sapete che il centralino può gestire al massimo 10000 numeri di telefono e prevedete che l'elenco venga consultato diverse centinaia di volte al giorno. Per memorizzare le informazioni usereste un vettore o una lista concatenata?
- \*\* **R14.14.** Immaginate di dover gestire un elenco di appuntamenti. Usereste una lista concatenata o un vettore di oggetti di tipo `Appointment`?
- \* **R14.15.** Supponete di scrivere un programma che simula un mazzo di carte. Le carte vengono pescate dalla cima del mazzo e distribuite ai giocatori. Quando le carte tornano nel mazzo, vengono poste al di sotto del mazzo stesso. Memorizzerete le carte in una pila o in una coda?
- \* **R14.16.** Ipotizzate che le stringhe "A" ... "Z" vengano inserite in una pila. Successivamente, vengono estratte dalla pila e inserite in una seconda pila. Infine, vengono estratte dalla seconda pila e visualizzate. In quale ordine?
- \* **R14.17.** Qual è la differenza fra un insieme e una mappa?
- \*\* **R14.18.** L'unione di due insiemi  $A$  e  $B$  è l'insieme di tutti gli elementi che sono contenuti in  $A$ , in  $B$  o in entrambi. L'intersezione è, invece, l'insieme di tutti gli elementi che sono contenuti sia in  $A$  sia in  $B$ . Come si possono calcolare l'unione e l'intersezione di due insiemi, usando i metodi `add` e `contains`, oltre a un iteratore?
- \*\* **R14.19.** Come si possono calcolare l'unione e l'intersezione di due insiemi usando i metodi forniti dall'interfaccia `java.util.Set`, ma senza usare un iteratore? Consultate la documentazione API dell'interfaccia nella libreria standard.

- \* **R14.20.** Una mappa può avere due chiavi associate allo stesso valore? E due valori associati alla stessa chiave?
- \*\* **R14.21.** Una mappa può essere realizzata mediante un insieme di coppie (*chiave, valore*). Date una spiegazione.
- \* **R14.22 (per Java 8).** Come si possono visualizzare tutte le coppie chiave/valore presenti in una mappa usando il metodo `keySet`? E usando il metodo `entrySet`? E il metodo `forEach` fornendo un'espressione lambda (descritta nella sezione Note per Java 8 10.4)?
- \*\*\* **R14.23.** Verificate che il codice di hash della stringa "Juliet" sia quello riportato nella Tabella 6.
- \*\*\* **R14.24.** Verificate che le stringhe "VII" e "Ugh" abbiano lo stesso codice di hash.
- \* **R14.25.** Considerate l'algoritmo di uscita da un labirinto visto nel Paragrafo 14.6.4, immaginando di partire dalla posizione A e di inserire nella pila le direzioni nell'ordine ovest, sud, est e nord. In quale ordine vengono visitate le posizioni indicate dalle lettere nel labirinto qui rappresentato?



- \* **R14.26.** Ripetete l'esercizio precedente, usando una coda invece di una pila.

## Esercizi di programmazione

- \*\* **E14.1.** Scrivete un metodo

```
public static void downsize(LinkedList<String> employeeNames, int n)
```

che elimini da una lista concatenata un impiegato ogni n.

- \*\* **E14.2.** Scrivete un metodo

```
public static void reverse(LinkedList<String> strings)
```

che inverta i dati presenti in una lista concatenata.

- \*\* **E14.3.** Realizzate il *crivello di Eratostene*, un metodo per calcolare i numeri primi noto agli antichi greci. Scegliete un numero *n*: questo metodo calcolerà tutti i numeri primi fino a *n*. Come prima cosa inserite in un insieme tutti i numeri da 2 a *n*. Poi, cancellate tutti i multipli di 2 (eccetto 2); vale a dire 4, 6, 8, 10, 12, ... Dopodiché, cancellate tutti i multipli di 3 (eccetto 3), cioè, 6, 9, 12, 15, ... Arrivate fino a  $n^{1/2}$ , quindi visualizzate l'insieme.

- \*\* **E14.4.** Scrivete un programma che usi una mappa in cui sia le chiavi sia i valori sono stringhe: rispettivamente, i nomi degli studenti e i loro voti in un esame. Chiedete all'utente del programma se vuole inserire o rimuovere studenti, modificarne il voto o stampare tutti i voti. La visualizzazione dovrebbe essere ordinata per nome e avere un aspetto simile a questo:

Carl: B+  
Joe: C  
Sarah: A

- \*\*\* **E14.5.** Scrivete un programma che legga un file di codice sorgente Java e generi un elenco di tutti gli identificatori presenti, visualizzando, accanto a ciascuno di essi, i numeri delle righe in cui compare. Per semplicità considereremo che qualsiasi stringa costituita soltanto da lettere, cifre numeriche e caratteri di sottolineatura sia un identificatore. Dichiarate la variabile `Scanner` in per leggere il file e invocate il metodo `in.useDelimiter("[^A-Za-z0-9_]+")`, in modo che ogni invocazione di `next` restituisca un identificatore.
- \*\* **E14.6 (per Java 8).** Leggete da un file tutte le parole presenti e aggiungetele a una mappa le cui chiavi siano le lettere iniziali delle parole e i cui valori siano insiemi contenenti le parole che iniziano con quella stessa lettera. Quindi, visualizzate gli insiemi di parole in ordine alfabetico.  
Risolvete l'esercizio in due modi, uno che usi il metodo `merge` (descritto nella sezione Note per Java 8 14.1) e uno che aggiorni la mappa come nella sezione Esempi completi 14.1.
- \*\* **E14.7 (per Java 8).** Leggete da un file tutte le parole presenti e aggiungetele a una mappa le cui chiavi siano le lunghezze delle parole e i cui valori siano stringhe composte da parole separate da virgole, con parole aventi tutte la stessa lunghezza. Quindi, visualizzate tali stringhe in ordine crescente di lunghezza delle loro singole parole componenti.  
Risolvete l'esercizio in due modi, uno che usi il metodo `merge` (descritto nella sezione Note per Java 8 14.1) e uno che aggiorni la mappa come nella sezione Esempi completi 14.1.
- \*\* **E14.8.** Usate una pila per invertire le parole di una frase. Continuate a leggere parole, aggiungendole alla pila, fin quando non trovate una parola che termina con un punto. A questo punto estraete tutte le parole dalla pila e visualizzatele, poi ripetete la procedura fino all'esaurimento dei dati in ingresso. Ad esempio, questa frase

Mary had a little lamb. Its fleece was white as snow

deve essere trasformata nella seguente

Lamb little a had mary. Snow as white was fleece its.

Fate attenzione alle lettere maiuscole e al posizionamento del punto che termina la frase.

- \* **E14.9.** Dovete scomporre un numero intero nelle sue singole cifre, trasformando, ad esempio, il numero 1729 nella sequenza di cifre 1, 7, 2 e 9. L'ultima cifra del numero  $n$  si ottiene facilmente calcolando  $n \% 10$ , ma procedendo in questo modo si ottengono le cifre in ordine inverso. Risolvete il problema usando una pila. Il programma deve chiedere all'utente di fornire un numero intero, per poi visualizzarne le singole cifre separate da spazi.
- \*\* **E14.10.** In occasione di manifestazioni particolari, il proprietario di una casa noleggia posti auto nel suo vialetto di casa, che può essere rappresentato da una pila, con il consueto comportamento "last-in, first-out". Quando il proprietario di un'automobile se ne va e la sua automobile non è l'ultima, tutte quelle che la bloccano devono essere spostate temporaneamente sulla strada, per poi rientrare nel vialetto. Scrivete un programma che simuli questo comportamento, usando una pila per il vialetto e una per la strada, con numeri interi a rappresentare le targhe delle automobili. Un numero positivo inserisce un'automobile nel vialetto, un numero negativo la fa uscire definitivamente e il numero zero termina la simulazione. Visualizzate il contenuto del vialetto al termine di ciascuna operazione.

- \* **E14.11.** Dovete realizzare un “elenco di cose da fare” (*to do list*). A ciascun compito viene assegnata una priorità, un numero intero da 1 a 9, e una descrizione. Quando l’utente digita il comando `add priorità descrizione` il programma aggiunge una cosa da fare, mentre quando l’utente digita `next` il programma elimina e visualizza la cosa da fare più urgentemente. Il comando `quit` termina il programma. Risolvete il problema usando una coda prioritaria.
- \* **E14.12.** Scrivete un programma che legga un testo da un file e lo suddivida in singole parole. Inserite le parole in un insieme realizzato mediante un albero. Dopo aver letto tutti i dati, visualizzate tutte le parole, seguite dalla dimensione dell’insieme risultante. Questo programma determina, quindi, quante parole diverse sono presenti in un testo.
- \* **E14.13.** Leggendo tutte le parole di un file di testo di grandi dimensioni (come il romanzo “War and Peace”, *Guerra e pace*, disponibile in Internet), inseritele in due insiemi, uno realizzato mediante tabella hash e uno realizzato mediante albero. Misurate i tempi di esecuzione: quale struttura agisce più velocemente?
- \* **E14.14.** Realizzate, nella classe `BankAccount` del Capitolo 8, metodi `hashCode` e `equals` che siano fra loro compatibili. Verificate la correttezza dell’implementazione del metodo `hashCode` visualizzando codici di hash e aggiungendo oggetti `BankAccount` a un insieme realizzato con tabella hash.
- \*\* **E14.15.** Un punto geometrico dotato di etichetta è caratterizzato dalle coordinate `x` e `y`, oltre che dall’etichetta, sotto forma di stringa. Progettate la classe `LabeledPoint` dotata del costruttore `LabeledPoint(int x, int y, String label)` e dei metodi `hashCode` e `equals`: due punti sono considerati uguali quando si trovano nella stessa posizione e hanno la stessa etichetta.
- \*\* **E14.16.** Realizzate una diversa versione della classe `LabeledPoint` vista nell’esercizio precedente, memorizzando la posizione del punto in un oggetto di tipo `java.awt.Point`. I metodi `hashCode` e `equals` devono invocare gli omonimi metodi della classe `Point`.
- \*\* **E14.17.** Modificate la classe `LabeledPoint` dell’Esercizio E14.15 in modo che implementi l’interfaccia `Comparable`. Fate in modo che i punti vengano ordinati innanzitutto in base alla loro coordinata `x`; se due punti hanno la stessa coordinata `x`, ordinatevi in base alla loro coordinata `y`; se due punti hanno le stesse coordinate, ordinatevi in base alla loro etichetta. Scrivete un programma di collaudo che verifichi tutti i casi, inserendo punti in un `TreeSet`.
- \* **E14.18.** Aggiungete al valutatore di espressioni visto nel Paragrafo 14.6.3 l’operatore `%`, che calcola il resto della divisione intera.
- \*\* **E14.19.** Aggiungete al valutatore di espressioni visto nel Paragrafo 14.6.3 l’operatore `^`, che effettua l’elevamento a potenza. Ad esempio,  $2^3$  ha come risultato 8. Come in matematica, l’elevamento a potenza deve essere valutato da destra verso sinistra, cioè  $2^3^2$  è uguale a  $2^3 \cdot 2^2$  e non a  $(2^3)^2$  (quest’ultima quantità si può calcolare come  $2^{(3 \times 2)}$ ).
- \* **E14.20.** Scrivete un programma che verifichi se una sequenza di marcatori HTML è annidata correttamente. Per ogni marcitore di apertura, come `<p>`, ci deve essere un marcitore di chiusura, `</p>`. All’interno di una coppia di marcatori, come `<p> . . . </p>`, possono essere presenti altri marcatori, come in questo esempio:

```
<p> <ul> <li> </li> </ul> <a> </a> </p>
```

I marcatori più interni deve essere racchiusi tra quelli più esterni. Il programma deve elaborare un file contenente marcatori: per semplicità, ipotizzate che i marcatori siano separati da spazi e che al loro interno non ci sia altro testo, ma solo altri marcatori.

- ★ **E14.21.** Modificate il solutore di labirinti visto nel Paragrafo 14.6.4 in modo che possa gestire anche labirinti con cicli. Utilizzate un insieme di incroci visitati: quando arrivate in un incrocio già visitato in precedenza, trattatelo come un vicolo cieco e non aggiungete alla pila le sue alternative.
- ★★ **E14.22.** In un programma per disegnare, l'operazione di "riempimento per inondazione" (*flood fill*) assegna un determinato colore a tutti i pixel vuoti di un disegno, fermandosi quando raggiunge pixel occupati. In questo esercizio svilupperete una semplice variante di questo algoritmo, riempiendo per inondazione un array di numeri interi di dimensione  $10 \times 10$ , inizialmente riempito di zeri. Chiedete all'utente la posizione iniziale: riga e colonna. Inserite in una pila la coppia <riga, colonna> (vi servirà una semplice classe *Pair*).

Quindi, ripetete queste operazioni finché la pila non è vuota.

- La coppia <riga, colonna> presente in cima alla pila viene estratta.
- Se la posizione corrispondente è ancora vuota, riempitela, usando via via i numeri 1, 2, 3, ecc., in modo da visualizzare l'ordine di riempimento delle celle.
- Le coordinate delle celle adiacenti in direzione nord, est, sud e ovest, che non siano state riempite, vengono inserite nella pila.

Al termine, visualizzate l'intero array.

Sul sito web dedicato al libro si trova una raccolta di progetti di programmazione più complessi.

# 15

## Programmazione generica



---

### Obiettivi del capitolo

---

- Capire gli obiettivi della programmazione generica
- Essere in grado di realizzare classi e metodi generici
- Comprendere il meccanismo di esecuzione di metodi generici all'interno della macchina virtuale Java
- Conoscere le limitazioni relative alla programmazione generica in Java

La programmazione generica riguarda la progettazione e realizzazione di strutture di dati e di algoritmi che siano in grado di funzionare con tipi di dati diversi. Conoscete già la classe generica `ArrayList`, i cui esemplari possono contenere dati di qualunque tipo. In questo capitolo imparerete a realizzare vostre classi generiche.

## 15.1 Classi generiche e tipi parametrici

La *programmazione generica* consiste nella creazione di costrutti di programmazione che possano essere utilizzati con molti tipi di dati diversi. Ad esempio, i programmatore della libreria Java che hanno realizzato la classe `ArrayList` hanno sfruttato le tecniche della programmazione generica: come risultato, è possibile creare vettori che contengano elementi di tipi diversi, come `ArrayList<String>`, `ArrayList<BankAccount>` e così via.

Una classe generica ha uno o più tipi parametrici.

Nella dichiarazione di una classe generica, occorre specificare una variabile di tipo per ogni tipo parametrico. Ecco come viene dichiarata la classe `ArrayList` nella libreria standard di Java, usando la *variabile di tipo* `E` per rappresentare il tipo degli elementi:

```
public class ArrayList<E>
{
    public ArrayList() {...}
    public void add(E element) {...}
    ...
}
```

In questo caso, `E` è una variabile di tipo, non una parola riservata di Java; invece di `E` potrete usare un nome diverso, come `ElementType`, ma per le variabili di tipo si è soliti usare nomi brevi e composti di lettere maiuscole.

Per poter usare una classe generica, dovete fornire un tipo effettivo che sostituisca il tipo parametrico; si può usare il nome di una classe oppure di un'interfaccia, come in questi esempi:

```
ArrayList<BankAccount>
ArrayList<Measurable>
```

Non si può, però, sostituire un tipo parametrico con uno degli otto tipi di dati primitivi, quindi sarebbe un errore creare un oggetto di tipo `ArrayList<double>`: usate la corrispondente classe involucro, `ArrayList<Double>`.

Quando create un esemplare di una classe generica, il tipo di dato che indicate va a sostituire tutte le occorrenze della variabile di tipo utilizzata nella dichiarazione della classe. Ad esempio, nel metodo `add` di un oggetto di tipo `ArrayList<BankAccount>`, la variabile di tipo, `E`, viene sostituita dal tipo `BankAccount`:

```
public void add(BankAccount element)
```



### Auto-valutazione

1. La libreria standard mette a disposizione la classe `HashMap<K, V>`, dove `K` è il tipo della chiave e `V` è il tipo del valore. Come esemplare di tale classe, costruite una mappa che memorizzi associazioni tra stringhe e numeri interi.

## 15.2 Realizzare tipi generici

In questo paragrafo imparerete a realizzare vostre classi generiche. Inizieremo con una classe generica molto semplice, che memorizza *coppie* di oggetti, ciascuno dei quali può essere di tipo qualsiasi. Ad esempio:

```
Pair<String, Integer> result = new Pair<>("Harry Morgan", 1729);
```

I metodi `getFirst` e `getSecond` restituiscono il primo e il secondo valore memorizzati nella coppia.

```
String name = result.getFirst();
Integer number = result.getSecond();
```

Questa classe può essere utile quando si realizza un metodo che calcola e deve restituire due valori: un metodo non può restituire contemporaneamente un esemplare di `String` e un esemplare di `Integer`, mentre può restituire un singolo oggetto di tipo `Pair<String, Integer>`.

La classe generica `Pair` richiede due tipi parametrici, uno per il tipo del primo elemento e uno per il tipo del secondo elemento.

Dobbiamo scegliere le variabili per questi tipi parametrici. Solitamente per le variabili di tipo si usano nomi brevi e composti di sole lettere maiuscole, come in questi esempi:

Variabile di tipo	Significato
E	Tipo di un elemento in una raccolta
K	Tipo di una chiave in una mappa
V	Tipo di un valore in una mappa
T	Tipo generico
S, U	Ulteriori tipi generici

Le variabili di tipo di una classe generica vanno indicate dopo il nome della classe, racchiuse tra parentesi angolari:  
il nome della classe e sono racchiuse tra parentesi angolari.

Per indicare i tipi generici delle variabili di esemplare, dei parametri dei metodi e dei valori da essi restituiti, usate le variabili di tipo.

Le variabili di tipo di una classe generica vanno indicate dopo il nome della classe, racchiuse tra parentesi angolari:

```
public class Pair<T, S>
```

Nelle dichiarazioni delle variabili di esemplare e dei metodi della classe `Pair`, usiamo la variabile di tipo `T` per indicare il tipo del primo elemento e la variabile di tipo `S` per il tipo del secondo elemento:

```
public class Pair<T, S>
{
    private T first;
    private S second;

    public Pair(T firstElement, S secondElement)
    {
        first = firstElement;
        second = secondElement;
    }
}
```

```

    public T getFirst() { return first; }
    public S getSecond() { return second; }
}

```

Alcuni trovano più semplice partire dalla dichiarazione di una classe normale, scegliendo tipi effettivi al posto delle variabili di tipo, come in questo esempio:

```

public class Pair // iniziamo con una coppia di String e Integer
{
    private String first;
    private Integer second;

    public Pair(String firstElement, Integer secondElement)
    {
        first = firstElement;
        second = secondElement;
    }
    public String getFirst() { return first; }
    public Integer getSecond() { return second; }
}

```

A questo punto è facile sostituire tutti i tipi `String` con la variabile di tipo `S` e tutti i tipi `Integer` con la variabile di tipo `T`.

Ciò completa la definizione della classe generica `Pair`, che ora è pronta per essere utilizzata ovunque abbiate bisogno di creare una coppia composta da due oggetti di tipo qualsiasi. L'esempio che segue mostra come usare un oggetto di tipo `Pair` per progettare un metodo che restituisca due valori.

### File `Pair.java`

```

/*
 * Questa classe memorizza una coppia di elementi di tipi diversi.
 */
public class Pair<T, S>
{
    private T first;
    private S second;

    /**
     * Costruisce una coppia contenente i due elementi ricevuti.
     * @param firstElement il primo elemento
     * @param secondElement il secondo elemento
     */
    public Pair(T firstElement, S secondElement)
    {
        first = firstElement;
        second = secondElement;
    }

    /**
     * Restituisce il primo elemento di questa coppia.
     * @return il primo elemento
     */
    public T getFirst() { return first; }
}

```

## Sintassi di Java

### 15.1 Dichiarazione di una classe generica

#### Sintassi

```
modalitàDiAccesso class NomeClasseGenerica<VariabileDiTipo1, VariabileDiTipo2, ...>
{
    variabili di esemplare
    costruttori
    metodi
}
```

#### Esempio

Specificate una variabile per ogni tipo parametrico.

Metodo che restituisce un valore di tipo parametrico.

```
public class Pair<T, S>
{
    private T first;
    private S second;
    ...
    public T getFirst() { return first; }
    ...
}
```

Variabili di esemplare di un tipo di dato parametrico.

```
/**
 * Restituisce il secondo elemento di questa coppia.
 * @return il secondo elemento
 */
public S getSecond() { return second; }

public String toString() { return "(" + first + ", " + second + ")"; }
```

#### File PairDemo.java

```
public class PairDemo
{
    public static void main(String[] args)
    {
        String[] names = { "Tom", "Diana", "Harry" };
        Pair<String, Integer> result = firstContaining(names, "a");
        System.out.println(result.getFirst());
        System.out.println("Expected: Diana");
        System.out.println(result.getSecond());
        System.out.println("Expected: 1");
    }

    /**
     * Restituisce la prima stringa contenente una stringa assegnata,
     * oltre al suo indice nell'array.
     * @param strings un array di stringhe
     * @param sub una stringa
     * @return una coppia (strings[i], i), dove strings[i] è la prima
     *         stringa in strings contenente sub, oppure una coppia
     *         (null, -1) se non si trovano corrispondenze
    */
}
```

```

    */
    public static Pair<String, Integer> firstContaining(
        String[] strings, String sub)
    {
        for (int i = 0; i < strings.length; i++)
        {
            if (strings[i].contains(sub))
            {
                return new Pair<String, Integer>(strings[i], i);
            }
        }
        return new Pair<String, Integer>(null, -1);
    }
}

```

### Esecuzione del programma

```

Diana
Expected: Diana
1
Expected: 1

```



### Auto-valutazione

2. Come usereste la classe generica `Pair` per costruire una coppia contenente le stringhe "Hello" e "World"?
3. Che differenza c'è tra un oggetto di tipo `ArrayList<Pair<String, Integer>>` e uno di tipo `Pair<ArrayList<String>, Integer>`?

## 15.3 Metodi generici

Un metodo generico è un metodo avendo un tipo parametrico.

Un metodo generico è un metodo che ha un tipo parametrico e si può anche trovare in una classe che, per se stessa, non è generica. Potete pensare a un tale metodo come a un insieme di metodi che differiscono tra loro soltanto per uno o più tipi di dati. Ad esempio, potremmo voler dichiarare un metodo che possa visualizzare un array di qualsiasi tipo:

```

public class ArrayUtil
{
    /**
     * Visualizza tutti gli elementi contenuti in un array.
     * @param a l'array da visualizzare
     */
    public <T> static void print(T[] a)
    {
        ...
    }
    ...
}

```

Come detto nel paragrafo precedente, spesso è più facile capire come si realizza un metodo generico partendo da un esempio concreto. Questo metodo visualizza tutti gli elementi presenti in un array di *stringhe*.

```
public class ArrayUtil
{
    public static void print(String[] a)
    {
        for (String e : a)
        {
            System.out.print(e + " ");
        }
        System.out.println();
    }
    ...
}
```

I tipi parametrici di un metodo generico vanno scritti tra i modificatori e il tipo del valore restituito dal metodo.

Per trasformare tale metodo in un metodo generico, sostituite il tipo `String` con un tipo parametrico, diciamo `E`, che rappresenti il tipo degli elementi dell'array. Aggiungete un elenco di tipi parametrici, racchiuso tra parentesi angolari, tra i modificatori (in questo caso `public` e `static`) e il tipo del valore restituito (in questo caso `void`):

```
public static <E> void print(E[] a)
{
    for (E e : a)
    {
        System.out.print(e + " ");
    }
    System.out.println();
}
```

Quando invocate il metodo generico, non dovete specificare i tipi effettivi da usare al posto dei tipi parametrici (e in questo aspetto i metodi generici differiscono dalle classi generiche): invocate semplicemente il metodo con i parametri appropriati e il compilatore metterà in corrispondenza i tipi parametrici con i tipi dei parametri. Ad esempio, considerate questa invocazione di metodo:

```
Rectangle[] rectangles = ...;
ArrayUtil.print(rectangles);
```

Quando invocate un metodo generico, non dovete specificare esplicitamente i tipi da usare al posto dei tipi parametrici.

Il tipo del parametro `rectangles` è `Rectangle[]`, mentre il tipo della variabile parametro è `E[]`: il compilatore ne deduce che il tipo effettivo da usare per `E` è `Rectangle`.

Questo particolare metodo generico è un metodo statico inserito in una classe normale (non generica), ma potete definire anche metodi generici che non siano statici. Potete, infine, definire metodi generici all'interno di classi generiche.

Come nel caso delle classi generiche, non potete usare tipi primitivi per sostituire tipi parametrici. Il metodo generico `print` può, quindi, visualizzare array di qualsiasi tipo, *eccetto* array di uno degli otto tipi primitivi. Ad esempio, non si può usare il metodo `print` per visualizzare un array di tipo `int[]`, ma questo non è un grande problema: realizzate semplicemente, oltre al metodo generico `print`, un metodo `print(int[] a)`.



## Auto-valutazione

4. Cosa fa esattamente il metodo generico `print` quando fornite come parametro un array di oggetti di tipo `BankAccount` contenente due conti bancari aventi saldo uguale a zero?
5. Il metodo `getFirst` della classe `Pair` è un metodo generico?

**Sintassi di Java****15.2 Dichiarazione di un metodo generico****Sintassi**

```
modificatori <VariabileDiTipo1, VariabileDiTipo2,...> tipoRestituito nomeMetodo(parametri)
{
    corpo
}
```

**Esempio**

Specificate la variabile di tipo prima del tipo restituito.

```
public static <E> void print(E[] a)
{
    for (E e : a)
        System.out.print(e + " ");
    System.out.println();
}
```

Variabile locale di un tipo  
di dato parametrico.

**15.4 Vincolare i tipi parametrici**

I tipi parametrici possono essere  
soggetti a vincoli.

Spesso è necessario specificare quali tipi possano essere usati in una classe generica oppure in un metodo generico. Considerate, ad esempio, un metodo generico, `min`, che abbia il compito di trovare l'elemento di valore minimo presente in un array di oggetti. Come è possibile trovare l'elemento di valore minimo quando non si ha alcuna informazione in merito al tipo degli elementi? Serve un meccanismo che consenta di confrontare gli elementi dell'array. Una soluzione consiste nel richiedere che gli elementi appartengano a un tipo che implementa l'interfaccia `Comparable`. In una situazione come questa, dobbiamo quindi *vinciare* il tipo parametrico.

```
public static <E extends Comparable> E min(E[] a)
{
    E smallest = a[0];
    for (int i = 1; i < a.length; i++)
    {
        if (a[i].compareTo(smallest) < 0) { smallest = a[i]; }
    }
    return smallest;
}
```

Potete invocare `min` con un array di tipo `String[]` ma non con un array di tipo `Rectangle[]`: la classe `String` realizza `Comparable`, ma `Rectangle` no.

La limitazione al tipo `Comparable` è necessaria per poter invocare il metodo `compareTo`: se non fosse stato specificato il vincolo, il metodo `min` non sarebbe stato compilato, perché non sarebbe stato lecito invocare `compareTo` con l'oggetto `a[i]`, del cui tipo non si avrebbe avuto alcuna informazione (in realtà, la stessa interfaccia `Comparable` è un tipo generico, ma per semplicità l'abbiamo utilizzata senza fornire un tipo come parametro; per maggiori informazioni, consultate Argomenti avanzati 15.1).

Vi capiterà raramente di dover indicare due o più vincoli. In tal caso, separateli con il carattere `&`, come in questo esempio:

`<E extends Comparable & Cloneable>`

La parola riservata `extends`, quando viene applicata ai tipi parametrici, significa in realtà “estende o implementa”; i vincoli possono essere classi o interfacce e i tipi parametrici possono essere sostituiti con il tipo effettivo di una classe o di un’interfaccia.



## Auto-valutazione

6. Come vincolereste il tipo parametrico di una classe `TreeSet` generica?
7. Modificate il metodo `min` in modo che identifichi l’elemento minimo all’interno di un array di elementi che realizzano l’interfaccia `Measurable` vista nel Capitolo 10.



## Errori comuni 15.1

### Genericità e ereditarietà

Se `SavingsAccount` è una sottoclasse di `BankAccount`, allora `ArrayList<SavingsAccount>` è una sottoclasse di `ArrayList<BankAccount>?` Anche se forse ne sarete sorpresi, la risposta è no: il legame di ereditarietà presente fra i tipi parametrici non genera un legame di ereditarietà fra le classi generiche corrispondenti e, quindi, non esiste alcuna relazione di ereditarietà tra `ArrayList<SavingsAccount>` e `ArrayList<BankAccount>`.

Questa limitazione è assolutamente necessaria per consentire la verifica della corrispondenza tra i tipi. Immaginate, infatti, che fosse possibile assegnare un oggetto di tipo `ArrayList<SavingsAccount>` a una variabile di tipo `ArrayList<BankAccount>`, in questo modo:

```
ArrayList<SavingsAccount> savingsAccounts
    = new ArrayList<SavingsAccount>();
// quanto segue non è lecito, ma supponiamo che lo sia
ArrayList<BankAccount> bankAccounts = savingsAccounts;
BankAccount harrysChecking = new CheckingAccount();
// CheckingAccount è una diversa sottoclasse di BankAccount
bankAccounts.add(harrysChecking);
// va bene, si possono aggiungere oggetti di tipo BankAccount
```

Ma `bankAccounts` e `savingsAccounts` fanno riferimento al medesimo vettore! Se l’assegnazione indicata in grassetto fosse lecita, saremmo in grado di aggiungere un oggetto di tipo `CheckingAccount` a un contenitore di tipo `ArrayList<SavingsAccount>`.

In molte situazioni queste limitazioni possono essere superate usando un carattere jolly (*wildcard*), come descritto in Argomenti avanzati 15.1.



## Argomenti avanzati 15.1

### Tipi con carattere jolly (*wildcard*)

Spesso si ha la necessità di formulare vincoli un po’ complessi per i tipi parametrici: per questo scopo sono stati inventati i tipi con carattere jolly (*wildcard*), che può essere usato in tre diversi modi.

Nome	Sintassi	Significato
Vincolo con limite inferiore	? extends B	Qualsiasi sottotipo di B
Vincolo con limite superiore	? super B	Qualsiasi supertipo di B
Nessun vincolo	?	Qualsiasi tipo

Un tipo specificato con carattere jolly è un tipo che può rimanere sconosciuto. Ad esempio, nella classe `LinkedList<E>` si può definire il metodo seguente, che aggiunge alla fine della lista concatenata tutti gli elementi contenuti in `other`.

```
public void addAll(LinkedList<? extends E> other)
{
    ListIterator<E> iter = other.listIterator();
    while (iter.hasNext()) { add(iter.next()); }
}
```

Il metodo `addAll` non richiede che il tipo degli elementi di `other` sia un qualche tipo specifico: consente l'utilizzo di qualsiasi tipo che sia un sottotipo di `E`. Ad esempio, potete usare `addAll` per aggiungere a un esemplare di `LinkedList<BankAccount>` tutti gli elementi contenuti in un esemplare di `LinkedList<SavingsAccount>`.

Per vedere un tipo *wildcard* con vincolo di tipo `super`, torniamo al metodo `min` del paragrafo precedente. Ricordate che `Comparable` è un'interfaccia generica e il tipo che la rende generica serve a specificare il tipo del parametro del metodo `compareTo`.

```
public interface Comparable<T>
{
    int compareTo(T other);
}
```

Di conseguenza, potremmo voler specificare un tipo vincolato:

```
public static <E extends Comparable<E>> E min(E[] a)
```

Questo vincolo, però, è troppo restrittivo. Immaginate che la classe `BankAccount` realizzzi l'interfaccia `Comparable<BankAccount>`: di conseguenza, anche la sua sottoclasse `SavingsAccount` realizza `Comparable<BankAccount>` e *non* `Comparable<SavingsAccount>`. Se volete usare il metodo `min` con un array di tipo `SavingsAccount[]`, allora il tipo che specifica il parametro dell'interfaccia `Comparable` deve essere *qualsiasi supertipo* del tipo di elemento dell'array:

```
public static <E extends Comparable<? super E>> E min(E[] a)
```

Ecco, invece, un esempio di carattere jolly che specifica l'assenza di vincoli. La classe `Collections` definisce il metodo seguente:

```
public static void reverse(List<?> list)
```

Una dichiarazione di questo tipo è, in pratica, un'abbreviazione per la seguente:

```
public static <T> void reverse(List<T> list)
```

## 15.5 Cancellazione dei tipi (*type erasure*)

La macchina virtuale elimina i tipi parametrici, sostituendoli con i loro vincoli o con `Object`.

Dato che i tipi generici sono stati introdotti nel linguaggio Java soltanto di recente, la macchina virtuale che esegue programmi Java non lavora con classi o metodi generici: i tipi parametrici vengono “cancellati”, cioè sostituiti da tipi Java ordinari. Ciascun tipo parametrico è sostituito dal relativo vincolo, oppure da `Object` se non è vincolato.

Ad esempio, la classe generica `Pair<T, S>` viene sostituita dalla seguente classe “grezza” (*raw*):

```
public class Pair
{
    private Object first;
    private Object second;

    public Pair(Object firstElement, Object secondElement)
    {
        first = firstElement;
        second = secondElement;
    }
    public Object getFirst() { return first; }
    public Object getSecond() { return second; }
}
```

Come potete vedere, i tipi parametrici, `T` e `S`, sono stati sostituiti da `Object`; il risultato è una classe ordinaria.

Ai metodi generici viene applicato il medesimo procedimento. Dopo la cancellazione dei tipi parametrici, il metodo `min` del paragrafo precedente si trasforma in un metodo ordinario; notate come, in questo esempio, il tipo parametrico sia sostituito dal suo vincolo, l’interfaccia `Comparable`.

```
public static Comparable min(Comparable[] a)
{
    Comparable smallest = a[0];
    for (int i = 1; i < a.length; i++)
    {
        if (a[i].compareTo(smallest) < 0) { smallest = a[i]; }
    }
    return smallest;
}
```

Non si possono costruire oggetti o array di un tipo generico..

Conoscere l’esistenza dei tipi grezzi aiuta a capire i limiti della programmazione generica in Java. Ad esempio, non potete costruire esemplari di un tipo generico. Il seguente metodo, che tenta di riempire un array con copie di oggetti predefiniti, sarebbe sbagliato:

```
public static <E> void fillWithDefaults(E[] a)
{
    for (int i = 0; i < a.length; i++)
    {
        a[i] = new E(); // ERRORE
    }
}
```

Per capire per quale motivo ciò costituisca un problema, eseguite il procedimento di cancellazione dei tipi parametrici, come se fosse il compilatore:

```
public static void fillWithDefaults(Object[] a)
{
    for (int i = 0; i < a.length; i++)
    {
        a[i] = new Object(); // inutile
    }
}
```

Ovviamente, se costruire un array di tipo `Rectangle[]`, non volete che il metodo lo riempia di esemplari di `Object`, ma, dopo la cancellazione dei tipi parametrici, questo è ciò che farebbe il codice che abbiamo scritto.

In situazioni come questa, il compilatore segnala un errore, per cui dovete necessariamente trovare un modo diverso per risolvere il vostro problema. In questo particolare esempio, potete fornire uno specifico oggetto predefinito:

```
public static <E> void fillWithDefaults(E[] a, E defaultValue)
{
    for (int i = 0; i < a.length; i++)
    {
        a[i] = defaultValue;
    }
}
```

Analogamente, non è possibile costruire un array di un tipo generico.

```
public class Stack<E>
{
    private E[] elements;
    ...
    public Stack()
    {
        elements = new E[MAX_SIZE]; // ERRORE
    }
}
```

Dato che l'espressione che costruisce l'array, `new E[]`, diventerebbe, dopo la cancellazione dei tipi parametrici, `new Object[]`, il compilatore non la consente. Come soluzione, si può usare un vettore:

```
public class Stack<E>
{
    private ArrayList<E> elements;
    ...
    public Stack()
    {
        elements = new ArrayList<E>(); // così va bene
    }
}
```

oppure si può usare un array di `Object`, inserendo un cast ogni volta che si legge un valore contenuto nell'array:

```
public class Stack<E>
{
    private Object[] elements;
    private int size;
    ...
    public Stack()
    {
        elements = new Object[MAX_SIZE]; // anche così va bene
    }
    ...
    public E pop()
    {
        size--;
        return (E) elements[size];
    }
}
```

Il cast genera un *avvertimento (warning)* in fase di compilazione perché non è verificabile.

Queste limitazioni sono, francamente, imbarazzanti: ci auguriamo che le future versioni di Java non effettuino più la cancellazione dei tipi parametrici, in modo da poter eliminare le attuali restrizioni che ne sono, appunto, conseguenza.



## Auto-valutazione

8. Cosa si ottiene applicando la cancellazione dei tipi parametrici al metodo `print` visto nel Paragrafo 15.3?
9. Si potrebbe realizzare una pila in questo modo?

```
public class Stack<E>
{
    private E[] elements;
    ...
    public Stack()
    {
        elements = (E[]) new Object[MAX_SIZE];
    }
}
```



## Errori comuni 15.2

### Usare tipi generici in un contesto statico

Non si possono usare tipi parametrici per dichiarare variabili statiche, metodi statici o classi interne statiche. Ad esempio, quanto segue non è lecito:

```
public class LinkedList<E>
{
    private static E defaultValue; // ERRORE
    ...
    public static List<E> replicate(E value, int n) {...} // ERRORE
```

```
private static class Node { public E data; public Node next; } // ERRORE
}
```

Nel caso di variabili statiche, questa limitazione è molto stringente. Dopo la cancellazione dei tipi generici, esiste un'unica variabile, `LinkedList.defaultValue`, mentre la dichiarazione della variabile statica lascerebbe falsamente intendere che ne esista una diversa per ogni diverso tipo di `LinkedList<E>`.

Per i metodi statici e le classi interne statiche esiste una semplice alternativa: aggiungere un tipo parametrico.

```
public class LinkedList<E>
{
    ...
    public static <T> List<T> replicate(T value, int n) {...} // va bene
    private static class Node<T> { public T data; public Node<T> next; }
                                // va bene
}
```

## Riepilogo degli obiettivi di apprendimento

### Classi generiche e tipi parametrici

- Una classe generica ha uno o più tipi parametrici.
- I tipi parametrici possono essere sostituiti, all'atto della creazione di esemplari, con nomi di classi o di interfacce.

### Realizzazione di classi e interfacce generiche

- Le variabili di tipo di una classe generica vanno indicate dopo il nome della classe e sono racchiuse tra parentesi angolari.
- Per indicare i tipi generici delle variabili di esemplare, dei parametri dei metodi e dei valori da essi restituiti, usate le variabili di tipo.

### Realizzazione di metodi generici

- Un metodo generico è un metodo avente un tipo parametrico.
- I tipi parametrici di un metodo generico vanno scritti tra i modificatori e il tipo del valore restituito dal metodo.
- Quando invocate un metodo generico, non dovete specificare esplicitamente i tipi da usare al posto dei tipi parametrici.

### Espressione di vincoli per i tipi parametrici

- I tipi parametrici possono essere soggetti a vincoli.

### Limitazioni sulla programmazione generica in Java imposte dalla cancellazione dei tipi parametrici

- La macchina virtuale elimina i tipi parametrici, sostituendoli con i loro vincoli o con `Object`.
- Non si possono costruire oggetti o array di un tipo generico.

## Esercizi di riepilogo e approfondimento

- ★ **R15.1.** Cos'è un tipo parametrico?

- \* **R15.2.** Qual è la differenza tra una classe generica e una classe ordinaria?
- \* **R15.3.** Qual è la differenza tra una classe generica e un metodo generico?
- \* **R15.4.** Nella libreria standard di Java, identificate un esempio di metodo generico non statico.
- \*\* **R15.5.** Nella libreria standard di Java, identificate quattro esempi di classi generiche che abbiano due tipi parametrici.
- \*\* **R15.6.** Nella libreria standard di Java, identificate un esempio di classe generica che non sia una delle classi che realizzano contenitori.
- \* **R15.7.** Perché nel metodo seguente serve un vincolo per il tipo parametrico, T?
 

```
<T extends Comparable> int binarySearch(T[] a, T key)
```
- \*\* **R15.8.** Perché nella classe `HashSet<E>` non serve un vincolo per il tipo parametrico, E?
- \* **R15.9.** Che cosa rappresenta un esemplare di `ArrayList<Pair<T, T>>`?
- \*\* **R15.10.** Illustrate i vincoli applicati ai tipi nel seguente metodo della classe `collections`:
 

```
public static <T extends Comparable<? super T>> void sort(List<T> a)
```

 Perché non è sufficiente scrivere `<T extends Comparable>` oppure `<T extends Comparable<T>>`?

- \*\* **R15.11.** Che risultato si ottiene con la seguente verifica di condizione?

```
ArrayList<BankAccount> accounts = new ArrayList<BankAccount>();
if (accounts instanceof ArrayList<String>) ...
```

Provate e fornite una spiegazione.

- \*\*\* **R15.12.** La classe `ArrayList<E>` della libreria standard di Java deve gestire un array di oggetti di tipo E, ma, in Java, non è lecito costruire un array generico, di tipo E[]. Osservate, nel codice sorgente della libreria, che fa parte del JDK, la soluzione che è stata adottata e fornite una spiegazione.

## Esercizi di programmazione

- \* **E15.1.** Modificate la classe generica `Pair` in modo che i due valori siano dello stesso tipo.
- \* **E15.2.** Aggiungete alla classe `Pair` dell'esercizio precedente un metodo `swap` che scambi tra loro il primo e il secondo elemento della coppia.
- \*\* **E15.3.** Realizzate un metodo statico generico `PairUtil.swap`, il cui parametro sia un oggetto di tipo `Pair`, usando la classe generica definita nel Paragrafo 15.2. Il metodo deve restituire una nuova coppia avente il primo ed il secondo elemento scambiati rispetto alla coppia originaria.
- \*\* **E15.4.** Scrivete un metodo statico generico `PairUtil.minmax` che identifichi gli elementi minimo e massimo presenti in un array di tipo T e restituisca una coppia contenente tali valori minimo e massimo. Esprimete il fatto che gli elementi dell'array devono implementare l'interfaccia `Measurable` vista nel Capitolo 10.
- \*\* **E15.5.** Risolvete nuovamente il problema dell'esercizio precedente, richiedendo però che gli elementi dell'array implementino l'interfaccia `Comparable`.

- \*\*\* **E15.6.** Risolvete nuovamente il problema dell'Esercizio E15.4, richiedendo però che il tipo parametrico estenda il tipo generico `Comparable`.
- \*\* **E15.7.** Realizzate una versione generica dell'algoritmo di ricerca binaria.
- \*\*\* **E15.8.** Realizzate una versione generica dell'algoritmo di ordinamento per fusione. Il programma deve essere compilato senza che vengano segnalati *warning*.
- \*\*\* **E15.9.** Dotate di un appropriato metodo `hashCode` la classe `Pair` vista nel Paragrafo 15.2 e realizzate una classe `HashMap` che usi un esemplare di `HashSet<Pair<K, V>>`.
- \*\*\* **E15.10.** Realizzate una versione generica del generatore di permutazioni visto nel Paragrafo 12.4, in modo che generi tutte le permutazioni degli elementi presenti in un esemplare di `List<E>`.
- \*\* **E15.11.** Scrivete un metodo statico generico, `print`, che visualizzi l'elenco degli elementi presenti in qualsiasi esemplare di una classe che implementi l'interfaccia `Iterable<E>`, inserendolo in un'appropriata classe di utilità. Gli elementi visualizzati devono essere separati da una virgola.

## Risposte alle domande di auto-valutazione

---

1. `HashMap<String, Integer>`
2. `new Pair<String, String>("Hello", "World")`
3. Il primo contiene coppie, ad esempio `[(Tom, 1), (Harry, 3)]`, mentre il secondo contiene un elenco di stringhe e un unico numero intero, ad esempio `[(Tom, Harry], 1)`.
4. Ciò che viene visualizzato dipende dalla definizione del metodo `toString` nella classe `BankAccount`.
5. No, il metodo non ha tipi parametrici, si tratta di un metodo ordinario all'interno di una classe generica.
6. `public class TreeSet <E extends Comparable>`
7. 

```
public static <E extends Measurable> E min(E[] a)
{
    E smallest = a[0];
    for (int i = 1; i < a.length; i++)
    {
        if (a[i].getMeasure() < smallest.getMeasure())
            smallest = a[i];
    }
    return smallest;
}
```
8. 

```
public static void print(Object[] a)
{
    for (Object e : a)
    {
        System.out.print(e + " ");
    }
    System.out.println();
}
```
9. La classe supera la compilazione (con un *warning*), ma si tratta di una tecnica debole e se, in futuro, non si effettuerà più la cancellazione dei tipi parametrici, questo codice sarà *errato*: il cast da `Object[]` a `String[]` provocherà il lancio di un'eccezione.



# A

## Il sottoinsieme Basic Latin di Unicode



**Tabella 1**

Il sottoinsieme Basic Latin  
(ASCII) di Unicode

Carattere	Codice	Decimale	Carattere	Codice	Decimale
!	'\u0021'	33	6	'\u0036'	54
"	'\u0022'	34	7	'\u0037'	55
#	'\u0023'	35	8	'\u0038'	56
\$	'\u0024'	36	9	'\u0039'	57
%	'\u0025'	37	:	'\u003A'	58
&	'\u0026'	38	;	'\u003B'	59
*	'\u0027'	39	<	'\u003C'	60
(	'\u0028'	40	=	'\u003D'	61
)	'\u0029'	41	>	'\u003E'	62
*	'\u002A'	42	?	'\u003F'	63
+	'\u002B'	43	@	'\u0040'	64
,	'\u002C'	44	A	'\u0041'	65
-	'\u002D'	45	B	'\u0042'	66
.	'\u002E'	46	C	'\u0043'	67
/	'\u002F'	47	D	'\u0044'	68
0	'\u0030'	48	E	'\u0045'	69
1	'\u0031'	49	F	'\u0046'	70
2	'\u0032'	50	G	'\u0047'	71
3	'\u0033'	51	H	'\u0048'	72
4	'\u0034'	52	I	'\u0049'	73
5	'\u0035'	53	J	'\u004A'	74

*(continua)*

(seguito)

Carattere	Codice	Decimale	Carattere	Codice	Decimale
K	'\u004B'	75	e	'\u0065'	101
L	'\u004C'	76	f	'\u0066'	102
M	'\u004D'	77	g	'\u0067'	103
N	'\u004E'	78	h	'\u0068'	104
O	'\u004F'	79	i	'\u0069'	105
P	'\u0050'	80	j	'\u006A'	106
Q	'\u0051'	81	k	'\u006B'	107
R	'\u0052'	82	l	'\u006C'	108
S	'\u0053'	83	m	'\u006D'	109
T	'\u0054'	84	n	'\u006E'	110
U	'\u0055'	85	o	'\u006F'	111
V	'\u0056'	86	p	'\u0070'	112
W	'\u0057'	87	q	'\u0071'	113
X	'\u0058'	88	r	'\u0072'	114
Y	'\u0059'	89	s	'\u0073'	115
Z	'\u005A'	90	t	'\u0074'	116
[	'\u005B'	91	u	'\u0075'	117
\	'\u005C'	92	v	'\u0076'	118
]	'\u005D'	93	w	'\u0077'	119
^	'\u005E'	94	x	'\u0078'	120
-	'\u005F'	95	y	'\u0079'	121
~	'\u0060'	96	z	'\u007A'	122
a	'\u0061'	97	{	'\u007B'	123
b	'\u0062'	98		'\u007C'	124
c	'\u0063'	99	}	'\u007D'	125
d	'\u0064'	100	~	'\u007E'	126

**Tabella 2**  
Alcuni caratteri di controllo

Carattere	Sequenza di escape	Decimale	Codice
Tab	'\t'	9	'\u0009'
Nuova riga	'\n'	10	'\u000A'
Invio	'\r'	13	'\u000D'
Spazio		32	'\u0020'

# B

## Linguaggio Java: operatori



Gli operatori sono elencati, nella tabella che segue, in gruppi di precedenza decrescente. Ad esempio,  $z = x - y$ ; significa  $z = (x - y)$ ; perché  $=$  ha una precedenza più bassa di  $-$ .

Operatore	Descrizione	Associatività
.	Accesso alle caratteristiche di una classe	da sinistra a destra
[]	Indice di array	da sinistra a destra
()	Invocazione di metodo	da sinistra a destra
++	Incremento	da destra a sinistra
--	Decremento	da destra a sinistra
!	Not booleano	da destra a sinistra
~	Not bit a bit	da destra a sinistra
+ (unario)	(nessun effetto)	da destra a sinistra
- (unario)	Inversione di segno	da destra a sinistra
(NomeTipo)	Cast	da destra a sinistra
new	Creazione di oggetto	da destra a sinistra
*	Moltiplicazione	da sinistra a destra
/	Divisione o divisione intera	da sinistra a destra
%	Resto della divisione intera	da sinistra a destra

(continua)

(seguito)

<b>Operatore</b>	<b>Descrizione</b>	<b>Associatività</b>
+	Addizione o concatenazione di stringhe	da sinistra a destra
-	Sottrazione	da sinistra a destra
<<	Scorrimento a sinistra	da sinistra a destra
>>	Scorrimento a destra con estensione del segno	da sinistra a destra
>>>	Scorrimento a destra con estensione di zeri	da sinistra a destra
<	Minore di	da sinistra a destra
<=	Minore di o uguale a	da sinistra a destra
>	Maggiore di	da sinistra a destra
>=	Maggiore di o uguale a	da sinistra a destra
instanceof	Verifica se un oggetto è di un certo tipo o di un suo sottotipo	da sinistra a destra
==	Uguale	da sinistra a destra
!=	Diverso	da sinistra a destra
&	And bit a bit	da sinistra a destra
^	Or esclusivo bit a bit	da sinistra a destra
	Or bit a bit	da sinistra a destra
&&	And booleano con "cortocircuito"	da sinistra a destra
	Or booleano con "cortocircuito"	da sinistra a destra
?:	Condizionale	da destra a sinistra
=	Assegnamento	da destra a sinistra
operatore=	Assegnamento con operatore binario (operatore può essere +, -, *, /, &,  , ^, <<, >>, >>>)	da destra a sinistra

# C

## Linguaggio Java: parole riservate



Parola riservata	Descrizione
abstract	Una classe o un metodo astratti
assert	Un'asserzione di una condizione che deve essere verificata
boolean	Il tipo booleano
break	Interrompe l'attuale ciclo o enunciato con etichetta
byte	Il tipo intero a 8 bit
case	Un'etichetta in un enunciato switch
catch	Il gestore di un'eccezione in un blocco try
char	Il tipo carattere Unicode a 16 bit
class	Definisce una classe
const	Non usata
continue	Ignora la parte restante del corpo di un ciclo
default	L'etichetta predefinita in un enunciato switch o un metodo predefinito
do	Un ciclo il cui corpo viene eseguito almeno una volta
double	Il tipo in virgola mobile con doppia precisione a 64 bit
else	La clausola alternativa in un enunciato if
enum	Un tipo enumerativo
extends	Indica che una classe è una sottoclassificazione di un'altra
final	Un valore che non può essere modificato dopo essere stato inizializzato, un metodo che non può essere ridefinito oppure una classe che non può essere estesa
finally	Una clausola di un blocco try che viene sempre eseguita

(continua)

(seguito)

Parola riservata	Descrizione
float	Il tipo in virgola mobile con singola precisione a 32 bit
for	Un ciclo con inizializzazione, condizione e espressioni di aggiornamento
goto	Non usata
if	Un enunciato per una diramazione condizionata
implements	Indica che una classe realizza un'interfaccia
import	Consente l'uso di nomi di classe senza il nome del pacchetto
instanceof	Verifica se il tipo di un oggetto è uguale al tipo specificato o a una sua sottoclasse
int	Il tipo intero a 32 bit
interface	Un tipo astratto con soli metodi astratti o predefiniti e costanti
long	Il tipo intero a 64 bit
native	Un metodo realizzato in un linguaggio diverso da Java
new	Crea un nuovo oggetto
package	Una raccolta di classi correlate
private	Una caratteristica che è accessibile soltanto da metodi della stessa classe
protected	Una caratteristica che è accessibile soltanto da metodi della stessa classe, di una sottoclasse o di un'altra classe nello stesso pacchetto
public	Una caratteristica che è accessibile da qualsiasi metodo
return	Termina un metodo
short	Il tipo intero a 16 bit
static	Una caratteristica definita per una classe, invece che per singoli esemplari
strictfp	Usa regole stringenti per i calcoli in virgola mobile
super	Invoca il costruttore della superclasse o un suo metodo
switch	Un enunciato di selezione
synchronized	Un blocco di codice che è accessibile da un solo thread per volta
this	Il parametro implicito di un metodo oppure l'invocazione di un altro costruttore della classe
throw	Lancia un'eccezione
throws	Le eccezioni che possono essere lanciate da un metodo
transient	Variabili di esemplare che non dovrebbero essere serializzate
try	Un blocco di codice con gestori di eccezione o con un gestore finally
void	Contrassegna un metodo che non restituisce alcun valore
volatile	Una variabile che potrebbe essere aggiornata da più thread senza sincronizzazione
while	Un enunciato di ciclo

# D

## Sistemi di numerazione



### Numeri binari

La notazione decimale rappresenta i numeri come potenze di 10, ad esempio

$$1729_{\text{decimale}} = 1 \times 10^3 + 7 \times 10^2 + 2 \times 10^1 + 9 \times 10^0$$

Non c'è nessun motivo particolare per la scelta del numero 10, a parte il fatto che molti sistemi di numerazione sono stati messi a punto da persone che contavano con le dita: altri sistemi numerici (con base 12, 20 o 60) sono stati usati da varie culture nel corso della storia dell'umanità. I computer, invece, usano un sistema numerico con base 2 perché è molto più facile costruire componenti elettronici che funzionino con due soli valori, che possono essere rappresentati da una corrente che scorre oppure no, piuttosto che rappresentare 10 valori diversi di un segnale elettrico. Un numero scritto in base 2 viene anche detto numero *binario*. Ad esempio

$$1101_{\text{binario}} = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8 + 4 + 1 = 13$$

Per le cifre che seguono il punto “decimale”, si usano le potenze negative di 2.

$$1.101_{\text{binario}} = 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 1 + 1/2 + 1/8 = 1 + 0.5 + 0.125 = 1.625$$

In generale, per convertire un numero binario nel suo equivalente decimale, si valutano semplicemente le potenze di 2 che corrispondono alle cifre di valore 1 e si sommano. La Tabella 1 mostra le prime potenze di 2.

**Tabella 1**

Potenze di due

	$2^0$	1
	$2^1$	2
	$2^2$	4
	$2^3$	8
	$2^4$	16
	$2^5$	32
	$2^6$	64
	$2^7$	128
	$2^8$	256
	$2^9$	512
	$2^{10}$	1024
	$2^{11}$	2048
	$2^{12}$	4096
	$2^{13}$	9192
	$2^{14}$	16384
	$2^{15}$	32768
	$2^{16}$	65536

Per convertire in binario un numero intero decimale, lo si divide ripetutamente per 2, tenendo traccia dei resti e fermandosi quando il dividendo diventa 0. Poi, si scrivono i resti come numero binario, iniziando dall'*ultimo*. Ad esempio:

$$\begin{array}{r|l} 100 & 2 = 50 \text{ resto } 0 \\ 50 & 2 = 25 \text{ resto } 0 \\ 25 & 2 = 12 \text{ resto } 1 \\ 12 & 2 = 6 \text{ resto } 0 \\ 6 & 2 = 3 \text{ resto } 0 \\ 3 & 2 = 1 \text{ resto } 1 \\ 1 & 2 = 0 \text{ resto } 1 \end{array}$$

Quindi,  $100_{\text{decimale}} = 1100100_{\text{binario}}$ .

Per convertire in binario, invece, un numero frazionario minore di 1, lo si moltiplica ripetutamente per 2: se il risultato è maggiore di 1, si sottrae 1; se il numero da moltiplicare diventa 0, la conversione è terminata. Poi, si scrivono le cifre che precedono il punto decimale come cifre binarie della parte frazionaria, iniziando dalla *prima*. Ad esempio:

$$\begin{aligned} 0.35 \cdot 2 &= 0.7 \\ 0.7 \cdot 2 &= 1.4 \\ 0.4 \cdot 2 &= 0.8 \\ 0.8 \cdot 2 &= 1.6 \\ 0.6 \cdot 2 &= 1.2 \\ 0.2 \cdot 2 &= 0.4 \end{aligned}$$

A questo punto lo schema si ripete, quindi la rappresentazione binaria di 0.35 è 0.01011001100110 ...

Per convertire in binario un numero decimale generico, convertite la parte intera e la parte frazionaria separatamente.

## Errori di arrotondamento e overflow

In Java, un valore di tipo int è un numero intero che occupa 32 bit. Eseguendo operazioni tra due valori di questo tipo, è possibile che il risultato non trovi posto in 32 bit: una situazione che prende il nome di *overflow* (“trabocco”). In tal caso vengono usati solamente gli ultimi 32 bit del risultato, dando luogo a un valore errato. Ad esempio, il frammento di codice seguente visualizza 705032704:

```
int fiftyMillion = 50000000;
System.out.println(100 * fiftyMillion); // Previsto: 5000000000
```

Per capire che cosa giustifica un risultato così curioso, si può eseguire la lunga moltiplicazione a mano:

```
1 1 0 0 1 0 0 * 1 0 1 1 1 1 0 1 0 1 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0
1 0 1 1 1 1 0 1 0 1 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0
0
0
1 0 1 1 1 1 0 1 0 1 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0
0
0

1 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0
```

Il risultato ha 33 bit, quindi non trova posto nei 32 bit di un valore di tipo int e il primo bit viene ignorato: gli altri 32 bit sono la rappresentazione binaria di 705032704 (infatti il bit ignorato vale  $2^{32} = 4294967296$  e la somma dei due valori è proprio 5000000000, il risultato corretto).

Elaborando numeri in virgola mobile si può verificare un altro tipo di errore: l'errore di arrotondamento (*roundoff*). Consideriamo questo esempio:

```
double price = 4.35;
double quantity = 100;
double total = price * quantity; // dovrebbe essere 100 * 4.35 = 435
System.out.println(total); // visualizza 434.9999999999999
```

Per capire come si possa verificare questo errore, eseguiamo anche in questo caso la moltiplicazione in colonna:

```
1 1 0 0 1 0 0 * 1 0 0 0 1 | 0 1 1 0 | 0 1 1 0 | 0 1 1 0 . . .
```

```
1 0 0 0 1 | 0 1 1 0 | 0 1 1 0 | 0 1 1 0 . . .
1 0 0 0 1 | 0 1 1 0 | 0 1 1 0 | 0 1 1 0 . . .
0
0
1 0 0 0 1 | 0 1 1 0 | 0 1 1 0 | 0 1 1 0 . . .
0
0
```

Quindi, il risultato è 434 seguito da un numero infinito di cifre 1: la sua parte frazionaria è l'equivalente binario del numero decimale periodico 0.999999..., che è matematicamente uguale a 1. Ma il computer può memorizzare soltanto un numero finito di cifre e quando converte il risultato in un numero decimale ne ignora alcune.

## Numeri interi in complemento a due

Per rappresentare numeri interi negativi esistono due comuni notazioni, chiamate "modulo e segno" e "complemento a due". La notazione con modulo e segno è semplice: si usa il bit più a sinistra per il segno (0 = positivo, 1 = negativo). Ad esempio, usando numeri di 8 bit:

$$-13 = 10001101 \text{ modulo e segno}$$

Tuttavia, costruire circuiti per sommare numeri diventa un po' più complicato quando occorre considerare il segno. La rappresentazione in complemento a due risolve questo problema. Per comporre il complemento a due di un numero:

- Cambiate il valore di tutti i bit.
- Quindi, aggiungete 1.

Ad esempio, per calcolare  $-13$  come valore di 8 bit, dapprima cambiate il valore di tutti i bit di 00001101 (che è la rappresentazione di 13), ottenendo 11110010, quindi aggiungete 1:

$$-13 = 11110011 \text{ complemento a due}$$

Con questa notazione non serve nessun circuito specifico per sommare due numeri: seguite semplicemente le normali regole dell'addizione, con il riporto nella posizione successiva nel caso in cui la somma delle cifre e del riporto precedente sia 2 o 3. Ad esempio:

$$\begin{array}{r} 1 \ 1111 \ 111 \\ +13 \quad 0000 \ 1101 \\ -13 \quad 1111 \ 0011 \\ \hline 1 \ 0000 \ 0000 \end{array}$$

Sono importanti, però, soltanto gli ultimi 8 bit, per cui  $+13$  e  $-13$  hanno somma 0, come dovrebbero.

In particolare, la rappresentazione in complemento a due di  $-1$  è 1111...1111, cioè tutti i bit valgono 1.

Il bit più a sinistra di un numero in complemento a due vale 0 se il numero è positivo e 1 se è negativo.

La rappresentazione in complemento a due con un determinato numero di bit può rappresentare un numero negativo in più rispetto al numero di valori positivi rappresentabili; ad esempio, i numeri in complemento a due con 8 bit variano da  $-128$  a  $+127$ .

Questo fenomeno è fonte di errori di programmazione. Ad esempio, considerate il codice seguente:

```
short b = . . .;
if (b < 0) { b = (byte) -b; }
```

Questo codice non garantisce che, al termine, *b* sia non negativo. Se *b* vale inizialmente -128, il calcolo del suo opposto fornisce nuovamente il valore -128. Provate: prendete 10000000, cambiate tutti i bit, e sommate 1.

## Standard IEEE per numeri in virgola mobile

Lo standard IEEE-754 (IEEE, Institute for Electrical and Electronics Engineering) definisce le rappresentazioni per i numeri in virgola mobile. La Figura 1 mostra come i valori in singola precisione (*float*) e in doppia precisione (*double*) siano composti da:

- un bit di segno
- un esponente
- una mantissa

I numeri in virgola mobile usano la notazione scientifica, nella quale un numero viene rappresentato come

$$b_0.b_1b_2b_3\dots \times 2^e$$

In questa rappresentazione, *e* è l'esponente, mentre le cifre  $b_0.b_1b_2b_3\dots$  compongono la mantissa. La rappresentazione *normalizzata* è quella avente  $b_0 \neq 0$ . Per esempio:

$$100_{\text{decimale}} = 1100100_{\text{binario}} = 1.100100_{\text{binario}} \times 2^6$$

Poiché il primo bit di una rappresentazione normalizzata deve necessariamente essere 1, questo in realtà nel sistema di numerazione binario non viene memorizzato nella mantissa, per cui dovete sempre aggiungerlo per ottenere il valore vero. Ad esempio, la mantissa 1.100100 viene memorizzata come 100100.

La parte della rappresentazione IEEE riservata all'esponente non usa né la rappresentazione in complemento a due né quella in modulo e segno: viene aggiunto all'esponente vero una quantità fissa, detta *bias*. Tale quantità è 127 per i numeri in singola precisione e 1023 per quelli in doppia precisione. Ad esempio, l'esponente *e* = 6 verrebbe memorizzato come 133 in un numero in singola precisione.

Quindi:

$$100_{\text{decimale}} = 0|10000101|100100000000000000000000_{\text{IEEE singola precisione}}$$

Ci sono, poi, alcuni valori speciali. Fra questi:

- Zero*: esponente con bias = 0, mantissa = 0.
- Infinito*: esponente con bias = 11...1, mantissa =  $\pm 0$ .
- NaN* (*not a number*, non è un numero valido): esponente con bias = 11...1, mantissa  $\neq \pm 0$ .

**Figura 1**  
Rappresentazione IEEE  
per numeri  
in virgola mobile

	1 bit	8 bit	23 bit
Segno	Esponente con bias $e + 127$	Mantissa (senza 1 iniziale)	
Singola precisione			
Segno	11 bit	Esponente con bias $e + 1023$	52 bit
Doppia precisione			

## Numeri esadecimales

Poiché i numeri binari sono di difficile lettura per le persone, spesso i programmati usano il sistema di numerazione esadecimale, con base 16. Le cifre vengono indicate con 0, 1, ..., 9, A, B, C, D, E, F (osservate la Tabella 2).

Quattro cifre binarie corrispondono a una cifra esadecimale: ciò rende semplici le conversioni fra valori binari e valori esadecimali. Ad esempio

$$11|1011|0001_{\text{binario}} = 3B1_{\text{esadecimale}}$$

In Java, i numeri esadecimali sono usati come valori per i caratteri Unicode, ad esempio \u03b1 (la lettera greca alfa minuscola). I numeri interi esadecimali vengono indicati con il prefisso 0x, come, ad esempio, 0x3B1.

## Operazioni con bit e scorrimenti

In Java sono disponibili quattro operazioni che agiscono su bit: la negazione unaria ( $\sim$ ) e le operazioni binarie *and* (`&`), *or* (`|`) e *or esclusivo* (`^`), detto anche *xor*.

**Tabella 2**  
Cifre esadecimale

	Esadecimale	Decimale	Binario
0	0	0	0000
1	1	1	0001
2	2	2	0010
3	3	3	0011
4	4	4	0100
5	5	5	0101
6	6	6	0110
7	7	7	0111
8	8	8	1000
9	9	9	1001
A	10	10	1010
B	11	11	1011
C	12	12	1100
D	13	13	1101
E	14	14	1110
F	15	15	1111

Le Tabelle 3 e 4 mostrano le tabelle di verità per le operazioni sui bit in Java. Quando un'operazione sui bit viene applicata a numeri interi, l'operazione viene eseguita sui bit corrispondenti.

**Tabella 1**  
L'operazione  
di negazione unaria

a	$\sim a$
0	1
1	0

**Tabella 2**  
Le operazioni binarie and,  
or e xor

a	b	a&b	a b	a^b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Ad esempio, supponete di voler calcolare  $46 \& 13$ . Per prima cosa convertite in binario entrambi i valori:  $46_{\text{decimale}} = 101110_{\text{binario}}$  (in realtà, essendo un numero intero a 32 bit, 00000000000000000000000000000000101110);  $13_{\text{decimale}} = 1101_{\text{binario}}$ . Poi, effettuate l'operazione sui bit corrispondenti:

0.....0101110  
& 0.....0001101

0.....0001100

La risposta è  $1100_{\text{binario}} = 12_{\text{decimale}}$ .

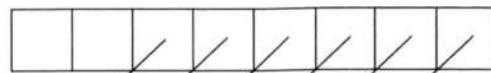
A volte si vede l'operatore | usato per combinare due schemi di bit. Ad esempio, **Font.BOLD** ha il valore 1, **Font.ITALIC** ha il valore 2. La combinazione **Font.BOLD | Font.ITALIC** ha impostati a 1 sia il bit per il grassetto sia quello per il corsivo:

0.....0000001  
| 0.....0000010  
0.....0000011

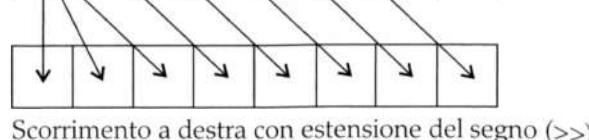
Non confondete gli operatori per i bit & e | con gli operatori && e ||. Questi ultimi operano soltanto su valori di tipo **boolean**, e non sui bit.

Oltre alle operazioni che agiscono su singoli bit, esistono anche tre operazioni di *scorrimento (shift)* che prendono lo schema di bit di un numero e lo spostano a sinistra o a destra di un certo numero di posizioni. Esistono tre operazioni di scorrimento: scorrimento a sinistra (<<), scorrimento a destra con estensione del segno (>>) e scorrimento a destra con estensione di zeri (>>>).

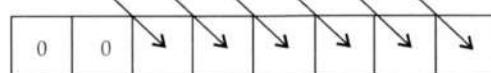
Lo scorrimento a sinistra sposta tutti i bit verso sinistra, inserendo zeri nei bit meno significativi. Lo spostamento a sinistra di  $n$  bit fornisce lo stesso risultato di una moltiplicazione per  $2^n$ . Lo scorrimento verso destra con estensione del segno sposta tutti i bit a destra, *propagando* il bit di segno: quindi, il risultato è uguale a quello della divisione intera per  $2^n$ , sia per valori positivi che per valori negativi. Infine, lo scorrimento a destra

**Figura 2**Le operazioni  
di scorrimento

Scorrimento a sinistra (&lt;&lt;)



Scorrimento a destra con estensione del segno (&gt;&gt;)



Scorrimento a destra con estensione di zeri (&gt;&gt;&gt;)

con estensione di zeri sposta tutti i bit a destra, inserendo zeri nei bit più significativi (osservate la Figura 2).

Notate che il valore a destra dell'operatore di scorrimento viene usato modulo 32 (per valori di tipo `int`) o modulo 64 (per valori di tipo `long`), determinando così l'effettivo numero di posizioni di cui spostare i bit.

Ad esempio, `1 << 35` è uguale a `1 << 3`. Spostare veramente il numero 1 verso sinistra di 35 posizioni non avrebbe senso: il risultato sarebbe 0.

L'espressione

$$1 \ll n$$

genera uno schema di bit in cui il solo bit  $n$ -esimo vale 1 (contando le posizioni a partire dalla posizione 0 del bit meno significativo).

Per impostare a 1 il bit  $n$ -esimo di un numero, eseguite l'operazione seguente:

$$x = x | 1 \ll n$$

Per controllare se il bit  $n$ -esimo di un numero vale 1, eseguite la verifica seguente:

$$\text{if } ((x \& 1 \ll n) != 0) \dots$$

Notate che le parentesi attorno all'operatore `&` sono necessarie, perché ha una precedenza minore degli operatori relazionali.

# Glossario



**Accesso casuale** La possibilità di accedere direttamente a qualsiasi valore senza dover leggere i valori che lo precedono.

**Accesso sequenziale** Accesso a valori in sequenza, senza saltarne alcuno.

**Accoppiamento** Il grado di mutua correlazione tra classi per effetto di dipendenze.

**Adattatore per eventi** Una classe che realizza un'interfaccia per ricevitore di eventi definendo tutti i suoi metodi in modo che non compiano alcuna azione.

**ADT (Abstract Data Type, tipo di dato astratto)** Una specifica delle operazioni fondamentali che caratterizzano un tipo di dati, senza fornirne una realizzazione.

**Algoritmo** Una specifica del modo di risolvere un problema che sia non ambigua, eseguibile e che termini in un tempo finito.

**Ambito di visibilità** La porzione di programma in cui è definita e visibile una variabile.

**Apertura di un file** Predisposizione di un file per operazioni di lettura o di scrittura.

**API (Application Programming Interface)** Una libreria di codice utilizzata per costruire programmi.

**Applet** Un programma grafico scritto in Java che viene

eseguito all'interno di un browser web o di un apposito visualizzatore di applet.

**Argomento** Un parametro effettivo fornito nell'invocazione di un metodo oppure uno dei valori (*operandi*) su cui agisce un operatore.

**Array** Una raccolta di valori del medesimo tipo, disposti in posizioni di memoria contigue, a ciascuno dei quali si può accedere mediante un indice intero.

**Array bidimensionale** Una disposizione tabulare di elementi in cui si accede a ciascun elemento mediante un indice di riga e un indice di colonna.

**Array paralleli** Array aventi la stessa lunghezza, in cui elementi corrispondenti sono tra loro correlati dal punto di vista logico.

**Array riempito solo in parte** Un array che non è riempito per la sua intera capacità, accompagnato da una variabile che indica il numero di elementi che vi sono realmente memorizzati.

**Assegnazione** Memorizzazione di un nuovo valore in una variabile.

**Asserzione** L'affermazione che una certa condizione sia vera in un particolare punto di un programma.

**Associatività degli operatori** La regola che stabilisce in

quale ordine debbano essere eseguiti operatori aventi la medesima precedenza. Ad esempio, in Java l'operatore `-` è associativo a sinistra, per cui `a - b - c` viene interpretato come `(a - b) - c`, mentre l'operatore `=` è associativo a destra, per cui `a = b = c` viene interpretato come `a = (b = c)`.

**Auto-boxing** Conversione automatica di un valore di tipo primitivo in un esemplare di una classe involucro (*wrapper*).

**Bit** Cifra binaria; la più piccola unità di informazione, con due possibili valori: 0 e 1. Un dato composto da  $n$  bit ha  $2^n$  possibili valori.

**Blocco** Un gruppo di enunciati racchiusi tra parentesi graffe.

**Breakpoint** Una posizione nella quale si desidera l'arresto del debugger, per ispezionare lo stato del programma in esecuzione controllata.

**Buffer** Una zona di memoria usata temporaneamente per memorizzare valori (ad esempio, caratteri digitati dall'utente) che sono in attesa di essere utilizzati (ad esempio, leggendo una riga per volta).

**Bug** Un errore di programmazione.

**Byte** Una quantità di informazione pari a otto bit. Tutti gli attuali produttori di calcolatori usano il byte come unità elementare di memorizzazione.

**Bytecode** Istruzioni per la macchina virtuale Java.

**Callback** Un meccanismo che consente di specificare un blocco di codice da eseguire successivamente.

**Carattere** Una singola lettera, una cifra o un simbolo.

**Carattere di escape** Un carattere che non va interpretato in modo letterale, ma ha un significato speciale quando viene combinato con il carattere (o i caratteri) seguente. Il carattere `\` è un carattere di escape nelle stringhe Java.

**Carattere di nuova riga (newline)** Il carattere '`\n`', che segnala la fine di una riga.

**Carattere di tabulazione** Il carattere '`\t`', che sposta il successivo carattere lungo la riga in modo che si allinei alla successiva posizione prefissata, denominata "posizione di tabulazione".

**Caratteri di spaziatura (white space)** L'insieme dei caratteri di spazio, tabulazione e nuova riga.

**Cartella (directory)** Una struttura di un *file system* che è in grado di contenere file o altre cartelle.

**Caso limite** Una situazione da collaudare che coinvolge valori che si trovano al limite del proprio insieme di validità. Ad esempio, il valore zero è un caso limite per collaudare una funzione che elabora valori non negativi.

**Cast** Conversione esplicita ("forzata") di un valore da un tipo a un tipo diverso. Ad esempio, se `x` è una variabile che contiene un numero in virgola mobile, la notazione

`(int) x` esprime, in Java, la conversione forzata in numero intero del valore contenuto in `x`.

**Ciclo** Una sequenza di istruzioni che viene eseguita ripetutamente.

**Ciclo annidato** Un ciclo contenuto all'interno di un altro ciclo.

**Ciclo e mezzo** Un ciclo la cui decisione di terminazione non si trova né all'inizio né alla fine.

**Classe** Un tipo di dato definito dal programmatore.

**Classe anonima** Una classe priva di nome.

**Classe astratta** Una classe di cui non si possono creare esemplari.

**Classe concreta** Una classe di cui si possono creare esemplari.

**Classe generica** Una classe con tipi parametrici.

**Classe immutabile** Una classe priva di metodi modificatori.

**Classe interna** Una classe definita all'interno di un'altra classe.

**Classe involucro (wrapper)** Una classe, come `Integer`, che contiene un valore di un tipo primitivo.

**Classe per eventi** Una classe che contiene informazioni in merito a un evento dell'interfaccia grafica; ad esempio, la sorgente dell'evento stesso.

**Clausola catch** Parte di un blocco `try` che viene eseguita quando un qualsiasi enunciato interno al blocco `try` lancia un'eccezione catturata dalla clausola.

**Clausola finally** Parte di un blocco `try` che viene eseguita indipendentemente dal modo in cui termina il blocco `try` stesso.

**Clausola throws** Specifica il tipo di eccezioni a controllo obbligatorio che possono essere lanciate da un metodo.

**Coda** Una raccolta di elementi che vengono estratti con la strategia "il primo entrato è il primo a uscire".

**Coda prioritaria** Una raccolta che consente di eseguire in modo efficiente l'inserimento di elementi e la rimozione dell'elemento di valore minimo.

**Codice di hash** Un valore calcolato da una funzione di hash.

**Codice macchina** Istruzioni che possono essere eseguite direttamente dalla CPU.

**Codice sorgente** Istruzioni, espresse in un linguaggio di programmazione, che necessitano di traduzione prima di poter essere eseguite da un calcolatore.

**Coesione** Una classe è coesa se le sue caratteristiche descrivono un'unica astrazione.

**Collaudo "a scatola bianca"** Un collaudo progettato, al contrario del collaudo "a scatola nera", prendendo in esame l'implementazione della classe da collaudare,

- ad esempio selezionando accuratamente i casi limite e garantendo la copertura di tutte le diramazioni del codice.
- Collaudo “a scatola nera”** Un collaudo progettato senza conoscere l’implementazione del metodo da collaudare.
- Collaudo di unità** Il collaudo di un metodo a sé stante, isolato dal resto del programma.
- Collaudo regressivo** Raccolta di tutti i casi di prova per utilizzarli nel collaudo di tutte le successive revisioni di un programma.
- Collisione** Si ha quando una funzione di hash calcola il medesimo codice per due oggetti diversi.
- Commento** Una spiegazione che aiuta un lettore umano a comprendere una porzione di programma; viene ignorata dal compilatore.
- Commento per la documentazione** Un commento all’interno di un file sorgente che può essere estratto automaticamente da un programma come javadoc per generare la documentazione del programma.
- Compilatore** Un programma che traduce codice scritto in un linguaggio di alto livello (come Java) in istruzioni macchina (come le istruzioni bytecode per la macchina virtuale Java).
- Componente dell’interfaccia utente** Un blocco costitutivo dell’interfaccia grafica di interazione con l’utente, come un pulsante o un campo di testo. I componenti dell’interfaccia utente vengono utilizzati per presentare informazioni all’utente e per consentire a quest’ultimo di fornire informazioni al programma.
- Concatenazione** L’accodamento di una stringa al termine di un’altra stringa, per formare una stringa più lunga.
- Conflitto tra nomi** L’utilizzo accidentale di uno stesso nome per indicare due diverse caratteristiche di un programma in un modo che il compilatore non è in grado di dirimere.
- Contesto grafico** Una classe mediante la quale un programmatore può disegnare forme grafiche all’interno di una finestra o in un file.
- Copertura del codice** La percentuale delle istruzioni di un programma che vengono eseguite durante un collaudo.
- Corpo** L’insieme degli enunciati di un metodo o di un blocco.
- Costante** Un valore che non può essere modificato dal programma. In Java, le costanti vengono definite mediante la parola riservata final.
- Costruttore** Una sequenza di enunciati che inizializza un oggetto appena creato.
- Costruzione** Impostazione dello stato iniziale di un oggetto appena creato.
- CPU (Central Processing Unit)** La parte di calcolatore che esegue le istruzioni in linguaggio macchina.

- CRC (scheda)** Una scheda che rappresenta una classe e ne elenca responsabilità e collaborazioni.
- Debugger** Un programma che consente all’utente l’esecuzione passo dopo passo di un altro programma, con la possibilità di interromperne l’esecuzione e di ispezionarne le variabili, per agevolare l’identificazione degli errori.
- Diagramma sintattico** Una rappresentazione grafica di regole sintattiche.
- Directory** Vedi Cartella
- Disco rigido** Un dispositivo che memorizza informazioni su dischi rotanti ricoperti di materiale magnetico.
- Divisione intera** Fornisce il quoziente della divisione tra due numeri interi, ignorando l’eventuale resto. In Java, quando entrambi gli operandi sono interi, il simbolo / indica la divisione intera: ad esempio, 11/4 vale 2, non 2.75.
- Documentazione API** Informazioni relative alle classi della libreria di Java.
- Eccezione** Una classe che segnala una condizione che impedisce la normale prosecuzione del programma: quando si verifica tale condizione, viene lanciato un esemplare di una classe eccezione.
- Eccezione a controllo non obbligatorio** Un’eccezione la cui cattura non viene resa obbligatoria dal compilatore.
- Eccezione a controllo obbligatorio** Un’eccezione la cui cattura viene resa obbligatoria dal compilatore: deve essere dichiarata o gestita.
- Editor** Un programma utilizzato per scrivere e modificare file di testo.
- Effetto collaterale** Un effetto visibile di un metodo, diverso dal valore restituito.
- Enumerazione** Un tipo di dato che può assumere un numero finito di valori, ciascuno dei quali è dotato di un proprio nome simbolico.
- Enunciato** Un’unità sintattica all’interno di un programma. In Java, un enunciato può essere un enunciato semplice, un enunciato composto o un blocco di enunciati.
- Enunciato break** Enunciato che pone termine a un ciclo o a un enunciato switch.
- Enunciato try** Un enunciato il cui corpo contiene enunciati che vengono eseguiti fino al termine del corpo stesso oppure fino al lancio di un’eccezione; contiene anche clausole che vengono invocate quando si verifica una specifica eccezione.
- Enunciato try con risorse** Un’versione dell’enunciato try la cui intestazione inizializza una variabile con un esemplare di una classe che implementa l’interfaccia AutoCloseable, il cui metodo close viene invocato quando l’enunciato try termina, normalmente o per il lancio di un’eccezione.

- Ereditarietà** La relazione che esiste fra una superclasse, più generica, e una sottoclasse, più specifica.
- Errore di arrotondamento** Un errore dovuto al fatto che il calcolatore può memorizzare soltanto un numero finito di cifre di un numero in virgola mobile.
- Errore di compilazione** Un errore che viene identificato durante la compilazione di un programma.
- Errore di limiti** Tentativo di accesso a un elemento di un array che si trova al di fuori dell'intervallo di indici leciti.
- Errore di sintassi** Un'istruzione che non segue le regole sintattiche del linguaggio di programmazione e viene, quindi, rifiutata dal compilatore (un tipo di errore di compilazione).
- Errore in esecuzione o logico** Un errore che si verifica durante l'esecuzione di un programma sintatticamente corretto, portandolo ad agire in modo diverso da quanto previsto.
- Errore per scarto di uno** Un frequente errore di programmazione che si verifica quando un valore è di un'unità più grande o più piccolo di quanto dovrebbe essere.
- Esemplare di una classe** Un oggetto il cui tipo è una classe.
- Espressione** Un costrutto sintattico composto di costanti, variabili, invocazioni di metodi e operatori che li combinano.
- Espressione canonica** Una stringa che definisce un insieme di stringhe selezionate in base al loro contenuto. Ciascuna parte di un'espressione canonica può essere: uno specifico carattere, che in tal modo diviene essenziale; un carattere appartenente a un insieme di caratteri ammessi, come [abc], che può anche essere un intervallo, come [a-z]; qualsiasi carattere che non appartenga a un insieme di caratteri proibiti, come [^0-9]; la ripetizione di una o più corrispondenze, come [0-9]++; oppure zero o più, come [ACGT]\*; un'opzione scelta in un insieme di alternative, come and|et|und; oppure, varie altre condizioni e possibilità. Ad esempio, all'espressione canonica "[A-Za-z]\*[0-9]++" corrispondono "Cloud9" oppure "007", ma non "Jack".
- Espressione lambda** Un'espressione che definisce con una notazione compatta i parametri e il valore restituito da un metodo.
- Estensione** L'ultima parte del nome di un file, che ne specifica la tipologia. Ad esempio, l'estensione .java identifica i file che contengono codice sorgente Java.
- Evento dell'interfaccia utente** La segnalazione, che viene notificata al programma, di un'azione compiuta dall'utente, come la pressione di un tasto, lo spostamento del mouse o la selezione di una voce di menu.
- File** Una sequenza di byte memorizzata su un disco.

- File di testo** Un file in cui i valori dei dati vengono memorizzati nella loro rappresentazione testuale.
- File sorgente** Un file contenente istruzioni espresse in un linguaggio di programmazione, come Java.
- Finestra di shell** Una finestra che consente di interagire con il sistema operativo mediante comandi di tipo testuale.
- Flag** Vedi Tipo booleano.
- Flusso (stream)** Un'astrazione di una sequenza di byte da cui si possono leggere o in cui si possono scrivere dati.
- Font** Un insieme di forme di caratteri aventi una specifica dimensione e un particolare stile.
- Frame** Una finestra con un bordo e una barra del titolo.
- Funzione** Una funzione matematica restituisce un risultato per ogni possibile assegnazione di valori ai suoi parametri. In Java, le funzioni possono essere implementate mediante espressioni lambda o esemplari di interfacce funzionali.
- Funzione di hash** Una funzione che calcola un numero intero a partire da un oggetto, in modo che sia molto probabile che per oggetti distinti vengano calcolati valori distinti.
- Garbage collection** Recupero automatico della memoria occupata da oggetti ai quali nessuna variabile fa più riferimento.
- Gestore di eccezioni** Una sequenza di enunciati a cui viene demandato il controllo d'esecuzione quando è stata lanciata e catturata un'eccezione di un particolare tipo.
- Gestore di eventi** Un metodo che viene eseguito quando accade un particolare evento.
- grep** Un programma che effettua ricerche mediante espressioni canoniche.
- GUI (Graphical User Interface)** Un'interfaccia mediante la quale l'utente fornisce dati a un programma usando componenti grafici come pulsanti, menu e campi di testo.
- Hardware** La parte fisica di un calcolatore o di un altro dispositivo.
- IDE (Integrated Development Environment)** Un ambiente di programmazione che contiene un editor, un compilatore e un debugger.
- Implementazione di un'interfaccia** Realizzazione di una classe che definisce tutti i metodi specificati da un'interfaccia.
- Importazione di una classe o di un pacchetto** Segnala l'intenzione di fare riferimento a una classe o a tutte le classi contenute in un pacchetto usando semplicemente il suo nome invece del nome "completo".
- Incapsulamento** L'occultamento dei dettagli realizzativi.
- Inizializzazione** Assegnazione di un valore a una variabile nel momento in cui questa viene creata.

**Insieme** Una raccolta non ordinata di elementi che consente di effettuare in modo efficiente l'inserimento, l'eliminazione e la ricerca di elementi.

**Interfaccia** Un tipo di dato privo di variabili di esemplare, contenente soltanto costanti e metodi astratti o predefiniti.

**Interfaccia funzionale** Un'interfaccia che ha un unico metodo astratto e ha lo scopo di definire un'unica funzione.

**Interfaccia pubblica** Le caratteristiche di una classe (metodi, variabili e tipi interni) che sono accessibili agli utilizzatori.

**Istanza** Vedi Esemplare

**Iteratore** Un oggetto che è in grado di ispezionare tutti gli elementi presenti in un contenitore, ad esempio una lista concatenata.

**javadoc** Il generatore di documentazione presente nell'ambiente di sviluppo Java SDK: estrae dai file sorgenti Java i commenti di documentazione e genera un insieme di file HTML collegati come ipertesto.

**Lancio di un'eccezione** Indica una condizione anomala e provoca la terminazione del flusso normale di esecuzione del programma, trasferendo il controllo a una clausola catch appropriata.

**Legge di De Morgan** Una regola riguardante le operazioni logiche: descrive come negare espressioni composte di operatori *and* e *or*.

**Letterale** La notazione che riguarda un valore fisso all'interno di un programma, come -2, 3.14, 6.02214115E23, "Harry" o 'H'.

**Libreria** Un insieme di classi pre-compilate che possono essere utilizzate in un programma.

**Limiti asimmetrici** Limiti di ciclo inclusivi del valore iniziale ma non del valore finale dell'indice.

**Limiti simmetrici** Limiti di ciclo inclusivi del valore iniziale e finale dell'indice.

**Linguaggio di alto livello** Un linguaggio di programmazione che privilegia la rapida progettazione di prototipi piuttosto della velocità di esecuzione o della facilità di manutenzione del codice.

**Lista concatenata** Una struttura di memorizzazione che può contenere un numero arbitrario di oggetti, ciascuno dei quali viene conservato in un nodo che contiene un puntatore al nodo successivo.

**Lista doppiamente concatenata** Una lista concatenata in cui ciascun nodo contiene sia un riferimento al nodo precedente sia un riferimento al nodo successivo.

**Locazione di memoria** Un valore che specifica la posizione di un dato all'interno della memoria di un calcolatore.

**Macchina di Turing** Un modello di elaborazione estremamente semplice che viene usato nell'informatica teorica per esplorare problemi di computabilità.

**Macchina virtuale** Un programma che simula il funzionamento di una CPU e che può essere realizzato in modo efficiente per un'ampia varietà di macchine reali. Il bytecode relativo a un programma può essere eseguito da qualsiasi macchina virtuale Java, indipendentemente dalla CPU utilizzata per eseguire la macchina virtuale stessa.

**Mappa** Una struttura di memorizzazione che associa oggetti che fungono da chiave a oggetti che hanno il ruolo di valori.

**Memoria secondaria** La memoria persistente, che mantiene il proprio contenuto anche in assenza di energia elettrica (ad esempio, un disco rigido).

**Mergesort** Vedi Ordinamento per fusione

**Metodo** Una sequenza di enunciati che ha un nome, può avere variabili parametro e può restituire un valore. Un metodo può essere invocato un numero qualsiasi di volte, con diversi valori attribuiti ai suoi parametri.

**Metodo astratto** Un metodo dotato di nome, tipi dei parametri e tipo del valore restituito, ma privo di implementazione.

**Metodo d'accesso** Un metodo che accede a un oggetto senza modificarlo.

**Metodo predefinito (o di default)** Un metodo non statico di cui si descrive l'implementazione in un'interfaccia.

**Metodo di esemplare** Un metodo avente un parametro implicito, cioè un metodo che viene invocato mediante un esemplare di una classe.

**Metodo main** Il primo metodo invocato quando viene eseguita un'applicazione Java.

**Metodo modificatore** Un metodo che può modificare lo stato di un oggetto.

**Metodo ricorsivo** Un metodo che invoca se stesso con dati più semplici. Deve gestire i casi più semplici senza invocare se stesso.

**Metodo statico o di classe** Un metodo privo di parametro隐式的.

**Mock** Vedi Oggetto semplificato.

**Modulo** L'operatore % che calcola il resto di una divisione intera.

**Notazione O-grande** La notazione  $g(n) = O(f(n))$ , che indica che la funzione  $g$  cresce rispetto a  $n$  con un comportamento limitato superiormente dalla crescita della funzione  $f$ . Ad esempio,  $10n^2 + 100n - 1000 = O(n^2)$ .

**Notazione polacca inversa (RPN)** Uno stile per la scrittura di espressioni in cui gli operatori vengono indicati dopo i propri operandi. Ad esempio,  $2 \ 3 \ 4 \ * \ +$  equivale a  $2 + 3 * 4$ .

**Numeri di Fibonacci** La sequenza di numeri 1, 1, 2, 3, 5, 8, 13, ..., in cui ogni elemento è la somma dei suoi due predecessori.

**Numeri pseudocasuali** Una sequenza di numeri che sembrano essere casuali ma sono, in realtà, generati mediante una formula matematica.

**Numero in virgola mobile** Un numero che può avere una parte frazionaria.

**Numero intero** Un numero che non può avere una parte frazionaria.

**Numero letterale** Un valore fisso all'interno di un programma che viene scritto esplicitamente sotto forma di numero, come -2 o 6.02214115E23.

**Numero magico** Un numero che compare in un programma senza alcuna spiegazione.

**Oggetto** Un valore il cui tipo è una classe.

**Oggetto anonimo** Un oggetto che non viene memorizzato in una variabile.

**Oggetto semplificato (mock)** Un oggetto che viene utilizzato durante il collaudo di un programma per sostituire un altro oggetto avente un comportamento simile; solitamente l'oggetto *mock* è di più semplice realizzazione o particolarmente progettato per agevolare il collaudo.

**Operatore** Un simbolo che rappresenta un'operazione matematica o logica, come + o &&.

**Operatore binario** Un operatore che richiede due argomenti o operandi, come l'operatore + nell'espressione  $x + y$ .

**Operatore booleano o logico** Un operatore che può essere applicato a variabili booleane. Java dispone di tre operatori booleani: &&, || e !.

**Operatore new** Un operatore che crea nuovi oggetti.

**Operatore relazionale** Un operatore che confronta due valori fornendo un risultato di tipo booleano.

**Operatore ternario** Un operatore dotato di tre operandi. Java ha un solo operatore ternario,  $a ? b : c$ .

**Operatore unario** Un operatore che ha un solo operando.

**Ordinamento lessicografico** L'ordinamento di stringhe in modo simile a quanto avviene in un dizionario, ignorando tutte le coppie di caratteri corrispondenti che siano identici e confrontando i primi caratteri di ciascuna stringa che differiscono tra loro. Ad esempio, nell'ordinamento lessicografico "orbit" precede "orchid". Notate che in Java, diversamente da quanto avviene in un dizionario, l'ordinamento è sensibile alla differenza tra maiuscole e minuscole: Z precede a.

**Ordinamento per fusione** Un algoritmo di ordinamento che ordina due metà di una struttura di dati, per poi fonderle insieme.

**Ordinamento per selezione** Un algoritmo di ordinamento che cerca e rimuove ripetutamente l'elemento di valore minimo, finché non rimangono più elementi.

**Ordinamento quicksort** Un algoritmo di ordinamento, solitamente veloce, che sceglie un elemento (denominato "pivot"), partiziona la sequenza da ordinare in due parti contenenti, rispettivamente, gli elementi minori e gli elementi maggiori del pivot, quindi ordina ricorsivamente tali sotto-sequenze.

**Pacchetto (package)** Un insieme di classi tra loro correlate. Per accedere a una o più classi contenute in un pacchetto si usa l'enunciato `import`.

**Pacchetto di prove (test suite)** Un insieme di casi di prova per un programma.

**Pannello** Un componente dell'interfaccia grafica privo di aspetti visibili, solitamente utilizzato per raggruppare altri componenti.

**Pannello dei contenuti** In Swing, quella porzione di frame che contiene i componenti dell'interfaccia utente.

**Parametro esplicito** In un metodo, un parametro diverso dall'oggetto con cui il metodo è stato invocato.

**Parametro implicito** L'oggetto con cui viene invocato un metodo. Ad esempio, nell'invocazione `x.f(y)`, l'oggetto `x` è il parametro implicito del metodo `f`.

**Parola riservata** Una parola che ha uno speciale significato in un linguaggio di programmazione e che, quindi, non può essere utilizzata come nome dai programmatore.

**Passaggio dei parametri** L'azione di definizione di espressioni che fungano da argomenti di un metodo nel momento in cui questo viene invocato.

**Percorso (di un file o cartella)** La sequenza di nomi di cartelle che descrive come raggiungere un file o una cartella a partire da una determinata posizione nel file system.

**Permutazione** Una disposizione degli elementi appartenenti a un insieme di valori.

**Pila** Una struttura di memorizzazione da cui gli elementi vengono estratti con una strategia "l'ultimo entrato è il primo a uscire". Gli elementi possono essere aggiunti ed eliminati in una sola posizione, denominata "cima" della pila.

**Pila delle invocazioni (call stack)** L'insieme ordinato di metodi che, in un certo istante, sono stati invocati ma non hanno ancora terminato la propria esecuzione; in cima alla pila c'è il metodo attualmente in esecuzione, mentre in fondo c'è il metodo `main`.

**Pila di esecuzione (run-time stack)** La struttura che memorizza le variabili locali di tutti i metodi invocati durante l'esecuzione di un programma.

**Polimorfismo** Selezione, in base all'effettivo tipo del parametro implicito, di uno tra vari metodi aventi lo stesso nome.

**Precedenza degli operatori** La regola che stabilisce quale operatore debba essere valutato per primo. Ad esempio, in Java l'operatore `&&` ha la precedenza sull'operatore `||`, per cui l'espressione `a || b && c` viene valutata come `a || (b && c)`.

**Principio di sostituzione** Il principio secondo il quale un esemplare di una sottoclasse può essere usato al posto di un esemplare di una sua superclasse.

**Progettazione orientata agli oggetti** Progettazione di un programma che avviene mediante la definizione di oggetti, delle loro proprietà e delle loro relazioni.

**Programma per calcolatore** Una sequenza di istruzioni che può essere eseguita da un calcolatore.

**Programma per console** Un programma Java privo di interfaccia grafica. Un programma per console legge i dati in ingresso dalla tastiera e visualizza i dati prodotti in uscita sulla finestra di terminale (*console*) in cui è stato eseguito.

**Programmazione** La progettazione e realizzazione di programmi per calcolatore.

**Prompt** Un messaggio che invita l'utente a fornire dati in ingresso.

**Pseudocodice** Una descrizione ad alto livello delle azioni intraprese da un programma o da un algoritmo, usando una sintassi informale, mista tra un linguaggio naturale e un linguaggio di programmazione.

**Quicksort** Vedi Ordinamento quicksort.

**Raccolta (o collezione)** Una struttura di memorizzazione dei dati che consente di inserire, rimuovere o ritrovare elementi.

**Redirezione** Collegamento dell'ingresso o dell'uscita di un programma a un file, anziché alla tastiera o, rispettivamente, allo schermo.

**Rete (di calcolatori)** Un sistema di elaborazione costituito da calcolatori e altri dispositivi interconnessi.

**Ricerca binaria** Un veloce algoritmo che cerca un valore all'interno di un array ordinato, dimezzando a ogni passo la porzione di array in cui viene effettuata la ricerca.

**Ricerca dinamica del metodo** Selezione del metodo da invocare durante l'esecuzione. In Java, la ricerca dinamica del metodo effettua la selezione sulla base della classe dell'oggetto che funge da parametro implicito.

**Ricerca lineare o sequenziale** Ricerca di un oggetto all'interno di un contenitore (come un array o una lista) che avviene mediante l'ispezione di ciascun suo elemento, uno dopo l'altro.

**Ricevitore di eventi** Un oggetto che, quando accade un evento, riceve una notifica dalla sorgente dell'evento stesso.

**Ricorsione** Una strategia che calcola un risultato mediante la scomposizione dei dati da elaborare in valori più semplici, applicandovi poi la medesima strategia.

**Ricorsione mutua** Metodi cooperanti che si invocano reciprocamente.

**Riferimento a oggetto** Un valore che rappresenta la posizione in memoria di un oggetto. In Java, una variabile il cui tipo sia una classe contiene in realtà un riferimento a un oggetto, esemplare di tale classe.

**Riferimento null** Un riferimento che non si riferisce ad alcun oggetto.

**Riga dei comandi** La riga in cui l'utente digita i comandi necessari per eseguire un programma nel sistema operativo DOS, Windows o Unix; è composta dal nome del programma seguito da suoi eventuali argomenti.

**Script per shell** Un file contenente comandi per l'esecuzione di programmi e la manipolazione di file. Digitando sulla riga dei comandi il nome del file contenente lo script per shell si provoca l'esecuzione dei comandi in esso contenuti.

**Sentinella** Un valore di ingresso che non viene usato come reale valore da elaborare, ma per segnalare la fine dei dati in ingresso.

**Sequenza di escape** Una sequenza di caratteri, come `\n` o `\\"`, che inizia con un carattere di escape.

**Shadowing** Nascondere una variabile definendone un'altra con lo stesso nome.

**Sintassi** Regole che definiscono la composizione delle istruzioni in un particolare linguaggio di programmazione.

**Sistema operativo** Il software che esegue i programmi applicativi e fornisce loro servizi (come il *file system*).

**Software** Le istruzioni e i dati che regolano il funzionamento di un calcolatore o di un altro dispositivo.

**Sorgente di eventi** Un oggetto che è in grado di inviare ad altri oggetti notifiche relative a specifici eventi.

**Sottoclasse** Una classe che eredita variabili e metodi da una superclasse, con la possibilità di aggiungere variabili di esemplare e di aggiungere o sovrascrivere metodi.

**Sovraccarico (overloading)** Attribuire più di un significato al nome di un metodo.

**Sovrascrittura (overriding)** Ridefinizione di un metodo all'interno di una sottoclasse.

**Specificatore di accesso** Una parola riservata che indica le modalità di accesso a una caratteristica; ad esempio, `public` o `private`.

**Stato** Il valore attuale di un oggetto, determinato dall'azione cumulativa di tutti i metodi che sono stati con esso invocati.

**Stringa** Una sequenza di caratteri.

**Superclasse** Una classe da cui una classe più specifica (denominata sottoclasse) può ereditare.

**Swing** Un insieme di strumenti Java per la realizzazione di interfacce grafiche per l'interazione con l'utente.

**Tabella hash** Una struttura in cui gli elementi vengono messi in corrispondenza con posizioni all'interno di un array in base ai valori loro assegnati da una funzione di hash.

**Tipo** Un insieme di valori dotato di nome e delle operazioni che con essi si possono svolgere.

**Tipo booleano** Un tipo di dato che può assumere due soli valori: true e false.

**Tipo parametrico** Un parametro presente nella dichiarazione di una classe generica o di un metodo generico; viene sostituito da un tipo effettivo.

**Tipo primitivo** In Java, un tipo numerico o il tipo boolean.

**Token** Una sequenza di caratteri consecutivi all'interno di una sorgente di dati che compongono, nel loro insieme, un'informazione significativa per l'analisi dei dati in ingresso. Ad esempio, un token può essere una sequenza di caratteri diversi dal carattere di spaziatura.

**Traccia della pila di esecuzione (stack trace)** La visualizzazione della pila di esecuzione, che elenca tutte le invocazioni di metodi in attesa di riprendere la propria esecuzione.

**UML (Unified Modeling Language)** Una notazione che consente di specificare, visualizzare, costruire e documentare tutti gli aspetti di un sistema software.

**Unicode** Una codifica standard che assegna valori di due byte ai caratteri usati nelle lingue scritte di tutto il mondo. Java memorizza tutti i caratteri usando i rispettivi codici Unicode.

**URL (Uniform Resource Locator)** Nel World Wide Web, un riferimento a una risorsa informativa, come una pagina HTML o un'immagine.

**Valore restituito** Il valore restituito da un metodo mediante un enunciato return.

**Valutazione in cortocircuito** Valutazione di una sola porzione di un'espressione, nel caso in cui la parte rimanente non possa modificare il risultato.

**Variabile** In un programma, un simbolo che identifica una locazione di memoria che può contenere diversi valori.

**Variabile di esemplare** Una variabile, definita in una classe, di cui esiste una copia, con un proprio valore, per ciascun esemplare della classe.

**Variabile locale** Una variabile che ha un blocco di enunciati come ambito di visibilità.

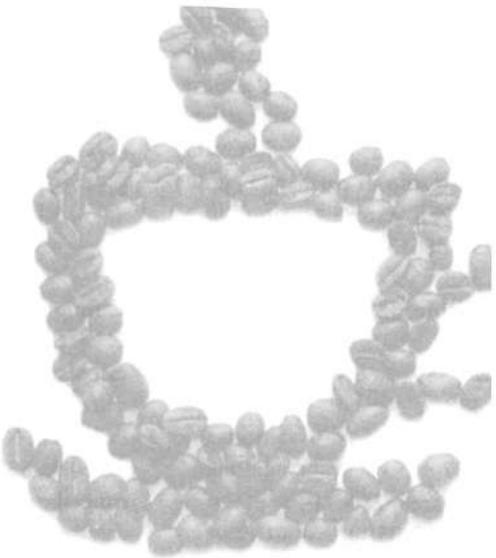
**Variabile non inizializzata** Una variabile che non ha ricevuto alcun valore iniziale. In Java, l'utilizzo di una variabile locale non inizializzata costituisce un errore di sintassi.

**Variabile parametro** In un metodo, una variabile che viene inizializzata con un valore fornito nel momento in cui il metodo viene invocato.

**Variabile statica** Una variabile di cui esiste una sola copia per l'intera classe, a cui possono accedere, eventualmente modificandone il valore, tutti i metodi della classe stessa.

**void** Una parola riservata usata per segnalare che un metodo non restituisce alcun valore.

# Indice analitico



I numeri di pagina che iniziano con W si riferiscono al Capitolo 15, disponibile sul sito web dedicato al libro.

## Simboli

;, 13, 16  
“, 14, 176  
. , 446  
. , 14  
+, 38, 150, 176-177  
-, 38, 150  
->, 539-540  
★, 38, 150, 445  
\*/ , 39, 95  
/, 38, 150-152  
//, 39  
/\*, 39  
/\*\*\*, 94-96  
\_, 38, 143  
\$, 38, 542  
@, 94-96  
%, 151-152, 161-162, 588-589  
\, 177-178, 579

\\", 178  
? , 201-202, 689-690  
; , 201-202  
<> , 396-397, W2-W16  
< , 285-286  
<= , 203  
<< , 789-790  
> , 285-286  
>= , 203  
>> , >>> , 789-790  
= , 40, 203  
== , 203  
! , 236  
!= , 203  
& , 788-789, W9  
&& , 235-237  
| , 788-789  
^ , 788-789  
|| , 235-237

[] , 338-339  
 $\Omega$  , 689-690  
 $\Theta$  , 689-690

## A

Altair 8800, 452  
ambiente di sviluppo, 9-10  
ambito di visibilità, 225-227  
Analytical Engine, 705  
Andersen, Marc, 507  
ANSI, 740  
API (application programming interface), 53-55  
Apple II, 452  
applet, 6  
applicazione grafica, 67-77  
architettura di un PC, 4-5  
argomento, *Vedi* parametro  
Ariane, 620

**A**  
**ArithmeticException**, 604  
**ARPANET**, 507  
array, 338-347  
algoritmi, 351-359  
bidimensionale, 376-386  
di tipo generico, W12-W13  
dichiarazione, 338  
length, 341, 396  
multidimensionale, 386  
non inizializzato, 345  
paralleli, 346-347  
riempito solo in parte, 343-344  
array list, *Vedi* vettore  
**ArrayList**, 386-397, 420, 730, W2  
add, 388  
costruttore, 387, 390  
get, 388  
remove, 389  
set, 388  
size, 388, 396  
**Arrays**, 357  
binarySearch, 714-715  
copyOf, 357-358  
sort, 362, 700, 713-714, 721-722  
toString, 681  
arrotondamento, 143, 785-786  
ASCII, 581-582  
assegnazione, 40-41, 64, 144, 153-154,  
157-159, 203  
asserzione, 613-614  
auto-boxing, 391-392  
auto-unboxing, 391-392  
AutoCloseable, 610, 614  
AWT (abstract windowing toolkit),  
55

**B**

Babbage, Charles, 705  
backtracking, 661-667  
backup, 12  
BASIC (linguaggio), 452  
batch, 399-400, 592  
Berners-Lee, Tim, 507  
BigDecimal, 143, 149  
BigInteger, 143, 149  
black box, 88-89, 231

blocco di codice, 225-227  
BlueJ, 49-50, 58, 97, 111, 314, 450  
Booch, Grady, 418  
Boole, George, 235  
boolean, 142, 235-237, 391  
Boolean, 391  
break, 287-288  
breakpoint, 315-317, 632-633  
buffer overrun, 348  
bug, 325, 397  
byte, 142, 391  
Byte, 391

**C**

C (linguaggio), 159  
C++ (linguaggio), 7, 159, 423  
calcolatore, 2-6  
calcolo ubiquo, 5-6  
call stack, 633  
callback, 535-537, 547  
cancellazione dei tipi, W11-W13  
carattere, 176, 178, 581-582, 777-778  
di controllo, 778  
di spaziatura, 582-583  
jolly, W9-W10  
cartella, 11, 446  
case, 218-219  
caso di prova, 231-232  
caso limite, 232  
cast, 70, 154, 502-503, 526  
catch, 604-608  
Cerf, Vinton, 507  
char, 142, 178, 391, 583  
Character, 391, 583-584  
chiave (in una mappa), 731, 745-748  
chiave privata/pubblica, 602-603  
CIA Fact Book, 584-585  
ciclicità, 397  
ciclo, 258-261  
algoritmi, 292-296  
annidato, 304-307  
controllo di un, 271-272  
definito, 272  
do, 280-281  
e mezzo, 286-288  
for, 271-274

for esteso, 349-350, 389-390  
indefinito, 272  
infinito, 264  
limiti, 278  
progettazione, 297-303  
while, 258-261  
cifratura, 602-603  
 cifre significative, 142  
**class**, 13  
class file, 10-11  
classe, 13, 35  
anonima, 543  
astratta, 485-487  
che implementa interfaccia, 517-518  
concreta, 486  
conversione a interfaccia, 524-525  
conversione da interfaccia, 526  
di collaudo, 111  
documentazione, 95-96  
final, 487  
generica, 387, W2-W6  
immutabile, 419-420  
interfaccia pubblica, 44, 90-97  
interna, 74, 541-542, 548-550  
involturco, 390-392  
progettazione, 416-451  
realizzazione, 98-101  
clonazione, 531-534  
**Cloneable**, 533-534, W9  
**CloneNotSupportedException**,  
533-534  
cloud computing, 67  
coda, 731, 756-757  
prioritaria, 731, 757-758  
codice,  
aperto, 565-566  
di hash, 500, 739, 753-755  
macchina, 6  
spaghetti, 228-229  
codifica dei caratteri, 581-582  
coerenza, 422  
coesione, 417-418  
collaudo, 56-58  
black-box, 231-232  
di unità, 110-111, 450-451  
regressivo, 397-399  
white-box, 231-232

- Collection, 730  
Collections, 713-715  
collezione, 730  
collisione, 753  
Color, 75-76  
commento, 39-40, 94-96  
**Comparable**, 514, 529-531, 540-541, 714-716, 741, 756, W8-W9  
**Comparator**, 523, 716-718  
compilatore, 6-7, 10-11, 16  
complemento a due, 786-787  
componente grafico, 69-72  
computer, 2-6  
concatenazione, 176-177  
confronto numerico, 203  
copia superficiale, 533  
console, 9-10  
contatore, 271-274  
conto grafico, *Vedi Graphics*, *Graphics2D*  
**continue**, 288-289  
coordinate grafiche, 71, 124  
copertura del codice, 231-232  
corpo di ciclo, 258  
corpo di metodo, 91  
cortocircuito (valutazione booleana), 239  
costante, 144-146, 521-522, 549  
costruttore, 92-93, 98-100  
    **super()**, 480-481  
    **this()**, 122-123  
costruzione di oggetto, 48-50  
CPU (Central Processing Unit), 2, 4, 6, 7  
cursor, *Vedi iteratore*
- D**  
DARPA, 244, 507  
De Morgan, Augustus (legge di), 240  
debugger, 314-325, 633  
debugging, 315-315, 499  
    di ricorsione, 632-633  
decisione, *Vedi diramazione*  
decremento, 151  
**default**, 219, 523-524  
diagramma,
- di flusso, 196-197, 216, 220, 228-230, 259, 274, 281, 282, 305  
sintattico, 654-656  
**UML**, 418-419, 464-465, 469-470, 518, 537  
diamante (sintassi), 396-397, 732  
dichiarazione,  
    di array, 338  
    di metodo, 47  
    di variabile, 36, 86-87, 98  
**Difference Engine**, 705  
Dijkstra, Edsger, 231  
dipendenza tra classi, 418-419  
diramazione, 196-198  
    annidata, 219-222  
    multipla, 214-217  
    switch, 218-219  
directory, 11, 446  
disco rigido, 3-4  
dispositivi periferici, 4-5  
diversità (operatore), 203  
divide and conquer, 698  
divisione intera, 151-152, 155-156  
**do**, 280-281  
documentazione, 53-55, 94  
doppia precisione, 142  
**double**, 38, 142-143, 391  
**Double**, 391  
    **compare**, 531  
    **parseDouble**, 183, 586  
**Dr. Java**, 58  
dynamic method lookup, 482, 484-485
- E**  
eccezione, 17, 181-182, 241, 620  
    cattura, 604-608  
    chiusura di risorse, 609-610  
    controllo, 608-609  
    gestione, 604-611  
    lancio, 604-607  
    **printStackTrace**, 606  
    progetto, 610-611  
Eclipse, 314, 450  
ECMA, 740  
editor, 9-10
- effetto collaterale, 420-421  
**Ellipse2D**, 73-74  
**else**, 196-198, 214-217  
    sospeso, 225  
**ENIAC**, 5  
**enum**, 227-228  
enumerazione, 227-228  
enunciato, 13  
erasure, W11-W13  
ereditarietà, 464-468, 519-520, W9  
**Error**, 605  
errore, 16-17  
    di limiti, 340-341, 344-345  
    di sintassi, 16-17  
    logico, 17  
    per scarto di uno, 265-266  
**ESA**, 620  
escape, 177-178, 579  
espressione, 150, 155  
    canonica, 590-591  
    lambda, 539-540, 552, 717-718  
    relazionale, 238  
    spaziatura, 157  
etichetta, 288  
evento, 546, 556, 559-560, 563-564  
**Exception**, 605  
**extends**, 70, 469-470, 611, W8-W9
- F**  
Facebook, 507  
**false**, 235  
Fibonacci (sequenza di), 643  
FIFO (first-in, first-out), 756  
file, 11, 13  
    di classe, 10-11  
    di testo, 576-578  
    sorgente, 9-11, 13, 446, 576  
**File**, 576-578  
    **isDirectory**, 638  
    **listFiles**, 638  
**FileNotFoundException**, 577-578, 606  
**Files**, 591  
**final**, 145-146, 487, 549  
**finally**, 614  
finestra,

con cornice, 67-69, 72  
di dialogo, 182-183, 580-581  
di terminale, 9-10  
flag, 299  
flessibilità, 2  
float, 142, 144, 391  
**Float**, 391  
floating point, 38, 142-144, 158, 204, 588, 787-788  
flowchart, *Vedi* diagramma di flusso  
folder, 11, 446  
**for**, 271-274  
esteso, 349-350, 389-390  
formato (per `printf`), 161-162, 588-589  
frame, 67-69, 72  
free software, 565-566  
funzione di hash, 753-755  
garbage collector, 117  
generatore di numeri pseudocasuali, 310-312  
gerarchia di ereditarietà, 464, 466, 488-499  
gestione di eccezioni, 604-611  
gestione di eventi, 545-552

**G**

**GNU**, 565-566, 603  
Gosling, James, 6  
**GPL**, 565-566  
**Graphics**, 70, 559  
**Graphics2D**, 70-75  
Green (linguaggio), 6  
grep, 590  
Gutenberg (progetto), 507

**H**

hardware, 2  
hard disk, 3-4  
**HashMap**, 730, 745-748, 753-755  
**HashSet**, 730, 739-744, 753-755  
Hewlett-Packard, 759  
Hoff, Marcian E. "Ted", 452  
HTML, 94-98, 576, 740

**I**

**IBM**, 66-67  
**IDE** (integrated development environment), 9-10  
**IEEE**, 158  
**IEEE 754** (standard), 787-788  
**IETF**, 740  
**if**, 196-198, 214-217  
**IllegalArgumentException**, 604, 606, 608, 611  
implementazione di interfaccia, 517-518, 551  
**implements**, 517  
**import**, 55, 74, 438, 444-445  
importazione, 55-56, 444-445 statica, 438  
incapsulamento, 88-89  
incremento, 151  
**IndexOutOfBoundsException**, 608  
indice, 339  
information hiding, 88-89  
inizializzazione, 36-37, 41, 117, 437-438  
mancata, 42  
predefinita, 117  
inner class, *Vedi* classe interna  
**InputMismatchException**, 586  
insertion sort, 690-692  
insieme, 731, 739-744  
**instanceof**, 502-505  
int, 36-37, 142-143, 391  
**Integer**, 391  
compare, 531  
MAX\_VALUE, 142  
MIN\_VALUE, 142  
parseInt, 183, 586  
Intel, 158, 686  
intelligenza artificiale, 244  
**IntelliJ**, 314  
interfaccia, 514-524, 745  
conversione a classe, 526  
conversione da classe, 524-525  
di smistamento, 535-537, 547  
funzionale, 539-540  
invocazione di metodi, 525  
interfaccia grafica, 545

interfaccia pubblica, 44-47, 90-97, 417-418  
**interface**, 515-516  
Internet, 5, 6-7, 507  
invocazione di metodo, 14, 44-47, 121-122, 422-426  
involucro, 390-392  
**IOException**, 580, 608  
ISO, 740  
**Iterator**, 737, 743  
iteratore, 734-737, 742  
iterazione, *Vedi* ciclo

**J**

Jacobson, Ivar, 418  
Java (linguaggio), 6-8  
Java Collections Framework, 730  
**java.applet**, 444  
**java.awt**, 55, 71, 444, 448  
**java.awt.geom**, 74  
**java.io**, 444, 577  
**java.lang**, 56, 444, 445  
**java.math**, 149  
**java.net**, 444, 580  
**java.nio.file**, 591  
**java.sql**, 444  
**java.util**, 160, 387, 444  
**javadoc**, 94-98  
**javax.swing**, 68, 444, 556  
**JButton**, 547, 552-555  
addActionListener, 547  
**JComponent**, 69-70  
addKeyListener, 564  
addMouseListener, 560  
paintComponent, 70, 124-125, 557, 559  
repaint, 557  
requestFocus, 564  
setFocusable, 564  
**JFileChooser**, 580-581  
**JFrame**, 67-68  
add, 72, 553  
**EXIT\_ON\_CLOSE**, 68  
setDefaultCloseOperation, 68  
setSize, 67  
setTitle, 68

**K**  
**JLabel**, 552  
**JOptionPane**, 182-183, 422  
**JPanel**, 552-553  
**JUnit**, 450-451  
**JVM** (Java virtual machine), 7, 10-11, 117

**K**  
**Kahn**, Bob, 507  
**KeyAdapter**, 565  
**KeyListener**, 563-564  
**KeyStroke**, 564

**L**  
**lambda** (espressione), 539-540, 552, 717-718  
**letterale**, 142-143, 176  
**libreria**, 7-8, 10-11, 53-55, 443-444  
**LIFO** (last-in, first-out), 755  
**limiti** in un ciclo, 278-279  
**Line2D**, 74  
**linguaggio di programmazione**, 6  
**LinkedList**, 730, 734-735, 756-757  
**Linux**, 7, 686,  
**List**, 730  
**lista**, 730  
  concatenata, 730, 733-735  
  doppiamente concatenata, 736  
**listener**, 546-552, 553, 555-556  
**ListIterator**, 735-737, 742  
**Logger**, 234-235  
**logging**, 234-235  
**long**, 142-143, 391  
**Long**, 391  
**loop**, *Vedi* ciclo

**M**  
**macchina di Turing**, 652-653  
**macchina virtuale**, 7, 10-11, 117  
**Macintosh**, 7, 581  
**main**, 13-14, 182, 416  
**mainframe**, 66  
**Map**, 730, 745-748

**mappa**, 731, 745-748  
**Mark II**, 325  
**Math**, 145, 445  
  E, 145  
  funzioni, 153  
  PI, 145  
  pow, 152-153  
  round, 154  
  sqrt, 152-153, 434-435  
**memoria**, 3-4  
**mergesort**, 692-698  
**metodo**, 13, 34, 44-47  
  astratto, 486  
  con numero variabile di parametri, 347  
**corpo**, 91  
**d'accesso**, 51, 419-420  
**definizione**, 90-92, 100-101  
**di default**, 522-524  
**di esemplare**, 159  
**dichiarazione**, 47  
**documentazione**, 94-96  
**ereditato**, 469-470  
**final**, 487  
**generico**, W6-W8  
**invocazione**, 44, 422-426  
**modificatore**, 51-52, 419-420  
**parametro**, 45  
**predefinito**, 522-524  
**progettazione**, 417-421  
**ricorsivo**, 629-631, 641-642  
**sovraffaccarico**, 47, 478-479, 642  
**sovrascritto**, 469-470, 474-475, 478-479  
**statico**, 159, 434-437, 522  
**valore restituito**, 45-46  
**Microsoft**, 67, 507, 740  
**MITS Electronics**, 452  
**Mycin**, 244  
**matrice**, 376-384  
**mock**, 544-545  
**Monte Carlo**, 312-313  
**Morris**, Robert, 348  
**Mosaic**, 507  
**MouseAdapter**, 565  
**MouseListener**, 559-565

**N**  
**NaN** (not a number), 787  
**NASA**, 620  
**Naughton**, Patrick, 6  
**NCSA**, 507  
**Netbeans**, 314  
**Netscape**, 67, 507  
**new**, 49-50, 64, 338  
**newline** (carattere), 162, 178, 579, 582, 588, 778  
**Nicely**, Thomas, 158  
**nome**, 38-39  
**NoSuchElementException**, 583, 586, 606, 736  
**null**, 117, 207  
**NumberFormatException**, 606  
**numero**,  
  binario, 783-784  
  casuale, 310  
  esadecimale, 788  
  frazionario, 38  
  in complemento a due, 786-787  
  in virgola mobile, 38, 142, 787-788  
  infinito, 787  
  intero, 36, 142  
  letterale, 142-143  
  magico, 149-150, 590  
  NaN, 787  
  pseudocasuale, 312  
  triangolare, 628-631

**O**  
**O-grande** (notazione), 688-690  
**Object**, 445, 499-504  
  clone, 531-534  
  equals, 499, 501-502, 506, 741, 755  
  erasure, W11-W13  
  hashCode, 499, 741, 753-755  
  toString, 499-501, 505-506  
**Objects.hash**, 754  
**oggetto**, 34-35, 63  
  anonimo, 543  
  confronto, 205-207  
  clonazione, 531-534  
  costruzione, 48-50

di sottoclasse, 470  
**equals**, 206  
 semplificato, 544-545  
 stato di, 86-87  
 omega (notazione), 689-690  
 open source, 565-566  
 operatore, 779-780  
     a bit, 788-790  
     aritmetico, 38, 150, 157-159  
     assegnazione, 40  
     booleano, 235-237  
     condizionale, 201-202  
     decremento, 151  
     incremento, 151  
     `instanceof`, 502-504  
     `new`, 49-50, 64, 338  
     precedenza, 150, 236, 779-780  
     relazionale, 203  
     scorrimento, 789-790  
     resto, 151-152  
 Oracle, 566  
 ordinamento, 362  
     di Shell, 718-722  
     lessicografico, 205  
     per fusione, 692-698  
     per inserimento, 690-692  
     per selezione, 680-688  
     quicksort, 698-700, 713  
 Otto Regine, 661-667  
**OutOfMemoryError**, 608  
 overflow, 142-143, 149, 620, 785  
 overload, *Vedi* sovraccarico  
 override, *Vedi* sovrascrittura

**P**

pacchetto, 8, 55-56, 443-446  
     accesso di, 448  
     di prove, 397-399  
     nome di, 445-446  
     predefinito, 444  
 package, 444  
 padding, 588  
 pannello, *Vedi* JPanel  
 parametro, 14, 45  
     di costruzione, 50

di tipo, 387  
 esplicito, 120  
 implicito, 120  
 in numero variabile, 347  
     `this`, 121-122  
 parola riservata, 38, 781-782  
**Paths**, 591  
 PC (personal computer), 3, 452-453  
 Pentium, 158  
 periferiche, 4-5  
 permutazione, 649  
 piattaforma, 7  
 pila, 731, 755-756  
     delle invocazioni, 633  
     di esecuzione, 182, 756  
 pirateria digitale, 270  
 pivot, 699-700  
 pixel, 68, 307-310  
**Point2D**, 74  
 polimorfismo, 480-483  
 portabilità, 7  
 precedenza tra operatori, 150, 236,  
     779-780  
 precisione numerica, 142-143, 149  
 prestazioni di algoritmi, 683-692,  
     695-698, 706-712  
 principio di sostituzione, 465  
**PrintStream**,  
     `print`, 15, 160, 162  
     `printf`, 161, 178, 588-589  
     `println`, 14-15, 47, 162  
**PrintWriter**, 576-579, 581-582  
     `close`, 576-578  
     `print/printf/println`, 576-578  
**PriorityQueue**, 730, 757-758  
**private**, 13, 87, 434, 448  
 problema della terminazione, 652-653  
 programma, 2, 12-15  
     di collaudo, 110-112  
     di simulazione, 310  
 programmazione, 2, 8-12  
     generica, W1-W16  
 prompt, 160, 182  
**protected**, 487-488  
 pseudocodice, 20-21, 23-25  
**public**, 13, 91, 145, 434, 448, 517, 521

pulsante grafico, *Vedi* JButton  
 punto base, 75  
 punto di arresto, 315-317, 632-633

**Q**

**Queue**, 730, 756-757  
 quicksort, 698-700, 713

**R**

raccolta, 730  
**Random**, 310-312  
 Raymond, Eric, 566  
**Rectangle**, 49-50, 54-55  
 redirezione, 285-286, 398-399  
 regular expression, 590-591  
 restituito (valore), 45-46, 88  
 resto, 151-152, 157  
 rete di calcolatori, 4-5  
**return**, 88  
 RGB (red, green, blue), 75-76  
 ricerca,  
     binaria, 353, 702-704  
     dinamica del metodo, 482, 484-485  
     lineare o sequenziale, 352-353,  
         700-702  
 ricevitore di eventi, 546-552, 553,  
     555-556  
 ricorsione, 628-673  
     efficienza, 643-648  
     infinita, 632  
     mutua, 654-661  
 riferimento, 63-65  
     ad array, 342  
     confronto, 205-207  
     `equals`, 206, 501-502  
     `null`, 117  
     `this`, 120-122  
 riga dei comandi, 591-594  
 riutilizzo del codice, 514, 518  
 Rivest, Ron, 602  
 RPN, 758-759  
 RSA, 602-603  
 Rumbaugh, James, 418  
**RuntimeException**, 608

**S**

Scanner, 160, 420, 576-578, 581  
 close, 576  
 hasNext, 582-583  
 hasNextDouble, 241  
 hasNextInt, 241  
 hasNextLine, 584, 587  
 next, 177, 582-583, 587  
 nextDouble, 160, 587  
 nextInt, 160, 587  
 nextLine, 584, 587-588  
 per un file, 576-578  
 per una stringa, 579, 585  
 useDelimiter, 583, 590  
 scatola chiusa o nera, 88-89, 231  
 Scratch (linguaggio), 453  
 script, 399-400, 592  
 selection sort, 680-688  
 sentinella, 282-285  
 Set, 730  
 Shamir, Adi, 602  
 shell, 399-400  
 Shell sort, 718-722  
 shift, 789-790  
 short, 142, 391  
 Short, 391  
 simulazione, 310  
 single step, 315-316  
 singola precisione, 142  
 software, 2  
 libero, 565-566  
 sorgente di eventi, 546  
 sottoclasse, 464, 468-474  
 sottostringa, 179-180,  
 source file, 9-11, 13, 446, 576  
 sovraccarico, 47, 478-479, 642  
 sovrascrittura, 469, 474-475, 478-479  
 SSA (social security administration),  
 598  
 SSD (solid-state drive), 3  
 stack, *Vedi* pila  
 Stack, 730, 756  
 stack trace, 182  
 stack overflow, 632  
 Stallman, Richard, 565-566  
 standardizzazione, 740-741

static, 13, 145, 433-438, 522  
 step (into/over), 316-317  
 storyboard, 289-291  
 streaming, 507  
 String, 13, 38, 176-180  
 charAt, 178, 583  
 compareTo, 205  
 equals, 204-205, 209  
 hashCode, 753-754  
 immutabile, 419-420  
 length, 44, 176, 396  
 operatore ==, 209  
 replace, 46  
 replaceAll, 591  
 split, 591  
 substring, 179-180  
 toUpperCase, 44  
 trim, 585  
 stringa, 14, 176-180, 588  
 concatenazione, 176-177  
 letterale, 176  
 sottostringa, 179-180  
 vuota, 176  
 StringIndexOutOfBoundsException, 182  
 Stuxnet, 349  
 Sun Microsystems, 6, 740  
 super, 475, 479-481, W10  
 superclasse, 464, 473-474  
 universale Object, 499-504  
 Swing, 68-69  
 switch, 218-219  
 System, 14-15  
 currentTimeMillis, 683-684  
 in, 160  
 out, 14-15, 421

**T**

tabella, 304, 376-384  
 hash, 731, 739-744  
 tabulazione (carattere), 201, 582, 778  
 TCP/IP, 507  
 temporizzatore, 556-559  
 test suite, 397-399, 450-45  
 testing, *Vedi* collaudo  
 Therac, 400  
 theta (notazione), 689-690

this, 120-123, 472  
 throw, 604-607  
 Throwable, 605, 611  
 throws, 577-578, 608-609  
 Timer, 556  
 tipo (di dato), 37-38  
 booleano, 235-237  
 enumerativo, 227-228  
 grezzo (raw), W11-W13  
 interfaccia, 514-520  
 numerico, 142  
 parametrico, 387, 540-541  
 primitivo, 142  
 token, 656  
 Torri di Hanoi, 668-673  
 trabocco, 142-143, 149, 620, 785  
 traccia dell'esecuzione, 113-116,  
 223-224, 266-269  
 transistor, 3  
 TreeMap, 730, 745-748  
 TreeSet, 730, 739-744  
 true, 235  
 try, 604-608  
 con indicazione di risorse, 609-610  
 finally, 614

**U**

uguaglianza, 203-207  
 UML (linguaggio), 418-419, 518  
 Unicode, 142, 178, 183, 581-582,  
 777-778  
 unit testing, 110-111, 450-451  
 unità centrale di elaborazione (CPU),  
 2, 4, 6, 7  
 Univac, 66  
 UNIX, 7, 446, 565, 590  
 URL, 579-580  
 UTF-8, 581-582

**V**

valore,  
 booleano o logico, 142  
 in una mappa, 731, 745-748  
 restituito da un metodo, 45-46, 88  
 sentinella, 282-285

variabile, 36-39  
ambito di visibilità, 225-227  
array, 338-339  
assegnazione, 40-41  
booleana, 235-237  
costante, 144-146  
di esemplare, 86-87, 98, 433-434,  
    436-437, 470-473  
di tipo, W2-W3  
dichiarazione, 36-37  
in un ciclo for, 279-280  
inizializzazione, 36-37, 41, 117  
ispezione, 315-316  
locale, 116-117, 225-227  
nome, 38-39, 43  
non inizializzata, 42  
oggetto, 63-65  
parametro, 94, 102, 117  
`protected`, 487-488

riferimento, 63-65  
statica, 433-434  
utilizzo, 37  
vettore, 386-395  
    algoritmi per, 392  
    ArrayList, 386-389  
    copiaatura, 390  
        for esteso, 389-390  
    vincolo per tipo parametrico,  
        W8-W9  
virgola mobile, 38, 142-144, 158,  
    204, 588, 787-788  
virus, 348-349  
visibilità, 225-227  
VisiCalc, 452  
voice-over-IP, 507  
`void`, 47, 97  
voto elettronico, 114-115

## W

W3C, 740  
`while`, 258-261  
    do, 280-281  
wildcard, W9-W10  
Wilkes, Maurice, 325  
Window, 448  
Windows, 7, 446, 579, 590, 740  
wrapper, 390-392  
WWW (World-Wide Web), 507,  
    740

## Z

Zimmermann, Phil, 603

